Developing Java Applications
Updated December, 2010
Note
Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 525.

Edition Notice
This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.
• To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
• To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
Contents

About this book ........................................... vii
Who should use this book ................................. vii

Chapter 1. Java application development for IBM data servers ............. 1
Supported drivers for JDBC and SQLJ ........................ 2
JDBC driver and database version compatibility .. 3

Chapter 2. Installing the IBM Data Server Driver for JDBC and SQLJ ........ 5
DB2Binder utility ........................................... 8
DB2LobTableCreator utility ................................. 15
Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties ..................................... 16
Special setup for accessing DB2 for z/OS servers from Java programs ........ 17
DB2T4XAlndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers ............................ 18

Chapter 3. JDBC application programming ..................................... 21
Example of a simple JDBC application ..................... 21
How JDBC applications connect to a data source .......... 23
How DB2 applications connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver ........... 25
Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ ....... 26
Connecting to a data source using the DataSource interface .................. 30
How to determine which type of IBM Data Server Driver for JDBC and SQLJ connectivity to use ..................... 32
JDBC connection objects ...................................... 33
Creating and deploying DataSource objects ................. 33
Java packages for JDBC support ............................. 34
Learning about a data source using DatabaseMetaData methods .......... 35
DatabaseMetaData methods for identifying the type of data source ........ 36
Variables in JDBC applications ................................ 37
JDBC interfaces for executing SQL ......................... 37
Creating and modifying database objects using the Statement.executeUpdate method .................. 38
Updating data in tables using the
PreparedStatement.executeUpdate method ............. 38
JDBC executeUpdate methods against a DB2 for z/OS server .......... 40
Making batch updates in JDBC applications ................. 41
Learning about parameters in a
PreparedStatement using ParameterMetaData methods .......... 44
Data retrieval in JDBC applications .......................... 45
Calling stored procedures in JDBC applications ............. 58
LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ .................. 66
ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ ............ 71
Distinct types in JDBC applications ......................... 73
Invocation of stored procedures with ARRAY
parameters in JDBC applications .......................... 74
Savepoints in JDBC applications ............................ 75
Retrieval of automatically generated keys in
JDBC applications .......................................... 76
Using named parameter markers in JDBC
applications .................................................. 80
Providing extended client information to the data source with IBM Data Server Driver for JDBC
and SQLJ-only methods ..................................... 83
Providing extended client information to the data source with client info properties ............ 84
Optimistic locking in JDBC applications .................... 87
XML data in JDBC applications ............................ 89
XML column updates in JDBC applications .................. 90
XML data retrieval in JDBC applications ................... 92
Invoking routines with XML parameters in
Java applications .......................................... 95
Java support for XML schema registration and
removal ........................................................ 96
Transaction control in JDBC applications .................. 98
IBM Data Server Driver for JDBC and SQLJ
isolation levels .............................................. 99
Committing or rolling back JDBC transactions .......... 99
Default JDBC autocommit modes ......................... 100
Exceptions and warnings under the IBM Data Server Driver for JDBC and SQLJ .................. 100
Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ .................. 103
Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ .................. 106
Retrieving information from a
BatchUpdateException ................................. 107
Handling an SQLException under the DB2 JDBC Type 2 Driver (deprecated) .......... 109
Handling an SQLWarning under the DB2 JDBC Type 2 Driver .......... 109
Disconnecting from data sources in JDBC applications ............. 110

Chapter 4. SQLJ application programming .................................... 113
Example of a simple SQLJ application ..................... 113
Connecting to a data source using SQLJ .................. 115
SQLJ connection technique 1: JDBC
DriverManager interface .................................. 115
SQLJ connection technique 2: JDBC
DriverManager interface .................................. 117
Example of enabling DB2 Database for Linux, UNIX, and Windows automatic client reroute support in Java applications ........................................... 219
Operation of automatic client reroute for connections to DB2 Database for Linux, UNIX, and Windows from Java clients ........................................... 220
Application programming requirements for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers ........................................... 223
Client affinities for DB2 Database for Linux, UNIX, and Windows ........................................... 223
Java client support for high availability for connections to IDS servers ........................................... 227
Configuration of IDS high-availability support for Java clients ........................................... 228
Example of enabling IDS high availability support in Java applications ........................................... 231
Operation of automatic client reroute for connections to IDS from Java clients ........................................... 232
Operation of workload balancing for connections to IDS from Java clients ........................................... 235
Application programming requirements for high availability for connections from Java clients to IDS servers ........................................... 236
Client affinities for connections to IDS from Java clients ........................................... 237
Java client support for high availability for connections to DB2 for z/OS servers ........................................... 240
Configuration of Sysplex workload balancing at a Java client ........................................... 242
Example of enabling DB2 for z/OS Sysplex workload balancing in Java applications ........................................... 244
Operation of Sysplex workload balancing for connections from Java clients to DB2 for z/OS servers ........................................... 246
Operation of automatic client reroute for connections from Java clients to DB2 for z/OS ........................................... 247
Application programming requirements for high availability for connections from Java clients to DB2 for z/OS servers ........................................... 248

Chapter 11. Java 2 Platform, Enterprise Edition ........................................... 249
Application components of Java 2 Platform, Enterprise Edition support ........................................... 249
Java 2 Platform, Enterprise Edition containers ........................................... 250
Java 2 Platform, Enterprise Edition Server ........................................... 250
Java 2 Platform, Enterprise Edition database requirements ........................................... 250
Java Naming and Directory Interface (JNDI) ........................................... 251
Java transaction management ........................................... 251
Example of a distributed transaction that uses JTA methods ........................................... 252
Setting the transaction timeout value for an XAResource instance ........................................... 256
Enterprise JavaBeans ........................................... 256

Chapter 12. JDBC and SQLJ connection pooling support ........................................... 259

Chapter 13. JDBC and SQLJ reference information ........................................... 261
Data types that map to database data types in Java applications ........................................... 261
Date, time, and timestamp values that can cause problems in JDBC and SQLJ applications ........................................... 267
Properties for the IBM Data Server Driver for JDBC and SQLJ ........................................... 270
Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products ........................................... 271
Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 servers ........................................... 290
Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IDS ........................................... 298
Common IBM Data Server Driver for JDBC and SQLJ properties for IDS and DB2 Database for Linux, UNIX, and Windows ........................................... 300
IBM Data Server Driver for JDBC and SQLJ properties for DB2 Database for Linux, UNIX, and Windows ........................................... 300
IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS ........................................... 302
IBM Data Server Driver for JDBC and SQLJ properties for IDS ........................................... 306
IBM Data Server Driver for JDBC and SQLJ configuration properties ........................................... 312
Driver support for JDBC APIs ........................................... 325
IBM Data Server Driver for JDBC and SQLJ support for SQL escape syntax ........................................... 353
SQLJ statement reference information ........................................... 354
  SQLJ clause ........................................... 354
  SQLJ host-expression ........................................... 354
  SQLJ implements-clause ........................................... 355
  SQLJ with-clause ........................................... 355
  SQLJ connection-declaration-clause ........................................... 357
  SQLJ iterator-declaration-clause ........................................... 357
  SQLJ executable-clause ........................................... 359
  SQLJ context-clause ........................................... 359
  SQLJ statement-clause ........................................... 360
  SQLJ SET-TRANSACTION-clause ........................................... 362
  SQLJ assignment-clause ........................................... 363
  SQLJ iterator-conversion-clause ........................................... 364
  Interfaces and classes in the sqlj.runtime package ........................................... 364
    sqlj.runtime.ConnectionContext interface ........................................... 365
    sqlj.runtime.ForUpdate interface ........................................... 370
    sqlj.runtime.NamedIterator interface ........................................... 370
    sqlj.runtime.PositionedIterator interface ........................................... 370
    sqlj.runtime.ResultSetIterator interface ........................................... 378
    sqlj.runtime.SQLNullException class ........................................... 386
    sqlj.runtime.StreamWrapper class ........................................... 386
    sqlj.runtime.UnicodeStream class ........................................... 387
  IBM Data Server Driver for JDBC and SQLJ extensions to JDBC ........................................... 387
    DBBatchUpdateException interface ........................................... 389
    DB2Administrator class ........................................... 390
About this book

This book describes DB2® for Linux, UNIX, and Windows support for Java. This support lets you access relational databases from Java application programs.

Who should use this book

This book is for the following users:

• DB2 for Linux, UNIX, and Windows application developers who are familiar with Structured Query Language (SQL) and who know the Java programming language.
• DB2 for Linux, UNIX, and Windows system programmers who are installing JDBC and SQLJ support.
Chapter 1. Java application development for IBM data servers

The DB2 and IBM® Informix® (IDS) database systems provide driver support for client applications and applets that are written in Java.

You can access data in DB2 and IDS database systems using JDBC, SQL, or pureQuery.

**JDBC**

JDBC is an application programming interface (API) that Java applications use to access relational databases. IBM data server support for JDBC lets you write Java applications that access local DB2 or IDS data or remote relational data on a server that supports DRDA®.

**SQLJ**

SQLJ provides support for embedded static SQL in Java applications. SQLJ was initially developed by IBM, Oracle, and Tandem to complement the dynamic SQL JDBC model with a static SQL model.

For connections to DB2, in general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL.

For connections to IDS, SQL statements in JDBC or SQLJ applications run dynamically.

Because SQLJ can inter-operate with JDBC, an application program can use JDBC and SQLJ within the same unit of work.

**pureQuery**

pureQuery is a high-performance data access platform that makes it easier to develop, optimize, secure, and manage data access. It consists of:

- Application programming interfaces that are built for ease of use and for simplifying the use of best practices
- Development tools, which are delivered in IBM Optim Development Studio, for Java and SQL development
- A runtime, which is delivered in IBM Optim pureQuery Runtime, for optimizing and securing database access and simplifying management tasks

With pureQuery, you can write Java applications that treat relational data as objects, whether that data is in databases or JDBC DataSource objects. Your applications can also treat objects that are stored in in-memory Java collections as though those objects are relational data. To query or update your relational data or Java objects, you use SQL.

For more information on pureQuery, see the Integrated Data Management Information Center.
Supported drivers for JDBC and SQLJ

The DB2 product includes support for two types of JDBC driver architecture.

According to the JDBC specification, there are four types of JDBC driver architectures:

**Type 1**
Drivers that implement the JDBC API as a mapping to another data access API, such as Open Database Connectivity (ODBC). Drivers of this type are generally dependent on a native library, which limits their portability. The DB2 database system does not provide a type 1 driver.

**Type 2**
Drivers that are written partly in the Java programming language and partly in native code. The drivers use a native client library specific to the data source to which they connect. Because of the native code, their portability is limited.

**Type 3**
Drivers that use a pure Java client and communicate with a data server using a data-server-independent protocol. The data server then communicates the client's requests to the data source. The DB2 database system does not provide a type 3 driver.

**Type 4**
Drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

DB2 Database for Linux, UNIX, and Windows supports the following drivers:

<table>
<thead>
<tr>
<th>Driver name</th>
<th>Packaged as</th>
<th>Driver type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows</td>
<td>db2java.zip</td>
<td>Type 2</td>
</tr>
<tr>
<td>IBM Data Server Driver for JDBC and SQLJ</td>
<td>db2jcc.jar and sqlj.zip for JDBC 3.0 support, db2jcc4.jar and sqlj4.zip for support of some JDBC 4.0 functions</td>
<td>Type 2 and Type 4</td>
</tr>
</tbody>
</table>

**IBM Data Server Driver for JDBC and SQLJ (type 2 and type 4)**

The IBM Data Server Driver for JDBC and SQLJ is a single driver that includes JDBC type 2 and JDBC type 4 behavior. When an application loads the IBM Data Server Driver for JDBC and SQLJ, a single driver instance is loaded for type 2 and type 4 implementations. The application can make type 2 and type 4 connections using this single driver instance. The type 2 and type 4 connections can be made concurrently. IBM Data Server Driver for JDBC and SQLJ type 2 driver behavior is referred to as IBM Data Server Driver for JDBC and SQLJ type 2 connectivity. IBM Data Server Driver for JDBC and SQLJ type 4 driver behavior is referred to as IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Two versions of the IBM Data Server Driver for JDBC and SQLJ are available. IBM Data Server Driver for JDBC and SQLJ version 3.5.x is JDBC 3.0-compliant. IBM Data Server Driver for JDBC and SQLJ version 4.x is JDBC 4.0-compliant.

The IBM Data Server Driver for JDBC and SQLJ supports these JDBC and SQLJ functions:
• Version 3.5x supports all of the methods that are described in the JDBC 3.0 specifications.
• Version 4.x supports all of the methods that are described in the JDBC 4.0 specifications.
• SQLJ application programming interfaces, as defined by the SQLJ standards, for simplified data access from Java applications.
• Connections that are enabled for connection pooling. WebSphere® Application Server or another application server does the connection pooling.
• Connections to a data server from Java user-defined functions and stored procedures use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only. Applications that call user-defined functions or stored procedures can use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to connect to a data server. The IBM Data Server Driver for JDBC and SQLJ is the default driver for Java routines.
• Support for distributed transaction management. This support implements the Java 2 Platform, Enterprise Edition (J2EE) Java Transaction Service (JTS) and Java Transaction API (JTA) specifications, which conform to the X/Open standard for distributed transactions (Distributed Transaction Processing: The XA Specification, available from http://www.opengroup.org).

**DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC type 2 driver) (deprecated)**

The DB2 JDBC type 2 driver lets Java applications make calls to DB2 through JDBC. Calls to the DB2 JDBC type 2 driver are implemented with Java native methods. The DB2 JDBC Type 2 Driver uses the DB2 CLI interface to communicate with DB2 data servers. The Java applications that use this driver must run on a DB2 client, through which JDBC requests flow to the DB2 data server. DB2 Connect™ must be installed before the DB2 JDBC application driver can be used to access DB2 for i data sources or data sources in the DB2 for z/OS® environments.

The DB2 JDBC type 2 driver supports these JDBC and SQLJ functions:
• Most of the methods that are described in the JDBC 1.2 specification, and some of the methods that are described in the JDBC 2.0 specification.
• SQLJ statements that perform equivalent operations to all JDBC methods
• Connection pooling
• Distributed transactions
• Java user-defined functions and stored procedures

The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows will not be supported in future releases. You should therefore consider moving to the IBM Data Server Driver for JDBC and SQLJ.

**JDBC driver and database version compatibility**

The compatibility of a particular version of the IBM Data Server Driver for JDBC and SQLJ with a database version depends on the type of driver connectivity that you are using and the type of data source to which you are connecting.

**Compatibility for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity**

The IBM Data Server Driver for JDBC and SQLJ is always downward compatible with DB2 databases at the previous release level. For example, IBM Data Server Driver for JDBC and SQLJ type 4 connectivity from the IBM Data Server Driver for
JDBC and SQLJ version 3.61, which is shipped with DB2 Database for Linux, UNIX, and Windows Version 9.7 Fix Pack 3, to a DB2 Database for Linux, UNIX, and Windows Version 8 database is supported.

The IBM Data Server Driver for JDBC and SQLJ is upward compatible with the next version of a DB2 database if the applications under which the driver runs use no new features. For example, IBM Data Server Driver for JDBC and SQLJ type 4 connectivity from the IBM Data Server Driver for JDBC and SQLJ version 2.x, which is shipped with DB2 for z/OS Version 8, to a DB2 for z/OS Version 9.1 database is supported, if the applications under which the driver runs contain no DB2 for z/OS Version 9.1 features.

IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to IBM Informix is supported only for IDS Version 11 and later.

**Compatibility for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity**

In general, IBM Data Server Driver for JDBC and SQLJ type 2 connectivity is intended for connections to the local database system, using the driver version that is shipped with that database version. For example, version 3.6x of the IBM Data Server Driver for JDBC and SQLJ is shipped with DB2 Database for Linux, UNIX, and Windows Version 9.5 and Version 9.7, and DB2 for z/OS Version 8 and later.

However, for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a local DB2 Database for Linux, UNIX, and Windows database, the database version can be one version earlier or one version later than the DB2 Database for Linux, UNIX, and Windows version with which the driver was shipped. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a local DB2 for z/OS subsystem, the subsystem version can be one version later than the DB2 for z/OS version with which the driver was shipped.

If the database version to which your applications are connecting is later than the database version with which the driver was shipped, the applications cannot use features of the later database version.
Chapter 2. Installing the IBM Data Server Driver for JDBC and SQLJ

After you install the IBM Data Server Driver for JDBC and SQLJ, you can prepare and run JDBC or SQLJ applications.

Before you install the IBM Data Server Driver for JDBC and SQLJ, you need the following software.

- An SDK for Java, 1.4.2 or later.
  For all DB2 products except the IBM Data Server Runtime Client and the IBM Data Server Driver Package, the DB2 Database for Linux, UNIX, and Windows installation process automatically installs the SDK for Java, Version 5.
  If you want to use JDBC 4.0 functions, you need to install an SDK for Java, 6 or later.
  If you plan to run JDBC or SQLJ applications on your system, but not to prepare them, you need a Java run-time environment only.

**Important:** Support for the SDK for Java 1.4.2 is deprecated for Java routines, and might be discontinued in a future release.

- JVM native threads support
  Any JVMs that run Java applications that access DB2 databases must include native threads support. You can specify native threads as the default thread support for some JVMs by setting the THREADS_FLAG environment variable to "native". Refer to the documentation for your Java environment for instructions on making native threads the default on your system.

- Unicode support for System i® servers
  If any SQLJ or JDBC programs will use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to connect to a DB2 for i server, the System i operating system must support the Unicode UTF-8 encoding scheme. The following table lists the System i PTFs that you need for Unicode UTF-8 support:

<table>
<thead>
<tr>
<th>System i version</th>
<th>PTF numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSR3 or later</td>
<td>None (support is included)</td>
</tr>
</tbody>
</table>

- Java support for HP-UX clients and servers
  **HP-UX servers:** The IBM Data Server Driver for JDBC and SQLJ does not support databases that are in the HP-UX default character set, Roman8. Therefore, when you create a database on an HP-UX server that you plan to access with the IBM Data Server Driver for JDBC and SQLJ, you need to create the database with a different character set.
  **HP-UX clients and servers:** The Java environment on an HP-UX system requires special setup to run stored procedures under the IBM Data Server Driver for JDBC and SQLJ.

**Restriction:** If you install the IBM Data Server Driver for JDBC and SQLJ on a Windows 64-bit operating system, you cannot use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to connect to a DB2 Database for Linux, UNIX, and Windows instance from a 32-bit Java application.
Follow these steps to install the IBM Data Server Driver for JDBC and SQLJ.

1. During the DB2 Database for Linux, UNIX, and Windows installation process, select Java support on UNIX or Linux, or JDBC support on Windows. These selections are defaults. If you have already installed DB2 Database for Linux, UNIX, and Windows without JDBC support, you can run the installation process in Custom mode to add JDBC support.

Selection of Java support or JDBC support causes the installation process to perform the following actions:

- Installs the IBM Data Server Driver for JDBC and SQLJ class files.
  The files are placed in the sqllib\java directory for Windows systems, or the sqllib/java directory for UNIX or Linux systems.
  The files names are:

  **db2jcc.jar or db2jcc4.jar**
  Include db2jcc.jar in the CLASSPATH if you plan to use the version of the IBM Data Server Driver for JDBC and SQLJ that includes only **JDBC 3.0 and earlier functions**.
  Include db2jcc4.jar in the CLASSPATH if you plan to use the version of the IBM Data Server Driver for JDBC and SQLJ that includes **JDBC 4.0 and earlier functions**.

  **sqlj.zip or sqlj4.zip**
  Include sqlj.zip in the CLASSPATH if you plan to prepare SQLJ applications that include only **JDBC 3.0 and earlier functions**.
  Include sqlj4.zip in the CLASSPATH if you plan to prepare SQLJ applications that include **JDBC 4.0 and earlier functions**.

- Modifies the CLASSPATH to include the IBM Data Server Driver for JDBC and SQLJ class files.

**Important:** This step is performed automatically only for the db2jcc.jar and sqlj.zip file. If you are using the db2jcc4.jar file or the sqlj4.zip file, you must modify the CLASSPATH manually. Change db2jcc.jar to db2jcc4.jar or sqlj.zip to sqlj4.zip in the CLASSPATH.

You also need to make this change in every DB2 command line window that you open.

**Important:** Include db2jcc.jar or db2jcc4.jar in the CLASSPATH. Do not include both files.

**Important:** Include sqlj.zip or sqlj4.zip in the CLASSPATH. Do not include both files. Do not include db2jcc.jar with sqlj4.zip, or db2jcc4.jar with sqlj.zip.

- If IBM Data Server Driver for JDBC and SQLJ client license files exist, the installation process installs them and modifies the CLASSPATH to include them.
  The files are placed in the sqllib\java directory for Windows systems, or the sqllib/java directory for UNIX or Linux systems. The file names are:

<table>
<thead>
<tr>
<th>License file</th>
<th>Server to which license file permits a connection</th>
<th>Product that includes license file</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2jcc_license_cisuz.jar</td>
<td>DB2 for z/OS, DB2 for i</td>
<td>All DB2 Connect products</td>
</tr>
</tbody>
</table>

Table 2. IBM Data Server Driver for JDBC and SQLJ license files
Client license files are not required for connections to DB2 Database for Linux, UNIX, and Windows, Cloudscape, or IBM Informix (IDS) databases from the IBM Data Server Driver for JDBC and SQLJ version 3.50 or later.

- Installs IBM Data Server Driver for JDBC and SQLJ native libraries for support of IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

The files are placed in the sqllib\bin directory for Windows systems, or the sqllib/lib directory for UNIX or Linux systems.

The file names are:

- `libdb2jcc2.so`
  - For AIX®, HP-UX on IPF, Linux, and Solaris

- `db2jcc2.dll`
  - For Windows

2. Customize the driver-wide configuration properties, if any of the defaults are inappropriate.

3. Configure TCP/IP:

   Servers must be configured for TCP/IP communication in the following cases:
   - JDBC or SQLJ applications that use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
   - JDBC or SQLJ applications that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, and specify `server` and `port` in the connection URL.

   Ensure that the TCP/IP listener is running. To activate the TCP/IP listener:
   a. Set the environment variable `DB2COMM` to TCPIP:
      ```
      db2set DB2COMM=TCPIP
      ```
   b. Update the database manager configuration file with the TCP/IP service name as specified in the services file:
      ```
      db2 update dbm cfg using SVCENAME TCP/IP-service-name
      ```

      The port number used for applets and SQLJ programs needs to be the same as the TCP/IP SVCENAME number used in the database manager configuration file.
   c. Execute the `db2stop` and `db2start` commands for the service name setting to take effect.

4. On DB2 Database for Linux, UNIX, and Windows servers on which you plan to run Java stored procedures or user-defined functions, ensure that the `DB2_USE_DB2JCC2_JROUTINE` environment variable is not set, or is set to its default value of YES, yes, ON, on, TRUE, true, or 1 on those database servers. This setting indicates that Java stored procedures run under the IBM Data Server Driver for JDBC and SQLJ.

   If you need to run stored procedures under the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows, set the `DB2_USE_DB2JCC2_JROUTINE` environment variable to OFF.

5. On DB2 Database for Linux, UNIX, and Windows servers on which you plan to run Java stored procedures or user-defined functions, update the database manager configuration to include the path where the SDK for Java is located. You can do this by entering commands similar to these on the server command line:

   - For database systems on UNIX or Linux:
     ```
     db2 update dbm cfg using JDK_PATH /home/db2inst/jdk15
     ```

     `/home/db2inst/jdk15` is the path where the SDK for Java is installed.
• For database systems on Windows:
  
  `db2 update dbm cfg using JDK_PATH c:\Program Files\jdk15`

  `c:\Program Files\jdk15` is the path where the SDK for Java is installed.

To verify the correct value for the JDK_PATH field in the DB2 database manager configuration, enter the following command on the database server:

`db2 get dbm cfg`

You might want to redirect the output to a file for easier viewing. The JDK_PATH field appears near the beginning of the output.

6. If you plan to call SQL procedures that are on DB2 Database for Linux, UNIX, and Windows servers from Java programs, and the date and time format that is associated with the territory code of the database servers is **not** the USA format, take the following actions:
   a. Set the `DB2_SQLROUTINE_PREPOPTS` registry variable on the database servers to indicate that the default datetime format is ISO:
      ```
      db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
      ```
   b. Redefine any existing SQL procedures that you plan to call from Java programs.

   These steps are necessary to ensure that the calling application receives date and time values correctly.

7. If you plan to access DB2 for z/OS database servers with your Java applications, follow the instructions in “Special setup for accessing DB2 for z/OS servers from Java programs” in *Developing Java Applications*.

**DB2Binder utility**

The DB2Binder utility binds the DB2 packages that are used at the data server by the IBM Data Server Driver for JDBC and SQLJ, and grants EXECUTE authority on the packages to PUBLIC. Optionally, the DB2Binder utility can rebind DB2 packages that are not part of the IBM Data Server Driver for JDBC and SQLJ.

**DB2Binder syntax**

```
```

```
-action add | replace | drop | rebind
-reopt none | always | once | auto
```

```
-blocking all | unambig | no
-tracelevel 0 | 1 | 2 | 3 | 4 | 5 | 6
-trace-option trace-option
```

```
-collection collection-name
-size integer
```

8 Developing Java Applications
DB2Binder option descriptions

-url
Specifies the data source at which the IBM Data Server Driver for JDBC and SQLJ packages are to be bound. The variable parts of the -url value are:

  server
  The domain name or IP address of the operating system on which the data server resides.

  port
  The TCP/IP server port number that is assigned to the data server. The default is 446.

  database
  The location name for the data server, as defined in the SYSIBM.LOCATIONS catalog table.

-user
Specifies the user ID under which the packages are to be bound. This user must have BIND authority on the packages.

-action
Specifies the action to perform on the packages.

  add
  Indicates that a package can be created only if it does not already exist. Add is the default.

  replace
  Indicates that a package can be created even if a package with the same name already exists. The new package replaces the old package.

  rebind
  Indicates that the existing package should be rebound. This option does not apply to IBM Data Server Driver for JDBC and SQLJ packages. If -action rebind is specified, -generic must also be specified.

  drop
  Indicates that packages should be dropped:
  • For IBM Data Server Driver for JDBC and SQLJ packages, -action drop indicates that some or all IBM Data Server Driver for JDBC and SQLJ packages should be dropped. The number of packages depends on the -size parameter.
  • For user packages, -action drop indicates that the specified package should be dropped.

  -action drop applies only if the target data server is DB2 for z/OS.

-size
Controls the number of Statement, PreparedStatement, or CallableStatement
objects that can be open concurrently, or the number of IBM Data Server Driver for JDBC and SQLJ packages that are dropped.

The meaning of the -size parameter depends on the -action parameter:

- If the value of -action is add or replace, the value of -size is an integer that is used to calculate the number of DB2 packages that the IBM Data Server Driver for JDBC and SQLJ binds. If the value of -size is integer, the total number of packages is:

\[
\text{number-of-isolation-levels} \times \text{number-of-holdability-values} \times \text{integer} + \text{number-of-packages-for-static-SQL} = 4 \times 2 \times \text{integer} + 1
\]

The default -size value for -action add or -action replace is 3.

In most cases, the default of 3 is adequate. If your applications throw SQLExceptions with -805 SQLCODEs, check that the applications close all unused resources. If they do, increase the -size value.

- If the value of -action is replace, and the value of -size results in fewer packages than already exist, no packages are dropped.

- If the value of -action is drop, the value of -size is the number of packages that are dropped. If -size is not specified, all IBM Data Server Driver for JDBC and SQLJ packages are dropped.

- If the value of -action is rebind, -size is ignored.

- collection
  Specifies the collection ID for IBM Data Server Driver for JDBC and SQLJ or user packages. The default is NULLID. DB2Binder translates this value to uppercase.

  You can create multiple instances of the IBM Data Server Driver for JDBC and SQLJ packages on a single data server by running `com.ibm.db2.jcc.DB2Binder` multiple times, and specifying a different value for -collection each time. At run time, you select a copy of the IBM Data Server Driver for JDBC and SQLJ by setting the `currentPackageSet` property to a value that matches a -collection value.

- tracelevel
  Specifies what to trace while DB2Binder runs.

- reopt
  Specifies whether data servers determine access paths at run time. This option is not sent to the data server if it is not specified. In that case, the data server determines the reoptimization behavior.

  -reopt applies to connections to DB2 for z/OS Version 8 or later, or DB2 Database for Linux, UNIX, and Windows Version 9.1 or later.

  none Specifies that access paths are not determined at run time.

  always
    Specifies that access paths are determined each time a statement is run.

  once
    Specifies that DB2 determines and caches the access path for a dynamic statement only once at run time. DB2 uses this access path until the prepared statement is invalidated, or until the statement is removed from the dynamic statement cache and needs to be prepared again.
**auto** Specifies that access paths are automatically determined by the data server. auto is valid only for connections to DB2 for z/OS data servers.

**-blocking**
Specifies the type of row blocking for cursors.
- **ALL** For cursors that are specified with the FOR READ ONLY clause or are not specified as FOR UPDATE, blocking occurs.
- **UNAMBIG** For cursors that are specified with the FOR READ ONLY clause, blocking occurs.
  
  Cursors that are not declared with the FOR READ ONLY or FOR UPDATE clause which are not ambiguous and are read-only will be blocked. Ambiguous cursors will not be blocked.
- **NO** Blocking does not occur for any cursor.
  
  For the definition of a read-only cursor and an ambiguous cursor, refer to "DECLARE CURSOR".

**-optprofile**
Specifies an optimization profile that is used for optimization of data change statements in the packages. This profile is an XML file that must exist on the target server. If -optprofile is not specified, and the CURRENT OPTIMIZATION PROFILE special register is set, the value of CURRENT OPTIMIZATION PROFILE is used. If -optprofile is not specified, and CURRENT OPTIMIZATION PROFILE is not set, no optimization profile is used.

-optprofile is valid only for connections to DB2 Database for Linux, UNIX, and Windows data servers.

**-owner**
Specifies the authorization ID of the owner of the packages. The default value is set by the data server.

-owner applies only to IBM Data Server Driver for JDBC and SQLJ packages.

**-sqlid**
Specifies a value to which the CURRENT SQUID special register is set before DB2Binder executes GRANT operations on the IBM Data Server Driver for JDBC and SQLJ packages. If the primary authorization ID does not have a sufficient level of authority to grant privileges on the packages, and the primary authorization ID has an associated secondary authorization ID that has those privileges, set -sqlid to the secondary authorization ID.

-sqlid is valid only for connections to DB2 for z/OS data servers.

**-generic**
Specifies that DB2Binder rebinds a user package instead of the IBM Data Server Driver for JDBC and SQLJ packages. If -generic is specified, -action rebind and -package must also be specified.

**-package**
Specifies the name of the package that is to be rebound. This option applies only to user packages. If -package is specified, -action rebind and -generic must also be specified.

**-version**
Specifies the version ID of the package that is to be rebound. If -version is specified, -action rebind, -package, and -generic must also be specified.
-**bindoptions**

Specifies a string that is enclosed in quotation marks. The contents of that string are one or more parameter and value pairs that represent options for rebinding a user package. All items in the string are delimited with spaces:

"parm1 value1 parm2 value2 ... parmn valuen"

**bindoptions** does not apply to IBM Data Server Driver for JDBC and SQLJ packages.

Possible parameters and values are:

- **bindObjectExistenceRequired**
  Specifies whether the data server issues an error and does not rebind the package, if all objects or needed privileges do not exist at rebind time. Possible values are:
  
  **true**  This option corresponds to the SQLERROR(NOPACKAGE) bind option.
  
  **false**  This option corresponds to the SQLERROR(CONTINUE) bind option.

- **degreeIOParallelism**
  Specifies whether to attempt to run static queries using parallel processing to maximize performance. Possible values are:
  
  **1**  No parallel processing.

  This option corresponds to the DEGREE(1) bind option.

  **-1**  Allow parallel processing.

  This option corresponds to the DEGREE(ANY) bind option.

- **packageAuthorizationRules**
  Determines the values that apply at runtime for the following dynamic SQL attributes:
  
  - The authorization ID that is used to check authorization
  - The qualifier that is used for unqualified objects
  - The source for application programming options that the data server uses to parse and semantically verify dynamic SQL statements
  - Whether dynamic SQL statements can include GRANT, REVOKE, ALTER, CREATE, DROP, and RENAME statements

  Possible values are:
  
  **0**  Use run behavior. This is the default.

  This option corresponds to the DYNAMICRULES(RUN) bind option.

  **1**  Use bind behavior.

  This option corresponds to the DYNAMICRULES(BIND) bind option.

  **2**  When the package is run as or runs under a stored procedure or user-defined function package, the data server processes dynamic SQL statements using invoke behavior. Otherwise, the data server processes dynamic SQL statements using run behavior.
This option corresponds to the DYNAMICRULES(INVOKEBIND) bind option.

3 When the package is run as or runs under a stored procedure or user-defined function package, the data server processes dynamic SQL statements using invoke behavior. Otherwise, the data server processes dynamic SQL statements using bind behavior.

This option corresponds to the DYNAMICRULES(DEFINERUN) bind option.

4 When the package is run as or runs under a stored procedure or user-defined function package, the data server processes dynamic SQL statements using define behavior. Otherwise, the data server processes dynamic SQL statements using run behavior.

This option corresponds to the DYNAMICRULES(DEFINEBIND) bind option.

packageOwnerIdentifier
Specifies the authorization ID of the owner of the packages.

isolationLevel
Specifies how far to isolate an application from the effects of other running applications. Possible values are:

1 Uncommitted read
This option corresponds to the ISOLATION(UR) bind option.

2 Cursor stability
This option corresponds to the ISOLATION(CS) bind option.

3 Read stability
This option corresponds to the ISOLATION(RS) bind option.

4 Repeatable read
This option corresponds to the ISOLATION(RR) bind option.

releasePackageResourcesAtCommit
Specifies when to release resources that a program uses at each commit point. Possible values are:

true This option corresponds to the RELEASE(COMMIT) bind option.

false This option corresponds to the RELEASE(DEALLOCATE) bind option.

If -bindoptions is specified, -generic must also be specified.
-verbose
Specifies that the DB2Binder utility displays detailed information about the
bind process.

-help
Specifies that the DB2Binder utility describes each of the options that it
supports. If any other options are specified with -help, they are ignored.

**DB2Binder return codes when the target operating system is not Windows**

If the target data source for DB2Binder is not on the Windows operating system,
DB2Binder returns one of the following return codes.

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>1</td>
<td>An error occurred during DB2Binder execution.</td>
</tr>
</tbody>
</table>

**DB2Binder return codes when the target operating system is Windows**

If the target data source for DB2Binder is on the Windows operating system,
DB2Binder returns one of the following return codes.

<table>
<thead>
<tr>
<th>Return code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful execution.</td>
</tr>
<tr>
<td>-100</td>
<td>No bind options were specified.</td>
</tr>
<tr>
<td>-101</td>
<td>-url value was not specified.</td>
</tr>
<tr>
<td>-102</td>
<td>-user value was not specified.</td>
</tr>
<tr>
<td>-103</td>
<td>-password value was not specified.</td>
</tr>
<tr>
<td>-200</td>
<td>No valid bind options were specified.</td>
</tr>
<tr>
<td>-114</td>
<td>The -package option was not specified, but the -generic option was specified.</td>
</tr>
<tr>
<td>-201</td>
<td>-url value is invalid.</td>
</tr>
<tr>
<td>-204</td>
<td>-action value is invalid.</td>
</tr>
<tr>
<td>-205</td>
<td>-blocking value is invalid.</td>
</tr>
<tr>
<td>-206</td>
<td>-collection value is invalid.</td>
</tr>
<tr>
<td>-207</td>
<td>-dbprotocol value is invalid.</td>
</tr>
<tr>
<td>-208</td>
<td>-keepdynamic value is invalid.</td>
</tr>
<tr>
<td>-210</td>
<td>-reopt value is invalid.</td>
</tr>
<tr>
<td>-211</td>
<td>-size value is invalid.</td>
</tr>
<tr>
<td>-212</td>
<td>-tracelevel value is invalid.</td>
</tr>
<tr>
<td>-307</td>
<td>-dbprotocol value is not supported by the target data server.</td>
</tr>
<tr>
<td>-308</td>
<td>-keepdynamic value is not supported by the target data server.</td>
</tr>
<tr>
<td>-310</td>
<td>-reopt value is not supported by the target data server.</td>
</tr>
<tr>
<td>Return Code</td>
<td>Meaning</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-313</td>
<td>-optprofile value is not supported by the target data server.</td>
</tr>
<tr>
<td>-401</td>
<td>The Binder class was not found.</td>
</tr>
<tr>
<td>-402</td>
<td>Connection to the data server failed.</td>
</tr>
<tr>
<td>-403</td>
<td>DatabaseMetaData retrieval for the data server failed.</td>
</tr>
<tr>
<td>-501</td>
<td>No more packages are available in the cluster.</td>
</tr>
<tr>
<td>-502</td>
<td>An existing package is not valid.</td>
</tr>
<tr>
<td>-503</td>
<td>The bind process returned an error.</td>
</tr>
<tr>
<td>-999</td>
<td>An error occurred during processing of an undocumented bind option.</td>
</tr>
</tbody>
</table>

**DB2LobTableCreator utility**

The DB2LobTableCreator utility creates tables on a DB2 for z/OS database server. Those tables are required by JDBC or SQLJ applications that use LOB locators to access data in DBCLOB or CLOB columns.

**DB2LobTableCreator syntax**

```bash
java com.ibm.db2.jcc.DB2LobTableCreator -url jdbc:db2://server:port/database
```

**DB2LobTableCreator option descriptions**

- `-url`
  Specifies the data source at which DB2LobTableCreator is to run. The variable parts of the `-url` value are:

  `-jdbc:db2:`
  Indicates that the connection is to a server in the DB2 family.

  `-server`
  The domain name or IP address of the database server.

  `-port`
  The TCP/IP server port number that is assigned to the database server. This is an integer between 0 and 65535. The default is 446.

  `-database`
  A name for the database server.

    `database` is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

    ```sql
    SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
    ```

- `-user`
  Specifies the user ID under which DB2LobTableCreator is to run. This user must have authority to create tables in the DSNATPDB database.
-password
   Specifies the password for the user ID.

-help
   Specifies that the DB2LobTableCreator utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

---

**Customization of IBM Data Server Driver for JDBC and SQLJ configuration properties**

The IBM Data Server Driver for JDBC and SQLJ configuration properties let you set property values that have driver-wide scope. Those settings apply across applications and DataSource instances. You can change the settings without having to change application source code or DataSource characteristics.

Each IBM Data Server Driver for JDBC and SQLJ configuration property setting is of this form:

```
property=value
```

You can set configuration properties in the following ways:

- Set the configuration properties as Java system properties. Configuration property values that are set as Java system properties override configuration property values that are set in any other ways.
  
  For stand-alone Java applications, you can set the configuration properties as Java system properties by specifying `-Dproperty=value` for each configuration property when you execute the java command.

- Set the configuration properties in a resource whose name you specify in the `db2.jcc.propertiesFile` Java system property. For example, you can specify an absolute path name for the `db2.jcc.propertiesFile` value.
  
  For stand-alone Java applications, you can set the configuration properties by specifying the `-Ddb2.jcc.propertiesFile=path` option when you execute the java command.

- Set the configuration properties in a resource named `DB2JccConfiguration.properties`. A standard Java resource search is used to find `DB2JccConfiguration.properties`. The IBM Data Server Driver for JDBC and SQLJ searches for this resource only if you have not set the `db2.jcc.propertiesFile` Java system property.
  
  `DB2JccConfiguration.properties` can be a stand-alone file, or it can be included in a JAR file.
  
  If the `DB2JccConfiguration.properties` file is in the ISO 8859-1 (Latin-1) encoding scheme, or is in the Latin-1 encoding scheme with some Unicode-encoded (`\udxxxx`) characters, you do not need to do character conversion before the IBM Data Server Driver for JDBC and SQLJ can use the file. If the `DB2JccConfiguration.properties` file is in some other encoding scheme, you need to use the Java native2ascii converter to convert the contents to Latin-1 or Unicode-encoded characters.

  If `DB2JccConfiguration.properties` is a stand-alone file, the path for `DB2JccConfiguration.properties` must be in the CLASSPATH concatenation.

  If `DB2JccConfiguration.properties` is in a JAR file, the JAR file must be in the CLASSPATH concatenation.
Special setup for accessing DB2 for z/OS servers from Java programs

If you plan to write JDBC or SQLJ applications that access DB2 for z/OS database servers, your IBM Data Server Driver for JDBC and SQLJ installation process requires additional steps.

Follow these steps to allow connectivity to DB2 for z/OS servers:

1. If you plan to connect to any DB2 for z/OS Version 7 or Version 8 database servers, install these PTFs on those database servers.

   Table 5. PTFs for DB2 for z/OS stored procedures
   
<table>
<thead>
<tr>
<th>DB2 for z/OS</th>
<th>PTF or APAR numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 7</td>
<td>UQ72083, UQ93889, UK21848</td>
</tr>
<tr>
<td>Version 8</td>
<td>UQ93890, UK21849</td>
</tr>
<tr>
<td>Version 9</td>
<td>PK44166</td>
</tr>
</tbody>
</table>

2. Run the com.ibm.db2.jcc.DB2Binder utility to bind the DB2 packages that are used at the server by the IBM Data Server Driver for JDBC and SQLJ.

3. On DB2 for z/OS database servers, customize and run job DSNTIJMS.
   
   DSNTIJMS is located in data set prefix.SDSNSAMP. It performs the following functions:
   
   - Creates the following stored procedures to support DatabaseMetaData methods, tracing, and error message formatting.
     - SQLCOLPRIVILEGES
     - SQLCOLUMNS
     - SQLFOREIGNKEYS
     - SQLFUNCTIONS
     - SQLFUNCTIONCOLUMNS
     - SQLGETTYPEINFO
     - SQLPRIMARKEYS
     - SQLPROCEDURECOLS
     - SQLPROCEDURES
     - SQLSPECIALCOLUMNS
     - SQLSTATISTICS
     - SQLTABLEPRIVILEGES
     - SQLTABLES
     - SQLUDTS
     - SQLCAMEMESSAGE
   
   - Creates the following tables to support efficient storing of data in CLOB or DBCLOB columns and the use of LOB locators for CLOB or DBCLOB retrieval.
     - SYSIBM.SYSDUMMYU
     - SYSIBM.SYSDUMMYA
     - SYSIBM.SYSDUMMYE
   
   An alternative way to create those tables is to run the com.ibm.db2.jcc.DB2LobTableCreator utility on the client, against each of the DB2 for z/OS servers.

4. Enable Unicode support for OS/390® and z/OS servers.
   
   If any SQLJ or JDBC programs will use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to connect to a DB2 for z/OS Version 7 server, the OS/390 or z/OS operating system must support the Unicode UTF-8 encoding scheme. This support requires OS/390 Version 2 Release 9 with APAR.
OW44581, or a later release of OS/390 or z/OS, plus the OS/390 R8/R9/R10 Support for Unicode. Information APARs II13048 and II13049 contain additional information.

5. If you plan to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to implement distributed transactions against DB2 for z/OS Version 7 servers, run the DB2T4XAIndoubtUtil utility once for each of those DB2 for z/OS Version 7 servers.

DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers

If you plan to implement distributed transactions using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity that include DB2 UDB for OS/390 and z/OS Version 7 servers, you need to run the DB2T4XAIndoubtUtil utility against those servers.

DB2T4XAIndoubtUtil allows Version 7 servers, which do not have built-in support for distributed transactions that implement the XA specification, to emulate that support.

DB2T4XAIndoubtUtil performs one or both of the following tasks:
- Creates a table named SYSIBM.INDOUBT and an associated index
- Binds DB2 packages named T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04

You should create and drop packages T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04 only by running DB2T4XAIndoubtUtil. You can create and drop SYSTEM.INDOUBT and its index manually, but it is recommended that you use the utility. See [DB2T4XAIndoubtUtil usage notes] for instructions on how to create those objects manually.

DB2T4XAIndoubtUtil authorization

To run the DB2T4XAIndoubtUtil utility to create SYSTEM.INDOUBT and bind packages T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04, you need SYSADM authority.

To run the DB2T4XAIndoubtUtil only to bind packages T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04, you need BIND authority on the packages.

DB2T4XAIndoubtUtil syntax

```
java -cp com.ibm.db2.jcc.DBT4XAIndoubtUtil -url jdbc:db2://server:port/database
-
-user user-ID -password password -owner owner-ID -help -delete
-
-priqty integer -secqty integer -bindonly -showSQL
```
DB2T4XAIndoubtUtil parameter descriptions

-url
Specifies the data source at which DB2T4XAIndoubtUtil is to run. The variable parts of the -url value are:

jdbc:db2:
Indicates that the connection is to a server in the DB2 family.

server
The domain name or IP address of the database server.

port
The TCP/IP server port number that is assigned to the database server. This is an integer between 0 and 65535. The default is 446.

database
A name for the database server.

database is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;

-user
Specifies the user ID under which DB2T4XAIndoubtUtil is to run. This user must have SYSADM authority or must be a member of a RACF® group that corresponds to a secondary authorization ID with SYSADM authority.

-password
Specifies the password for the user ID.

-owner
Specifies a secondary authorization ID that has SYSADM authority. Use the -owner parameter if the -user parameter value does not have SYSADM authority. The -user parameter value must be a member of a RACF group whose name is owner-ID.

When the -owner parameter is specified, DB2T4XAIndoubtUtil uses owner-ID as:

• The authorization ID for creating the SYSIBM.INDOUBT table.
• The authorization ID of the owner of the T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04 packages. SQL statements in those packages are executed using the authority of owner-ID.

-help
Specifies that the DB2T4XAIndoubtUtil utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

-delete
Specifies that the DB2T4XAIndoubtUtil utility deletes the objects that were created when DB2T4XAIndoubtUtil was run previously.

-priqty
Specifies the primary space allocation, in kilobytes, for the table space that contains the SYSIBM.INDOUBT table. The default value for -priqty is 1000.
**Important:** The -priqty value divided by the page size for the table space in which SYSIBM.INDOUBT resides must be greater than the maximum number of indoubt transactions that are allowed at a given time. For example, for a 4 KB page size, the default -priqty value of 1000 allows about 250 concurrent indoubt transactions.

**-secqty**

Specifies the secondary space allocation, in kilobytes, for the table space that contains the SYSIBM.INDOUBT table. The default value for -secqty is 0.

**Recommendation:** Always use the default value of 0 for the -secqty value, and specify a -priqty value that is large enough to accommodate the maximum number of concurrent indoubt transactions.

**-bindonly**

Specifies that the DB2T4XAIndoubtUtil utility binds the T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04 packages and grants permission to PUBLIC to execute the packages, but does not create the SYSIBM.INDOUBT table.

**-showSQL**

Specifies that the DB2T4XAIndoubtUtil utility displays the SQL statements that it executes.

**-jdbcCollection**

Specifies the value of the -collection parameter that was used when the IBM Data Server Driver for JDBC and SQLJ packages were bound with the DB2Binder utility. The -jdbcCollection parameter must be specified if the explicitly or implicitly specified value of the -collection parameter was not NULLID.

The default is -jdbcCollection NULLID.

**DB2T4XAIndoubtUtil usage notes**

To create the SYSTEM.INDOUBT table and its index manually, use these SQL statements:

```sql
CREATE TABLESPACE INDBTTS
USING STOGROUP
LOCKSIZE ROW
BUFFERPOOL BP0
SEGSize 32
CCSID EBCDIC;

CREATE TABLE SYSIBM.INDOUBT(indbtXid VARCHAR(140) FOR BIT DATA NOT NULL,
                            uowId VARCHAR(25) FOR BIT DATA NOT NULL,
                            pSyncLog VARCHAR(150) FOR BIT DATA,
                            cSyncLog VARCHAR(150) FOR BIT DATA)
IN INDBTTS;

CREATE UNIQUE INDEX INDBTIDX ON SYSIBM.INDOUBT(indbtXid, uowId);
```

**DB2T4XAIndoubtUtil example**

Run the DB2T4XAIndoubtUtil to allow a DB2 for OS/390 and z/OS Version 7 subsystem that has IP address mvs1, port number 446, and DB2 location name SJCEC1 to participate in XA distributed transactions.

```
java com.ibm.db2.jcc.DB2T4XAIndoubtUtil -url jdbc:db2://mvs1:446/SJCEC1 -user SYSADM -password mypass
```
Chapter 3. JDBC application programming

Writing a JDBC application has much in common with writing an SQL application in any other language.

In general, you need to do the following things:
- Access the Java packages that contain JDBC methods.
- Declare variables for sending data to or retrieving data from DB2 tables.
- Connect to a data source.
- Execute SQL statements.
- Handle SQL errors and warnings.
- Disconnect from the data source.

Although the tasks that you need to perform are similar to those in other languages, the way that you execute those tasks is somewhat different.

Example of a simple JDBC application

A simple JDBC application demonstrates the basic elements that JDBC applications need to include.

```
import java.sql.*;

public class EzJava
{
    public static void main(String[] args)
    {
        String urlPrefix = "jdbc:db2:";
        String url;
        String user;
        String password;
        String empNo;
        Connection con;
        Statement stmt;
        ResultSet rs;

        System.out.println("**** Enter class EzJava");

        // Check the that first argument has the correct form for the portion
        // of the URL that follows jdbc:db2:,
        // as described
        // in the Connecting to a data source using the DriverManager
        // interface with the IBM Data Server Driver for JDBC and SQLJ topic.
        // For example, for IBM Data Server Driver for
        // JDBC and SQLJ type 2 connectivity,
        // args[0] might be MVS10B2M. For
        // type 4 connectivity, args[0] might
        // be //stimvs1:10110/MVS1DB2M.
        if (args.length!=3)
        {
            System.err.println("Invalid value. First argument appended to "+
            "jdbc:db2: must specify a valid URL.");
            System.err.println("Second argument must be a valid user ID.");
            System.err.println("Third argument must be the password for the user ID.");
```
System.exit(1);
}
url = urlPrefix + args[0];
user = args[1];
password = args[2];
try
{
    // Load the driver
    Class.forName("com.ibm.db2.jcc.DB2Driver");
    System.out.println("**** Loaded the JDBC driver");

    // Create the connection using the IBM Data Server Driver for JDBC and SQLJ
    con = DriverManager.getConnection (url, user, password);
    // Commit changes manually
    con.setAutoCommit(false);
    System.out.println("**** Created a JDBC connection to the data source");

    // Create the Statement
    stmt = con.createStatement();
    System.out.println("**** Created JDBC Statement object");

    // Execute a query and generate a ResultSet instance
    rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
    System.out.println("**** Created JDBC ResultSet object");

    // Print all of the employee numbers to standard output device
    while (rs.next()) {
        empNo = rs.getString(1);
        System.out.println("Employee number = " + empNo);
    }
    System.out.println("**** Fetched all rows from JDBC ResultSet");
    // Close the ResultSet
    rs.close();
    System.out.println("**** Closed JDBC ResultSet");

    // Close the Statement
    stmt.close();
    System.out.println("**** Closed JDBC Statement");

    // Connection must be on a unit-of-work boundary to allow close
    con.commit();
    System.out.println ("**** Transaction committed");

    // Close the connection
    con.close();
    System.out.println("**** Disconnected from data source");
    System.out.println("**** JDBC Exit from class EzJava - no errors");
}
catch (ClassNotFoundException e)
{
    System.err.println("Could not load JDBC driver");
    System.out.println("Exception: " + e);
    e.printStackTrace();
}
catch(SQLException ex)
{
    System.err.println("SQLException information");
    while(ex!=null) {
        System.err.println ("Error msg: " + ex.getMessage());
        System.err.println ("SQLSTATE: " + ex.getSQLState());
        System.err.println ("Error code: " + ex.getErrorCode());
        ex.printStackTrace();
        ex = ex.getNextException(); // For drivers that support chained exceptions
}
Notes to Figure 1 on page 21:

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This statement imports the java.sql package, which contains the JDBC core API. For information on other Java packages that you might need to access, see &quot;Java packages for JDBC support&quot;.</td>
</tr>
<tr>
<td>2</td>
<td>String variable empNo performs the function of a host variable. That is, it is used to hold data retrieved from an SQL query. See &quot;Variables in JDBC applications&quot; for more information.</td>
</tr>
<tr>
<td>3a</td>
<td>These two sets of statements demonstrate how to connect to a data source using one of two available interfaces. See &quot;How JDBC applications connect to a data source&quot; for more details.</td>
</tr>
<tr>
<td>3b</td>
<td>Step 3a (loading the JDBC driver) is not necessary if you use JDBC 4.0.</td>
</tr>
<tr>
<td>4a</td>
<td>These two sets of statements demonstrate how to perform a SELECT in JDBC. For information on how to perform other SQL operations, see &quot;JDBC interfaces for executing SQL&quot;.</td>
</tr>
<tr>
<td>4b</td>
<td>For more information on handling SQL errors, see &quot;Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ&quot;. For information on handling SQL warnings, see &quot;Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ&quot;.</td>
</tr>
<tr>
<td>5</td>
<td>This try/catch block demonstrates the use of the SQLException class for SQL error handling. For more information on handling SQL errors, see &quot;Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ&quot;. For information on handling SQL warnings, see &quot;Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ&quot;.</td>
</tr>
<tr>
<td>6</td>
<td>This statement disconnects the application from the data source. See &quot;Disconnecting from data sources in JDBC applications&quot;.</td>
</tr>
</tbody>
</table>

How JDBC applications connect to a data source

Before you can execute SQL statements in any SQL program, you must be connected to a data source.

The IBM Data Server Driver for JDBC and SQLJ supports type 2 and type 4 connectivity. Connections to DB2 databases can use type 2 or type 4 connectivity. Connections to IBM Informix (IDS) databases can use type 4 connectivity.

The following figure shows how a Java application connects to a data source using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.
The following figure shows how a Java application connects to a data source using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

*Java byte code executed under JVM, and native code

**Figure 2. Java application flow for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity**

The following figure shows how a Java application connects to a data source using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

*Java byte code executed under JVM

**Figure 3. Java application flow for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity**
How DB2 applications connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver

A JDBC application can establish a connection to a data source using the JDBC DriverManager interface, which is part of the java.sql package.

The Java application first loads the JDBC driver by invoking the Class.forName method. After the application loads the driver, it connects to a data source by invoking the DriverManager.getConnection method.

For the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver), you load the driver by invoking the Class.forName method with the following argument:

```java
COM.ibm.db2.jdbc.app.DB2Driver
```

The following code demonstrates loading the DB2 JDBC Type 2 Driver:

```java
try {
    // Load the DB2 JDBC Type 2 Driver with DriverManager
    Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

The catch block is used to print an error if the driver is not found.

After you load the driver, you connect to the data source by invoking the DriverManager.getConnection method. You can use one of the following forms of getConnection:

```java
getConnection(String url);
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

The `url` argument represents a data source.

For the DB2 JDBC Type 2 Driver, specify a URL of the following form:

**Syntax for a URL for the DB2 JDBC Type 2 Driver:**

```sql
jdbc:db2:database
```

The parts of the URL have the following meanings:

- **jdbcdb2:**
  - `jdbcdb2`: indicates that the connection is to a DB2 data source.
  
- **database**
  - A database alias. The alias refers to the DB2 database catalog entry on the DB2 client.

The `info` argument is an object of type java.util.Properties that contains a set of driver properties for the connection. Specifying the `info` argument is an alternative to specifying `property=value` strings in the URL.

**Specifying a user ID and password for a connection:** There are several ways to specify a user ID and password for a connection:

- Use the form of the `getConnection` method that specifies `user` and `password`. 

Chapter 3. JDBC application programming 25
Use the form of the getConnection method that specifies info, after setting the user and password properties in a java.util.Properties object.

Example: Setting the user ID and password in user and password parameters:
```java
String url = "jdbc:db2:toronto"; // Set URL for data source
String user = "db2adm";
String password = "db2adm";
Connection con = DriverManager.getConnection(url, user, password); // Create connection
```

Example: Setting the user ID and password in a java.util.Properties object:
```java
Properties properties = new Properties(); // Create Properties object
properties.put("user", "db2adm"); // Set user ID for connection
properties.put("password", "db2adm"); // Set password for connection
String url = "jdbc:db2:toronto"; // Set URL for data source
Connection con = DriverManager.getConnection(url, properties); // Create connection
```

Connecting to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ

A JDBC application can establish a connection to a data source using the JDBC DriverManager interface, which is part of the java.sql package.

The steps for establishing a connection are:
1. Load the JDBC driver by invoking the Class.forName method.
   - If you are using JDBC 4.0, you do not need to explicitly load the JDBC driver.
   - For the IBM Data Server Driver for JDBC and SQLJ, you load the driver by invoking the Class.forName method with the following argument:
     ```java
     com.ibm.db2.jcc.DB2Driver
     ```
   - For compatibility with previous JDBC drivers, you can use the following argument instead:
     ```java
     COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
     ```
   - The following code demonstrates loading the IBM Data Server Driver for JDBC and SQLJ:
     ```java
     try {
         // Load the IBM Data Server Driver for JDBC and SQLJ with DriverManager
         Class.forName("com.ibm.db2.jcc.DB2Driver");
     } catch (ClassNotFoundException e) {
         e.printStackTrace();
     }
     ```
   - The catch block is used to print an error if the driver is not found.

2. Connect to a data source by invoking the DriverManager.getConnection method.
   - You can use one of the following forms of getConnection:
     ```java
     getConnection(String url);
     getConnection(String url, user, password);
     getConnection(String url, java.util.Properties info);
     ```
   - For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, the getConnection method must specify a user ID and password, through parameters or through property values.
The *url* argument represents a data source, and indicates what type of JDBC connectivity you are using.

The *info* argument is an object of type java.util.Properties that contains a set of driver properties for the connection. Specifying the *info* argument is an alternative to specifying *property=value*; strings in the URL. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for the properties that you can specify.

There are several ways to specify a user ID and password for a connection:

- Use the form of the getConnection method that specifies *url* with
  
  *property=value*; clauses, and include the user and password properties in the URL.

- Use the form of the getConnection method that specifies *user* and *password*.

- Use the form of the getConnection method that specifies *info*, after setting the user and password properties in a java.util.Properties object.

*Example: Establishing a connection and setting the user ID and password in a URL:*

```java
String url = "jdbc:db2://myhost:5021/mydb:" + 
  "user=dbadm;password=dbadm;";

// Set URL for data source
Connection con = DriverManager.getConnection(url);
// Create connection
```

*Example: Establishing a connection and setting the user ID and password in user and password parameters:*

```java
String url = "jdbc:db2://myhost:5021/mydb";
String user = "dbadm";
String password = "dbadm";
Connection con = DriverManager.getConnection(url, user, password);
// Create connection
```

*Example: Establishing a connection and setting the user ID and password in a java.util.Properties object:*

```java
Properties properties = new Properties(); // Create Properties object
properties.put("user", "dbadm"); // Set user ID for connection
properties.put("password", "dbadm"); // Set password for connection
String url = "jdbc:db2://myhost:5021/mydb";
// Set URL for data source
Connection con = DriverManager.getConnection(url, properties);
// Create connection
```

**URL format for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity**

If you are using type 4 connectivity in your JDBC application, and you are making a connection using the DriverManager interface, you need to specify a URL in the DriverManager.getConnection call that indicates type 4 connectivity.

**IBM Data Server Driver for JDBC and SQLJ type 4 connectivity URL syntax**

```
jdbc:db2: //server /database
jdbc:db2j:net://server /database
jdbc:ids://server /database
```

```bash
:port
://property=value
```
IBM Data Server Driver for JDBC and SQLJ type 4 connectivity URL option descriptions

The parts of the URL have the following meanings:

jdbc:db2: or jdbc:db2j:net:
   The meanings of the initial portion of the URL are:
   jdbc:db2:
      Indicates that the connection is to a DB2 for z/OS, DB2 Database for Linux, UNIX, and Windows.
      jdbc:db2: can also be used for a connection to an IBM Informix (IDS) database, for application portability.
   jdbc:db2j:net:
      Indicates that the connection is to a remote IBM Cloudscape server.
   jdbc:ids:
      Indicates that the connection is to an IDS data source.
   jdbc:informix-sqli: also indicates that the connection is to an IDS data source, but jdbc:ids: should be used.

server
   The domain name or IP address of the data source.

port
   The TCP/IP server port number that is assigned to the data source. This is an integer between 0 and 65535. The default is 446.

database
   A name for the data source.
   - If the connection is to a DB2 for z/OS server, database is the DB2 location name that is defined during installation. All characters in the DB2 location name must be uppercase characters. The IBM Data Server Driver for JDBC and SQLJ does not convert lowercase characters in the database value to uppercase for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
   You can determine the location name by executing the following SQL statement on the server:
   SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
   - If the connection is to a DB2 for z/OS server or a DB2 for i server, all characters in database must be uppercase characters.
   - If the connection is to a DB2 Database for Linux, UNIX, and Windows server, database is the database name that is defined during installation.
   - If the connection is to an IDS server, database is the database name. The name is case-insensitive. The server converts the name to lowercase.
   - If the connection is to an IBM Cloudscape server, the database is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:
   "c:/databases/testdb"

property=value;
   A property and its value for the JDBC connection. You can specify one or more property and value pairs. Each property and value pair, including the last one, must end with a semicolon (;). Do not include spaces or other white space characters anywhere within the list of property and value strings.
Some properties with an int data type have predefined constant field values. You must resolve constant field values to their integer values before you can use those values in the \texttt{url} parameter. For example, you cannot use \texttt{com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL} in a \texttt{url} parameter. However, you can build a URL string that includes \texttt{com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL}, and assign the URL string to a \texttt{String} variable. Then you can use the \texttt{String} variable in the \texttt{url} parameter:

```java
String url =
"jdbc:db2://sysmvs1.stl.ibm.com:5021/STLEC1" +
*:user=dbadm;password=dbadm;" +
"traceLevel=" +
(com.ibm.db2.jcc.DB2BaseDataSource.TRACE\_ALL) + "*;";
Connection con =
java.sql.DriverManager.getConnection(url);
```

**URL format for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity**

If you are using type 2 connectivity in your JDBC application, and you are making a connection using the \texttt{DriverManager} interface, you need to specify a URL in the \texttt{DriverManager.getConnection} call that indicates type 2 connectivity.

**IBM Data Server Driver for JDBC and SQLJ type 2 connectivity URL syntax**

The parts of the URL have the following meanings:

- \texttt{jdbc\_db2: or jdbc\_db2os390: or jdbc\_db2os390sqlj: or jdbc\_default:connection}
- \texttt{database}
- \texttt{property=value}

**IBM Data Server Driver for JDBC and SQLJ type 2 connectivity URL options descriptions**

The meanings of the initial portion of the URL are:

- \texttt{jdbc\_db2: or jdbc\_db2os390: or jdbc\_db2os390sqlj:}
  - Indicates that the connection is to a DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows server. \texttt{jdbc\_db2os390:} and \texttt{jdbc\_db2os390sqlj:} are for compatibility of programs that were written for older drivers, and apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS only.

- \texttt{jdbc\_default:connection}
  - Indicates that the URL is for a connection to the local subsystem through a DB2 thread that is controlled by CICS®, IMS®, or the Java stored procedure environment.

- \texttt{database}
  - A name for the database server.
  - \texttt{database} is the database name that is defined during installation, if the value of the serverName connection property is null. If the value of serverName property is not null, \texttt{database} is a database alias.
property=value;
A property and its value for the JDBC connection. You can specify one or more
property and value pairs. Each property and value pair, including the last one,
must end with a semicolon (;). Do not include spaces or other white space
characters anywhere within the list of property and value strings.

Some properties with an int data type have predefined constant field values.
You must resolve constant field values to their integer values before you can
use those values in the url parameter. For example, you cannot use
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL in a url parameter. However,
you can build a URL string that includes
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL, and assign the URL string
to a String variable. Then you can use the String variable in the url parameter:

```java
String url =
"jdbc:db2:STLEC1" +
":user=dbadm;password=dbadm;" +
"traceLevel=" +
(com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";;"
Connection con =
java.sql.DriverManager.getConnection(url);
```

Connecting to a data source using the DataSource interface

If your applications need to be portable among data sources, you should use the
DataSource interface.

Using DriverManager to connect to a data source reduces portability because the
application must identify a specific JDBC driver class name and driver URL. The
driver class name and driver URL are specific to a JDBC vendor, driver
implementation, and data source.

When you connect to a data source using the DataSource interface, you use a
DataSource object.

The simplest way to use a DataSource object is to create and use the object in the
same application, as you do with the DriverManager interface. However, this
method does not provide portability.

The best way to use a DataSource object is for your system administrator to create
and manage it separately, using WebSphere Application Server or some other tool.
The program that creates and manages a DataSource object also uses the Java
Naming and Directory Interface (JNDI) to assign a logical name to the DataSource
object. The JDBC application that uses the DataSource object can then refer to the
object by its logical name, and does not need any information about the underlying
data source. In addition, your system administrator can modify the data source
attributes, and you do not need to change your application program.

To learn more about using WebSphere to deploy DataSource objects, go to this
URL on the web:
http://www.ibm.com/software/webservers/appserv/

To learn about deploying DataSource objects yourself, see "Creating and deploying
DataSource objects".

You can use the DataSource interface and the DriverManager interface in the same
application, but for maximum portability, it is recommended that you use only the
DataSource interface to obtain connections.
To obtain a connection using a DataSource object that the system administrator has already created and assigned a logical name to, follow these steps:

1. From your system administrator, obtain the logical name of the data source to which you need to connect.
2. Create a Context object to use in the next step. The Context interface is part of the Java Naming and Directory Interface (JNDI), not JDBC.
3. In your application program, use JNDI to get the DataSource object that is associated with the logical data source name.
4. Use the DataSource.getConnection method to obtain the connection.

   You can use one of the following forms of the getConnection method:
   
   ```java
   getConnection();
   getConnection(String user, String password);
   ```

   Use the second form if you need to specify a user ID and password for the connection that are different from the ones that were specified when the DataSource was deployed.

**Example of obtaining a connection using a DataSource object that was created by the system administrator:** In this example, the logical name of the data source that you need to connect to is `jdbc/sampledb`. The numbers to the right of selected statements correspond to the previously-described steps.

```java
import java.sql.*; // JDBC base
import javax.sql.*; // Additional methods for JDBC
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC and SQLJ

Context ctx=new InitialContext();
... // interfaces
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
Connection con=ds.getConnection();
```

*Figure 4. Obtaining a connection using a DataSource object*

**Example of creating and using a DataSource object in the same application:**

```java
import java.sql.*; // JDBC base
import javax.sql.*; // Additional methods for JDBC
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC and SQLJ

DB2SimpleDataSource dbds=new DB2SimpleDataSource();
dbds.setDatabaseName("dbloc1");
// Assign the location name
dbds.setDescription("Our Sample Database");
// Description for documentation
dbds.setUser("john");
// Assign the user ID
dbds.setPassword("dbadm");
// Assign the password
Connection con=dbds.getConnection();
// Create a Connection object
```

*Figure 5. Creating and using a DataSource object in the same application*

**Note**

1. Import the package that contains the implementation of the DataSource interface.
2. Creates a DB2SimpleDataSource object. DB2SimpleDataSource is one of the IBM Data Server Driver for JDBC and SQLJ implementations of the DataSource interface. See "Creating and deploying DataSource objects" for information on DB2's DataSource implementations.
The setDatabaseName, setDescription, setUser, and setPassword methods assign attributes to the DB2SimpleDataSource object. See “Properties for the IBM Data Server Driver for JDBC and SQLJ” for information about the attributes that you can set for a DB2SimpleDataSource object under the IBM Data Server Driver for JDBC and SQLJ.

Establishes a connection to the data source that DB2SimpleDataSource object dbds represents.

How to determine which type of IBM Data Server Driver for JDBC and SQLJ connectivity to use

The IBM Data Server Driver for JDBC and SQLJ supports two types of connectivity: type 2 connectivity and type 4 connectivity.

For the DriverManager interface, you specify the type of connectivity through the URL in the DriverManager.getConnection method. For the DataSource interface, you specify the type of connectivity through the driverType property.

The following table summarizes the differences between type 2 connectivity and type 4 connectivity:

<table>
<thead>
<tr>
<th>Function</th>
<th>IBM Data Server Driver for JDBC and SQLJ type 2 connectivity</th>
<th>IBM Data Server Driver for JDBC and SQLJ type 4 connectivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sysplex workload balancing and connection concentrator</td>
<td>Supported through DB2 Connect</td>
<td>Supported directly by the driver for a connection within a single JVM Supported through DB2 Connect across JVMs</td>
</tr>
<tr>
<td>Communication protocols</td>
<td>TCP/IP</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>Performance</td>
<td>Better for accessing a local DB2 server</td>
<td>Better for accessing a remote DB2 server</td>
</tr>
<tr>
<td>Installation</td>
<td>Requires installation of native libraries in addition to Java classes</td>
<td>Requires installation of Java classes only</td>
</tr>
<tr>
<td>Stored procedures</td>
<td>Can be used to call or execute stored procedures</td>
<td>Can be used only to call stored procedures</td>
</tr>
<tr>
<td>Distributed transaction processing (XA)</td>
<td>Supported</td>
<td>Supported</td>
</tr>
<tr>
<td>J2EE 1.4 compliance</td>
<td>Compliant</td>
<td>Compliant</td>
</tr>
</tbody>
</table>

The following points can help you determine which type of connectivity to use.

Use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity under these circumstances:
- Your JDBC or SQLJ application runs locally most of the time. Local applications have better performance with type 2 connectivity.
- You are running a Java stored procedure.

A stored procedure environment consists of two parts: a client program, from which you call a stored procedure, and a server program, which is the stored
You can call a stored procedure in a JDBC or SQLJ program that uses type 2 or type 4 connectivity, but you must run a Java stored procedure using type 2 connectivity.

Use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity under these circumstances:
- Your JDBC or SQLJ application runs remotely most of the time. Remote applications have better performance with type 4 connectivity.
- You are using IBM Data Server Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing support.

**JDBC connection objects**

When you connect to a data source by either connection method, you create a Connection object, which represents the connection to the data source.

You use this Connection object to do the following things:
- Create Statement, PreparedStatement, and CallableStatement objects for executing SQL statements. These are discussed in "Executing SQL statements in JDBC applications".
- Gather information about the data source to which you are connected. This process is discussed in "Learning about a data source using DatabaseMetaData methods".
- Commit or roll back transactions. You can commit transactions manually or automatically. These operations are discussed in "Commit or roll back a JDBC transaction".
- Close the connection to the data source. This operation is discussed in "Disconnecting from data sources in JDBC applications".

**Creating and deploying DataSource objects**

JDBC versions starting with version 2.0 provide the DataSource interface for connecting to a data source. Using the DataSource interface is the preferred way to connect to a data source.

Using the DataSource interface involves two parts:
- Creating and deploying DataSource objects. This is usually done by a system administrator, using a tool such as WebSphere Application Server.
- Using the DataSource objects to create a connection. This is done in the application program.

This topic contains information that you need if you create and deploy the DataSource objects yourself.

The IBM Data Server Driver for JDBC and SQLJ provides the following DataSource implementations:
- com.ibm.db2.jcc.DB2SimpleDataSource, which does not support connection pooling. You can use this implementation with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
- com.ibm.db2.jcc.DB2ConnectionPoolDataSource, which supports connection pooling. You can use this implementation with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
• com.ibm.db2.jcc.DB2XADatasource, which supports connection pooling and distributed transactions. The connection pooling is provided by WebSphere Application Server or another application server. You can use this implementation only with IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

The DB2 JDBC Type 2 Driver provides the following DataSource implementations:
• COM.ibm.db2.jdbc.DB2DataSource, which is enabled for connection pooling. With this implementation, connection pooling is handled internally and is transparent to the application.
• COM.ibm.db2.jdbc.DB2XADatasource, which does not have built-in support for distributed transactions and connection pooling. With this implementation, you must manage the distributed transactions and connection pooling yourself, either by writing your own code or by using a tool such as WebSphere Application Server.

When you create and deploy a DataSource object, you need to perform these tasks:
1. Create an instance of the appropriate DataSource implementation.
2. Set the properties of the DataSource object.
3. Register the object with the Java Naming and Directory Interface (JNDI) naming service.

The following example shows how to perform these tasks.

```java
import java.sql.*; // JDBC base
import javax.naming.*; // JNDI Naming Services
import javax.sql.*; // Additional methods for JDBC
import com.ibm.db2.jcc.*; // IBM Data Server Driver for
// JDBC and SQLJ
// implementation of JDBC
// standard extension APIs

DB2SimpleDataSource dbds = new com.ibm.db2.jcc.DB2SimpleDataSource();  // Creates an instance of the DB2SimpleDataSource class.
dbds.setDatabaseName("db2loc1");  // This statement and the next three statements set values for properties of this
dbds.setDescription("Our Sample Database");  // DB2SimpleDataSource object.
dbds.setUser("john");
dbds.setPassword("mypw");
...
Context ctx=new InitialContext();  // Creates a context for use by JNDI.
Ctx.bind("jdbc/sampledb",dbds);  // Associates DBSimple2DataSource object dbds with the logical name
cjdbc/sampledb. An application that uses this object can refer to it by the name
cjdbc/sampledb.
```

Figure 6. Example of creating and deploying a DataSource object

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Creates an instance of the DB2SimpleDataSource class.</td>
</tr>
<tr>
<td>2</td>
<td>This statement and the next three statements set values for properties of this DB2SimpleDataSource object.</td>
</tr>
<tr>
<td>3</td>
<td>Creates a context for use by JNDI.</td>
</tr>
<tr>
<td>4</td>
<td>Associates DBSimple2DataSource object dbds with the logical name jdbc/sampledb. An application that uses this object can refer to it by the name jdbc/sampledb.</td>
</tr>
</tbody>
</table>

### Java packages for JDBC support

Before you can invoke JDBC methods, you need to be able to access all or parts of various Java packages that contain those methods.
You can do that either by importing the packages or specific classes, or by using the fully-qualified class names. You might need the following packages or classes for your JDBC program:

java.sql
   Contains the core JDBC API.

javax.naming
   Contains classes and interfaces for Java Naming and Directory Interface (JNDI), which is often used for implementing a DataSource.

javax.sql
   Contains methods for producing server-side applications using Java

javax.transaction
   Contains JDBC support for distributed transactions for the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver).

com.ibm.db2.jcc
   Contains the implementation of JDBC for the IBM Data Server Driver for JDBC and SQLJ.

COM.ibm.db2.jdbc
   Contains the implementation of the JDBC for the DB2 JDBC Type 2 Driver.

---

Learning about a data source using DatabaseMetaData methods

The DatabaseMetaData interface contains methods that retrieve information about a data source. These methods are useful when you write generic applications that can access various data sources.

In generic applications that can access various data sources, you need to test whether a data source can handle various database operations before you execute them. For example, you need to determine whether the driver at a data source is at the JDBC 3.0 level before you invoke JDBC 3.0 methods against that driver.

DatabaseMetaData methods provide the following types of information:

- Features that the data source supports, such as the ANSI SQL level
- Specific information about the JDBC driver, such as the driver level
- Limits, such as the maximum number of columns that an index can have
- Whether the data source supports data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE)
- Lists of objects at the data source, such as tables, indexes, or procedures
- Whether the data source supports various JDBC functions, such as batch updates or scrollable ResultSets
- A list of scalar functions that the driver supports

To invoke DatabaseMetaData methods, you need to perform these basic steps:

1. Create a DatabaseMetaData object by invoking the getMetaData method on the connection.
2. Invoke DatabaseMetaData methods to get information about the data source.
3. If the method returns a ResultSet:

   a. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.
   b. Invoke the close method to close the ResultSet object.
Example: The following code demonstrates how to use DatabaseMetaData methods to determine the driver version, to get a list of the stored procedures that are available at the data source, and to get a list of datetime functions that the driver supports. The numbers to the right of selected statements correspond to the previously-described steps.

```
Connection con;
DatabaseMetaData dbmtadta;
ResultSet rs;
int mtadtaint;
String procSchema;
String procName;
String dtfnList;
...
```

```
dbmtadta = con.getMetaData(); // Create the DatabaseMetaData object
mtadtaint = dmtadta.getDriverVersion(); // Check the driver version
    System.out.println("Driver version: "+ mtadtaint);
rs = dbmtadta.getProcedures(null, null, ";"); // Get information for all procedures
    System.out.println("Driver version: "+ mtadtaint);
while (rs.next()) { // Position the cursor
    procSchema = rs.getString("PROCEDURE_SCHEM"); // Get procedure schema
    procName = rs.getString("PROCEDURE_NAME"); // Get procedure name
    System.out.println(procSchema + "." + procName); // Print the qualified procedure name
}
dtfnList = dbmtadta.getTimeDateFunctions(); // Get list of supported datetime functions
    System.out.println("Supported datetime functions:");
System.out.println(dtfnList); // Print the list of datetime functions
rs.close(); // Close the ResultSet
```

DatabaseMetaData methods for identifying the type of data source

You can use the DatabaseMetaData.getDatabaseProductName and DatabaseMetaData.getProductVersion methods to identify the type and level of the database manager to which you are connected, and the operating system on which the database manager is running.

DatabaseMetaData.getDatabaseProductName returns a string that identifies the database manager and the operating system. The string has one of the following formats:

- `database-product`
- `database-product/operating-system`

The following table shows examples of values that are returned by DatabaseMetaData.getDatabaseProductName.

<table>
<thead>
<tr>
<th>getDatabaseProductName value</th>
<th>Database product</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2</td>
<td>DB2 for z/OS</td>
</tr>
<tr>
<td>DB2/LINUXX8664</td>
<td>DB2 Database for Linux, UNIX, and Windows on Linux on x86</td>
</tr>
<tr>
<td>IDS/UNIX64</td>
<td>IBM Informix (IDS) on UNIX</td>
</tr>
</tbody>
</table>
DatabaseMetaData.getDatabaseVersionName returns a string that contains the
database product indicator and the version number, release number, and
maintenance level of the data source.

The following table shows examples of values that are returned by
DatabaseMetaData.getDatabaseProductVersion.

<table>
<thead>
<tr>
<th>getDatabaseProductVersion value</th>
<th>Database product version</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSN09015</td>
<td>DB2 for z/OS Version 9.1 in new-function mode</td>
</tr>
<tr>
<td>SQL09010</td>
<td>DB2 Database for Linux, UNIX, and Windows Version 9.1</td>
</tr>
<tr>
<td>I FX11100</td>
<td>IDS Version 11.10</td>
</tr>
</tbody>
</table>

Variables in JDBC applications

As in any other Java application, when you write JDBC applications, you declare
variables. In Java applications, those variables are known as Java identifiers.

Some of those identifiers have the same function as host variables in other
languages: they hold data that you pass to or retrieve from database tables.
Identifier empNo in the following code holds data that you retrieve from the
EMPNO table column, which has the CHAR data type.

```java
String empNo;
// Execute a query and generate a ResultSet instance
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");
while (rs.next()) {
  String empNo = rs.getString(1);
  System.out.println("Employee number = " + empNo);
}
```

Your choice of Java data types can affect performance because DB2 picks better
access paths when the data types of your Java variables map closely to the DB2
data types.

JDBC interfaces for executing SQL

You execute SQL statements in a traditional SQL program to update data in tables,
retrieve data from the tables, or call stored procedures. To perform the same
functions in a JDBC program, you invoke methods.

Those methods are defined in the following interfaces:

- The Statement interface supports all SQL statement execution. The following
  interfaces inherit methods from the Statement interface:
  - The PreparedStatement interface supports any SQL statement containing
    input parameter markers. Parameter markers represent input variables. The
    PreparedStatement interface can also be used for SQL statements with no
    parameter markers.
    With the IBM Data Server Driver for JDBC and SQLJ, the PreparedStatement
    interface can be used to call stored procedures that have input parameters
    and no output parameters, and that return no result sets. However, the
    preferred interface is CallableStatement.
  - The CallableStatement interface supports the invocation of a stored procedure.
The CallableStatement interface can be used to call stored procedures with input parameters, output parameters, or input and output parameters, or no parameters. With the IBM Data Server Driver for JDBC and SQLJ, you can also use the Statement interface to call stored procedures, but those stored procedures must have no parameters.

- The ResultSet interface provides access to the results that a query generates. The ResultSet interface has the same purpose as the cursor that is used in SQL applications in other languages.

Creating and modifying database objects using the Statement.executeUpdate method

The Statement.executeUpdate is one of the JDBC methods that you can use to update tables and call stored procedures.

You can use the Statement.executeUpdate method to do the following things:
- Execute data definition statements, such as CREATE, ALTER, DROP, GRANT, REVOKE
- Execute INSERT, UPDATE, DELETE, and MERGE statements that do not contain parameter markers.
- With the IBM Data Server Driver for JDBC and SQLJ, execute the CALL statement to call stored procedures that have no parameters and that return no result sets.

To execute these SQL statements, you need to perform these steps:
1. Invoke the Connection.createStatement method to create a Statement object.
2. Invoke the Statement.executeUpdate method to perform the SQL operation.
3. Invoke the Statement.close method to close the Statement object.

Suppose that you want to execute this SQL statement:
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'

The following code creates Statement object stmt, executes the UPDATE statement, and returns the number of rows that were updated in numUpd. The numbers to the right of selected statements correspond to the previously-described steps.

```java
Connection con;
Statement stmt;
int numUpd;
...
stmt = con.createStatement(); // Create a Statement object 1
numUpd = stmt.executeUpdate(
    "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'"'); 2
    // Perform the update
stmt.close(); // Close Statement object 3
```

Figure 8. Using Statement.executeUpdate

Updating data in tables using the PreparedStatement.executeUpdate method

The Statement.executeUpdate method works if you update DB2 tables with constant values. However, updates often need to involve passing values in variables to DB2 tables. To do that, you use the PreparedStatement.executeUpdate method.
With the IBM Data Server Driver for JDBC and SQLJ, you can also use PreparedStatement.executeUpdate to call stored procedures that have input parameters and no output parameters, and that return no result sets.

DB2 for z/OS does not support dynamic execution of the CALL statement. For calls to stored procedures that are on DB2 for z/OS data sources, the parameters can be parameter markers or literals, but not expressions. The following types of literals are supported:

- Integer
- Double
- Decimal
- Character
- Hexadecimal
- Graphic

For calls to stored procedures that are on IBM Informix data sources, the PreparedStatement object can be a CALL statement or an EXECUTE PROCEDURE statement.

When you execute an SQL statement many times, you can get better performance by creating the SQL statement as a PreparedStatement.

For example, the following UPDATE statement lets you update the employee table for only one phone number and one employee number:

```
UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'
```

Suppose that you want to generalize the operation to update the employee table for any set of phone numbers and employee numbers. You need to replace the constant phone number and employee number with variables:

```
UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?
```

Variables of this form are called parameter markers. To execute an SQL statement with parameter markers, you need to perform these steps:

1. Invoke the Connection.prepareStatement method to create a PreparedStatement object.
2. Invoke the PreparedStatement.setXXX methods to pass values to the input variables.
   - This step assumes that you use standard parameter markers. Alternatively, if you use named parameter markers, you use IBM Data Server Driver for JDBC and SQLJ-only methods to pass values to the input parameters.
3. Invoke the PreparedStatement.executeUpdate method to update the table with the variable values.
4. Invoke the PreparedStatement.close method to close the PreparedStatement object when you have finished using that object.

The following code performs the previous steps to update the phone number to '4657' for the employee with employee number '000010'. The numbers to the right of selected statements correspond to the previously-described steps.
You can also use the PreparedStatement.executeUpdate method for statements that have no parameter markers. The steps for executing a PreparedStatement object with no parameter markers are similar to executing a PreparedStatement object with parameter markers, except you skip step 2 on page 39. The following example demonstrates these steps.

```java
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement("UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");  // Create a PreparedStatement object
pstmt.setString(1,"4657");  // Assign first value to first parameter
pstmt.setString(2,"000010");  // Assign first value to second parameter
numUpd = pstmt.executeUpdate();  // Perform first update
pstmt.setString(1,"4658");  // Assign second value to first parameter
pstmt.setString(2,"000020");  // Assign second value to second parameter
numUpd = pstmt.executeUpdate();  // Perform second update
pstmt.close();  // Close the PreparedStatement object
```

Figure 9. Using PreparedStatement.executeUpdate for an SQL statement with parameter markers

You can also use the PreparedStatement.executeUpdate method for statements that have no parameter markers. The steps for executing a PreparedStatement object with no parameter markers are similar to executing a PreparedStatement object with parameter markers, except you skip step 2 on page 39. The following example demonstrates these steps.

```java
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement("UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");  // Create a PreparedStatement object
numUpd = pstmt.executeUpdate();  // Perform the update
pstmt.close();  // Close the PreparedStatement object
```

Figure 10. Using PreparedStatement.executeUpdate for an SQL statement without parameter markers

**JDBC executeUpdate methods against a DB2 for z/OS server**

The JDBC standard states that the executeUpdate method returns a row count or 0. However, if the executeUpdate method is executed against a DB2 for z/OS server, it can return a value of -1.

For executeUpdate statements against a DB2 for z/OS server, the value that is returned depends on the type of SQL statement that is being executed:

- For an SQL statement that can have an update count, such as an INSERT, UPDATE, DELETE, or MERGE statement, the returned value is the number of affected rows. It can be:
  - A positive number, if a positive number of rows are affected by the operation, and the operation is not a mass delete on a segmented table space.
  - 0, if no rows are affected by the operation.
  - -1, if the operation is a mass delete on a segmented table space.
- For an SQL CALL statement, a value of -1 is returned, because the data source cannot determine the number of affected rows. Calls to getUpdateCount or getMoreResults for a CALL statement also return -1.
- For any other SQL statement, a value of -1 is returned.
Making batch updates in JDBC applications

With batch updates, instead of updating rows of a table one at a time, you can direct JDBC to execute a group of updates at the same time. Statements that can be included in the same batch of updates are known as batchable statements.

If a statement has input parameters or host expressions, you can include that statement only in a batch that has other instances of the same statement. This type of batch is known as a homogeneous batch. If a statement has no input parameters, you can include that statement in a batch only if the other statements in the batch have no input parameters or host expressions. This type of batch is known as a heterogeneous batch. Two statements that can be included in the same batch are known as batch compatible.

Use the following Statement methods for creating, executing, and removing a batch of SQL updates:
- addBatch
- executeBatch
- clearBatch

Use the following PreparedStatement and CallableStatement method for creating a batch of parameters so that a single statement can be executed multiple times in a batch, with a different set of parameters for each execution.
- addBatch

Restrictions on executing statements in a batch:
- If you try to execute a SELECT statement in a batch, a BatchUpdateException is thrown.
- A CallableStatement object that you execute in a batch can contain output parameters. However, you cannot retrieve the values of the output parameters. If you try to do so, a BatchUpdateException is thrown.
- You cannot retrieve ResultSet objects from a CallableStatement object that you execute in a batch. A BatchUpdateException is not thrown, but the getResultSet method invocation returns a null value.

To make batch updates using several statements with no input parameters, follow these basic steps:
1. For each SQL statement that you want to execute in the batch, invoke the addBatch method.
2. Invoke the executeBatch method to execute the batch of statements.
3. Check for errors. If no errors occurred:
   a. Get the number of rows that were affect by each SQL statement from the array that the executeBatch invocation returns. This number does not include rows that were affected by triggers or by referential integrity enforcement.
   b. If AutoCommit is disabled for the Connection object, invoke the commit method to commit the changes.

To make batch updates using a single statement with several sets of input parameters, follow these basic steps:
1. If the batched statement is a searched UPDATE, searched DELETE, or MERGE statement, set the autocommit mode for the connection to false.
2. Invoke the prepareStatement method to create a PreparedStatement object.
3. For each set of input parameter values:
   a. Execute setXXX methods to assign values to the input parameters.
   b. Invoke the addBatch method to add the set of input parameters to the batch.
4. Invoke the executeBatch method to execute the statements with all sets of parameters.
5. If no errors occurred:
   a. Get the number of rows that were updated by each execution of the SQL statement from the array that the executeBatch invocation returns. The number of affected rows does not include rows that were affected by triggers or by referential integrity enforcement.
      If the following conditions are true, the IBM Data Server Driver for JDBC and SQLJ returns Statement.SUCCESS_NO_INFO (-2), instead of the number of rows that were affected by each SQL statement:
      • The application is connected to a subsystem that is in DB2 for z/OS Version 8 new-function mode, or later.
      • The application is using Version 3.1 or later of the IBM Data Server Driver for JDBC and SQLJ.
      • The IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT operations to execute batch updates.
      This occurs because with multi-row INSERT, the database server executes the entire batch as a single operation, so it does not return results for individual SQL statements.
   b. If AutoCommit is disabled for the Connection object, invoke the commit method to commit the changes.
      If AutoCommit is enabled for the Connection object, the IBM Data Server Driver for JDBC and SQLJ adds a commit method at the end of the batch.
   c. If the PreparedStatement object returns automatically generated keys, call DB2PreparedStatement.getDBGeneratedKeys to retrieve an array of ResultSet objects that contains the automatically generated keys.
      Check the length of the returned array. If the length of the returned array is 0, an error occurred during retrieval of the automatically generated keys.
6. If errors occurred, process the BatchUpdateException.

In the following code fragment, two sets of parameters are batched. An UPDATE statement that takes two input parameters is then executed twice, once with each set of parameters. The numbers to the right of selected statements correspond to the previously-described steps.

```java
try {
    ...
    PreparedStatement preps = conn.prepareStatement(
        "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");  // 2
    ps.setString(1,mgrnum1);  // 3a
    ps.setString(2,deptnum1);
    ps.addBatch();  // 3b

    ps.setString(1,mgrnum2);
    ps.setString(2,deptnum2);
    ps.addBatch();

    int [] numUpdates=ps.executeBatch();  // 4
    for (int i=0; i < numUpdates.length; i++) {  // 5
        if (numUpdates[i] == SUCCESS_NO_INFO)
            System.out.println("Execution": + i + ": unknown number of rows updated");
        else
```
In the following code fragment, a batched INSERT statement returns automatically generated keys.

```java
import java.sql.*;
import com.ibm.db2.jcc.*;
...
Connection conn;
...
try {
  PreparedStatement ps = conn.prepareStatement(
      "INSERT INTO DEPT (DEPTNO, DEPTNAME, ADMRDEPT) VALUES (?, ?, ?)"
  .withProperties(Statement.RETURN_GENERATED_KEYS));
  ps.setString(1, "X01");
  ps.setString(2, "Finance");
  ps.setString(3, "A00");
  ps.addBatch();
  ps.setString(1, "Y01");
  ps.setString(2, "Accounting");
  ps.setString(3, "A00");
  ps.addBatch();
  int[] numUpdates = ps.executeBatch();
  for (int i = 0; i < numUpdates.length; i++) {
    if (numUpdates[i] == SUCCESS_NO_INFO)
      System.out.println("Execution "+i+" successful: "+numUpdates[i]+" rows updated");
    else
      System.out.println("Execution "+i+" unknown number of rows updated");
  }
  conn.commit();
  ResultSet[] resultList = ((DB2PreparedStatement)ps).getDBGeneratedKeys();
  if (resultList.length != 0) {
    for (i = 0; i < resultList.length; i++) {
      while (resultList[i].next()) {
        java.math.BigDecimal idColVar = rs.getBigDecimal(1);
        System.out.println("Automatically generated key value = "+idColVar);
      }
    }
  } else {
    System.out.println("Error retrieving automatically generated keys");
  }
} catch(BatchUpdateException b) {
  // process BatchUpdateException
}
```

In the following code fragment, a batched UPDATE statement returns automatically generated keys. The code names the DEPTNO column as an automatically generated key, updates two rows in the DEPT table in a batch, and
retrieves the values of DEPTNO for the updated rows. The numbers to the right of selected statements correspond to the previously described steps.

```java
import java.sql.*;
import com.ibm.db2.jcc.*;
...
Connection conn;
...
String[] agkNames = {"DEPTNO"};
try {
...
    conn.setAutoCommit(false);
    PreparedStatement ps = conn.prepareStatement("UPDATE DEPT SET DEPTNAME=? WHERE DEPTNO=?",agkNames);
    ps.setString(1,"X01");
    ps.setString(2,"Planning");
    ps.addBatch();
    ps.setString(1,"Y01");
    ps.setString(2,"Bookkeeping");
    ps.addBatch();
    int[] numUpdates=ps.executeBatch();
    for (int i=0; i < numUpdates.length; i++) {
        if (numUpdates[i] == SUCCESS_NO_INFO)
            System.out.println("Execution" + i + ": unknown number of rows updated");
        else
            System.out.println("Execution " + i + "successful: " numUpdates[i] + " rows updated");
    }
    conn.commit();
    ResultSet[] resultList = ((DB2PreparedStatement)ps).getDBGeneratedKeys();
    if (resultList.length != 0) {
        for (i = 0; i < resultList.length; i++) {
            while (resultList[i].next()) {
                String deptNoKey = rs.getString(1);
                // Get automatically generated key
                // value
                System.out.println("Automatically generated key value = " + deptNoKey);
            }
        }
    } else {
        System.out.println("Error retrieving automatically generated keys");
    }
} catch(BatchUpdateException b) {
    // process BatchUpdateException
}
```

**Learning about parameters in a PreparedStatement using ParameterMetaData methods**

The IBM Data Server Driver for JDBC and SQLJ includes support for the ParameterMetaData interface. The ParameterMetaData interface contains methods that retrieve information about the parameter markers in a PreparedStatement object.

ParameterMetaData methods provide the following types of information:
- The data types of parameters, including the precision and scale of decimal parameters.
• The parameters’ database-specific type names. For parameters that correspond to
table columns that are defined with distinct types, these names are the distinct
type names.
• Whether parameters are nullable.
• Whether parameters are input or output parameters.
• Whether the values of a numeric parameter can be signed.
• The fully-qualified Java class name that PreparedStatement.setObject uses when
it sets a parameter value.

To invoke ParameterMetaData methods, you need to perform these basic steps:
1. Invoke the Connection.prepareStatement method to create a PreparedStatement
object.
2. Invoke the PreparedStatement.getParameterMetaData method to retrieve a
ParameterMetaData object.
3. Invoke ParameterMetaData.getParameterCount to determine the number of
parameters in the PreparedStatement.
4. Invoke ParameterMetaData methods on individual parameters.

The following code demonstrates how to use ParameterMetaData methods to
determine the number and data types of parameters in an SQL UPDATE statement.
The numbers to the right of selected statements correspond to the
previously-described steps.

```java
Connection con;
ParameterMetaData pmtadta;
int mtadtacnt;
String sqlType;
...
pstmt = con.prepareStatement(
   "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?"); // Create a PreparedStatement object
   // Create a ParameterMetaData object
pmtadta = pstmt.getParameterMetaData();
mtadtacnt = pmtadta.getParameterCount(); // Determine the number of parameters
System.out.println("Number of statement parameters: " + mtadtacnt);
for (int i = 1; i <= mtadtacnt; i++) {
   sqlType = pmtadta.getParameterTypeName(i); // Get SQL type for each parameter
   System.out.println("SQL type of parameter " + i " is " + sqlType);
}
...
pstmt.close(); // Close the PreparedStatement
```

Figure 11. Using ParameterMetaData methods to get information about a PreparedStatement

Data retrieval in JDBC applications

In JDBC applications, you retrieve data using ResultSet objects. A ResultSet
represents the result set of a query.

Retrieving data from tables using the Statement.executeQuery method

To retrieve data from a table using a SELECT statement with no parameter
markers, you can use the Statement.executeQuery method.

This method returns a result table in a ResultSet object. After you obtain the result
table, you need to use ResultSet methods to move through the result table and
obtain the individual column values from each row.
With the IBM Data Server Driver for JDBC and SQLJ, you can also use the `Statement.executeQuery` method to retrieve a result set from a stored procedure call, if that stored procedure returns only one result set. If the stored procedure returns multiple result sets, you need to use the `Statement.execute` method.

This topic discusses the simplest kind of ResultSet, which is a read-only ResultSet in which you can only move forward, one row at a time. The IBM Data Server Driver for JDBC and SQLJ also supports updatable and scrollable ResultSets.

To retrieve rows from a table using a SELECT statement with no parameter markers, you need to perform these steps:

1. Invoke the `Connection.createStatement` method to create a `Statement` object.
2. Invoke the `Statement.executeQuery` method to obtain the result table from the SELECT statement in a ResultSet object.
3. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using `getXXX` methods. `XXX` represents a data type.
4. Invoke the `ResultSet.close` method to close the ResultSet object.
5. Invoke the `Statement.close` method to close the Statement object when you have finished using that object.

The following code demonstrates how to retrieve all rows from the employee table. The numbers to the right of selected statements correspond to the previously-described steps.

```java
String empNo;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();  // Create a Statement object 1
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");  // Get the result table from the query 2
while (rs.next())  // Position the cursor 3
    empNo = rs.getString(1);  // Retrieve only the first column value
    System.out.println("Employee number = " + empNo);
    // Print the column value
rs.close();  // Close the ResultSet 4
stmt.close();  // Close the Statement 5
```

Figure 12. Using `Statement.executeQuery`

**Retrieving data from tables using the `PreparedStatement.executeQuery` method**

To retrieve data from a table using a SELECT statement with parameter markers, you use the `PreparedStatement.executeQuery` method.

This method returns a result table in a ResultSet object. After you obtain the result table, you need to use ResultSet methods to move through the result table and obtain the individual column values from each row.

With the IBM Data Server Driver for JDBC and SQLJ, you can also use the `PreparedStatement.executeQuery` method to retrieve a result set from a stored procedure call, if that stored procedure returns only one result set and has only input parameters. If the stored procedure returns multiple result sets, you need to use the `PreparedStatement.execute` method.
You can also use the PreparedStatement.executeQuery method for statements that have no parameter markers. When you execute a query many times, you can get better performance by creating the SQL statement as a PreparedStatement.

To retrieve rows from a table using a SELECT statement with parameter markers, you need to perform these steps:

1. Invoke the Connection.prepareStatement method to create a PreparedStatement object.
2. Invoke PreparedStatement.setXXX methods to pass values to the input parameters.
3. Invoke the PreparedStatement.executeQuery method to obtain the result table from the SELECT statement in a ResultSet object.
4. In a loop, position the cursor using the ResultSet.next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.
5. Invoke the ResultSet.close method to close the ResultSet object.
6. Invoke the PreparedStatement.close method to close the PreparedStatement object when you have finished using that object.

The following code demonstrates how to retrieve rows from the employee table for a specific employee. The numbers to the right of selected statements correspond to the previously-described steps.

```java
String empnum, phonenum;
Connection con;
PreparedStatement pstmt;
ResultSet rs;
...
pstmt = con.prepareStatement("SELECT EMPNO, PHONENO FROM EMPLOYEE WHERE EMPNO=?");
 pstmt.setString(1,"000010"); // Create a PreparedStatement object
 rs = pstmt.executeQuery(); // Get the result table from the query
 while (rs.next()) { // Position the cursor
   empnum = rs.getString(1); // Retrieve the first column value
   phonenum = rs.getString(2); // Retrieve the first column value
   System.out.println("Employee number = " + empnum +
   "Phone number = " + phonenum); // Print the column values
 }
 rs.close(); // Close the ResultSet
 pstmt.close(); // Close the PreparedStatement
```

Figure 13. Example of using PreparedStatement.executeQuery

**Making batch queries in JDBC applications**

The IBM Data Server Driver for JDBC and SQLJ provides a IBM Data Server Driver for JDBC and SQLJ-only DB2PreparedStatement interface that lets you perform batch queries on a homogeneous batch.

To make batch queries using a single statement with several sets of input parameters, follow these basic steps:

1. Invoke the prepareStatement method to create a PreparedStatement object for the SQL statement with input parameters.
2. For each set of input parameter values:
a. Execute PreparedStatement.setXXX methods to assign values to the input parameters.
b. Invoke the PreparedStatement.addBatch method to add the set of input parameters to the batch.

3. Cast the PreparedStatement object to a DB2PreparedStatement object, and invoke the DB2PreparedStatement.executeDB2QueryBatch method to execute the statement with all sets of parameters.

4. In a loop, retrieve the ResultSet objects:
   a. Retrieve each ResultSet object.
   b. Retrieve all the rows from each ResultSet object.

**Example:** In the following code fragment, two sets of parameters are batched. A SELECT statement that takes one input parameter is then executed twice, once with each parameter value. The numbers to the right of selected statements correspond to the previously described steps.

```java
java.sql.Connection con = java.sql.DriverManager.getConnection(url, properties);
java.sql.Statement s = con.createStatement();
// Clean up from previous executions
try {
    s.executeUpdate("drop table TestQBatch");
} catch (Exception e) {
}

// Create and populate a test table
s.executeUpdate("create table TestQBatch (col1 int, col2 char(10))");
s.executeUpdate("insert into TestQBatch values (1, 'test1')");
s.executeUpdate("insert into TestQBatch values (2, 'test2')");
s.executeUpdate("insert into TestQBatch values (3, 'test3')");
s.executeUpdate("insert into TestQBatch values (4, 'test4')");
s.executeUpdate("insert into TestQBatch values (1, 'test5')");
s.executeUpdate("insert into TestQBatch values (2, 'test6')");

try {
    PreparedStatement pstmt =
        con.prepareStatement("Select * from TestQBatch where col1 = ?");
    pstmt.setInt(1, 1);
    pstmt.addBatch();
    // Add some more values to the batch
    pstmt.setInt(1, 2);
    pstmt.addBatch();
    pstmt.setInt(1, 3);
    pstmt.addBatch();
    pstmt.setInt(1, 4);
    pstmt.addBatch();
    ((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).executeDB2QueryBatch();
    catch (BatchUpdateException b) {
        // process BatchUpdateException
    }
    ResultSet rs;
    while (pstmt.getMoreResults()) {
        rs = pstmt.getResultSet();
        while (rs.next()) {
            System.out.print (rs.getInt (1) + " ");
            System.out.println (rs.getString (2));
        }
        System.out.println();
        rs.close();
    }
} catch(BatchUpdateException b) {
    // process BatchUpdateException
}
```

48 Developing Java Applications
Learning about a ResultSet using ResultSetMetaData methods

You cannot always know the number of columns and data types of the columns in a table or result set. This is true especially when you are retrieving data from a remote data source.

When you write programs that retrieve unknown ResultSets, you need to use ResultSetMetaData methods to determine the characteristics of the ResultSets before you can retrieve data from them.

ResultSetMetaData methods provide the following types of information:
- The number of columns in a ResultSet
- The qualifier for the underlying table of the ResultSet
- Information about a column, such as the data type, length, precision, scale, and nullability
- Whether a column is read-only

After you invoke the executeQuery method to generate a ResultSet for a query on a table, follow these basic steps to determine the contents of the ResultSet:
1. Invoke the getMetaData method on the ResultSet object to create a ResultSetMetaData object.
2. Invoke the getColumnCount method to determine how many columns are in the ResultSet.
3. For each column in the ResultSet, execute ResultSetMetaData methods to determine column characteristics.
   - The results of ResultSetMetaData.getColumnName call reflects the column name information that is stored in the DB2 catalog for that data source.

The following code demonstrates how to determine the data types of all the columns in the employee table. The numbers to the right of selected statements correspond to the previously-described steps.
The IBM Data Server Driver for JDBC and SQLJ provides support for scrollable, updatable, and holdable cursors.

In addition to moving forward, one row at a time, through a ResultSet, you might want to do the following things:

- Move backward or go directly to a specific row
- Update, delete, or insert rows in a ResultSet
- Leave the ResultSet open after a COMMIT

The following terms describe characteristics of a ResultSet:

**scrollability**
Whether the cursor for the ResultSet can move forward only, or forward one or more rows, backward one or more rows, or to a specific row.

If a cursor for a ResultSet is scrollable, it also has a sensitivity attribute, which describes whether the cursor is sensitive to changes to the underlying table.

**updatability**
Whether the cursor can be used to update or delete rows. This characteristic does not apply to a ResultSet that is returned from a stored procedure, because a stored procedure ResultSet cannot be updated.

**holdability**
Whether the cursor stays open after a COMMIT.

You set the updatability, scrollability, and holdability characteristics of a ResultSet through parameters in the `Connection.prepareStatement` or `Connection.createStatement` methods. The ResultSet settings map to attributes of a

```java
String s;
Connection con;
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmtadta;
int colCount;
int mtadaint;
int i;
String colName;
String colType;
...
stmt = con.createStatement(); // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE"); // Get the ResultSet from the query
rsmtadta = rs.getMetaData(); // Create a ResultSetMetaData object
colCount = rsmtadta.getColumnCount(); // Find number of columns in EMP
for (i=1; i<=colCount; i++) {
    colName = rsmtadta.getColumnName(); // Get column name
    colType = rsmtadta.getColumnTypeName(); // Get column data type
    System.out.println("Column = " + colName + 
    " is data type " + colType); // Print the column value
}
```

*Figure 14. Using ResultSetMetaData methods to get information about a ResultSet*
cursor in the database. The following table lists the JDBC scrollability, updatability, and holdability settings, and the corresponding cursor attributes.

Table 9. JDBC ResultSet characteristics and SQL cursor attributes

<table>
<thead>
<tr>
<th>JDBC setting</th>
<th>DB2 cursor setting</th>
<th>IBM Informix cursor setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONCUR_READ_ONLY</td>
<td>FOR READ ONLY</td>
<td>FOR READ ONLY</td>
</tr>
<tr>
<td>CONCUR_UPDATABLE</td>
<td>FOR UPDATE</td>
<td>FOR UPDATE</td>
</tr>
<tr>
<td>HOLD_CURSORS_OVER_COMMIT</td>
<td>WITH HOLD</td>
<td>WITH HOLD</td>
</tr>
<tr>
<td>TYPE_FORWARD_ONLY</td>
<td>SCROLL not specified</td>
<td>SCROLL not specified</td>
</tr>
<tr>
<td>TYPE_SCROLL_INSENSITIVE</td>
<td>INSENSITIVE SCROLL</td>
<td>SCROLL</td>
</tr>
</tbody>
</table>
| TYPE_SCROLL_SENSITIVE         | SENSITIVE STATIC, SENSITIVE
                                  DYNAMIC, or ASENSITIVE,
                                  depending on the cursorSensitvity
                                  Connection and DataSource property
                                  Not supported

If a JDBC ResultSet is static, the size of the result table and the order of the rows in the result table do not change after the cursor is opened. This means that if you insert rows into the underlying table, the result table for a static ResultSet does not change. If you delete a row of a result table, a delete hole occurs. You cannot update or delete a delete hole.

Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications:

You use special parameters in the Connection.prepareStatement or Connection.createStatement methods to specify the updatability, scrollability, and holdability of a ResultSet.

By default, ResultSet objects are not scrollable and not updatable. The default holdability depends on the data source, and can be determined from the DatabaseMetaData.getResultSetHoldability method. To change the scrollability, updatability, and holdability attributes for a ResultSet, follow these steps:

1. If the SELECT statement that defines the ResultSet has no input parameters, invoke the createStatement method to create a Statement object. Otherwise, invoke the prepareStatement method to create a PreparedStatement object. You need to specify forms of the createStatement or prepareStatement methods that include the resultSetType, resultSetConcurrency, or resultSetHoldability parameters.

The form of the createStatement method that supports scrollability and updatability is:

```java
createStatement(int resultSetType, int resultSetConcurrency);
```

The form of the createStatement method that supports scrollability, updatability, and holdability is:

```java
createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability);
```

The form of the prepareStatement method that supports scrollability and updatability is:

```java
prepareStatement(String sql, int resultSetType, int resultSetConcurrency);
```

The form of the prepareStatement method that supports scrollability, updatability, and holdability is:
prepareStatement(String sql, int resultSetType,
    int resultSetConcurrency, int resultSetHoldability);

The following table contains a list of valid values for resultSetType and
resultSetConcurrency.

<table>
<thead>
<tr>
<th>resultSetType value</th>
<th>resultSetConcurrency value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_FORWARD_ONLY</td>
<td>CONCUR_READ_ONLY</td>
</tr>
<tr>
<td>TYPE_FORWARD_ONLY</td>
<td>CONCUR_UPDATABLE</td>
</tr>
<tr>
<td>TYPE_SCROLL_INSENSITIVE</td>
<td>CONCUR_READ_ONLY</td>
</tr>
<tr>
<td>TYPE_SCROLL_SENSITIVE</td>
<td>CONCUR_READ_ONLY</td>
</tr>
<tr>
<td>TYPE_SCROLL_SENSITIVE</td>
<td>CONCUR_UPDATABLE</td>
</tr>
</tbody>
</table>

Note:
1. This value does not apply to connections to IBM Informix.

resultSetHoldability has two possible values: HOLD_CURSORS_OVER_COMMIT and
CLOSE_CURSORS_AT_COMMIT. Either of these values can be specified with any
valid combination of resultSetConcurrency and resultSetHoldability. The value that
you set overrides the default holdability for the connection.

Restriction: If the ResultSet is scrollable, and the ResultSet is used to select
columns from a table on a DB2 Database for Linux, UNIX, and Windows
server, the SELECT list of the SELECT statement that defines the ResultSet
cannot include columns with the following data types:
• LONG VARCHAR
• LONG VARGRAPHIC
• BLOB
• CLOB
• XML
• A distinct type that is based on any of the previous data types in this list
• A structured type

2. If the SELECT statement has input parameters, invoke setXXX methods to pass
values to the input parameters.
3. Invoke the executeQuery method to obtain the result table from the SELECT
statement in a ResultSet object.
4. For each row that you want to access:
   a. Position the cursor using one of the methods that are listed in the following
table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>first[1]</td>
<td>On the first row of the ResultSet</td>
</tr>
<tr>
<td>last[1]</td>
<td>On the last row of the ResultSet</td>
</tr>
<tr>
<td>next[1]</td>
<td>On the next row of the ResultSet</td>
</tr>
<tr>
<td>previous[1]</td>
<td>On the previous row of the ResultSet</td>
</tr>
<tr>
<td>absolute(int n)[1][3]</td>
<td>If n&gt;0, on row n of the ResultSet. If n&lt;0, and m is the number of rows in the ResultSet, on row m+n+1 of the ResultSet.</td>
</tr>
<tr>
<td>relative(int n)[1][3]</td>
<td>If n&gt;0, on the row that is n rows after the current row. If n&lt;0, on the row that is n rows before the current row. If n=0, on the current row.</td>
</tr>
</tbody>
</table>
Table 11. ResultSet methods for positioning a scrollable cursor (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>afterLast</td>
<td>After the last row in the ResultSet</td>
</tr>
<tr>
<td>beforeFirst</td>
<td>Before the first row in the ResultSet</td>
</tr>
</tbody>
</table>

Notes:
1. This method does not apply to connections to IBM Informix.
2. If the cursor is before the first row of the ResultSet, this method positions the cursor on the first row.
3. If the cursor is after the last row of the ResultSet, this method positions the cursor on the last row.
4. If the absolute value of \( n \) is greater than the number of rows in the result set, this method positions the cursor after the last row if \( n \) is positive, or before the first row if \( n \) is negative.
5. The cursor must be on a valid row of the ResultSet before you can use this method. If the cursor is before the first row or after the last row, the method throws an SQLException.
6. Suppose that \( m \) is the number of rows in the ResultSet and \( x \) is the current row number in the ResultSet. If \( n > 0 \) and \( x + n > m \), the driver positions the cursor after the last row. If \( n < 0 \) and \( x + n < 1 \), the driver positions the cursor before the first row.

b. If you need to know the current cursor position, use the getRow, isFirst, isLast, isFirst, or isAfterLast method to obtain this information.

c. If you specified a resultSetType value of TYPE_SCROLL_SENSITIVE in step 1 on page 51 and you need to see the latest values of the current row, invoke the refreshRow method.

Recommendation: Because refreshing the rows of a ResultSet can have a detrimental effect on the performance of your applications, you should invoke refreshRow only when you need to see the latest data.

d. Perform one or more of the following operations:
   - To retrieve data from each column of the current row of the ResultSet object, use getXXX methods.
   - To update the current row from the underlying table, use updateXXX methods to assign column values to the current row of the ResultSet. Then use updateRow to update the corresponding row of the underlying table. If you decide that you do not want to update the underlying table, invoke the cancelRowUpdates method instead of the updateRow method.
     The resultSetConcurrency value for the ResultSet must be CONCUR_UPDATABLE for you to use these methods.
   - To delete the current row from the underlying table, use the deleteRow method. Invoking deleteRow causes the driver to replace the current row of the ResultSet with a hole.
     The resultSetConcurrency value for the ResultSet must be CONCUR_UPDATABLE for you to use this method.

5. Invoke the close method to close the ResultSet object.
6. Invoke the close method to close the Statement or PreparedStatement object.

The following code demonstrates how to retrieve all rows from the employee table in reverse order, and update the phone number for employee number "000010". The numbers to the right of selected statements correspond to the previously-described steps.
Multi-row SQL operations in JDBC applications:

The IBM Data Server Driver for JDBC and SQLJ supports multi-row INSERT, UPDATE, and FETCH for connections to data sources that support these operations.

**Multi-row INSERT**

In JDBC applications, when you execute INSERT or MERGE statements that use parameter markers in a batch, if the data server supports multi-row INSERT, the IBM Data Server Driver for JDBC and SQLJ can transform the batch INSERT or MERGE operations into multi-row INSERT statements. Multi-row INSERT operations can provide better performance in the following ways:

- For local applications, multi-row INSERTs result in fewer accesses of the data server.
- For distributed applications, multi-row INSERTs result in fewer network operations.

You cannot execute a multi-row INSERT operation by including a multi-row INSERT statement in a statement string in your JDBC application.

Multi-row INSERT is used by default. You can use the Connection or DataSource property enableMultiRowInsertSupport to control whether multi-row INSERT is used. Multi-row INSERT cannot be used for INSERT FROM SELECT statements that are executed in a batch.
**Multi-row FETCH**

Multi-row FETCH can provide better performance than retrieving one row with each FETCH statement. For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, multi-row FETCH can be used for forward-only cursors and scrollable cursors. For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, multi-row FETCH can be used only for scrollable cursors.

When you retrieve data in your applications, the IBM Data Server Driver for JDBC and SQLJ determines whether to use multi-row FETCH, depending on several factors:

- The settings of the enableRowsetSupport and useRowsetCursor properties
- The type of IBM Data Server Driver for JDBC and SQLJ connectivity that is being used
- The version of the IBM Data Server Driver for JDBC and SQLJ

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS, one of the following sets of conditions must be true for multi-row FETCH to be used.

- First set of conditions:
  - The IBM Data Server Driver for JDBC and SQLJ version is 3.51 or later.
  - The enableRowsetSupport property value is `com.ibm.db2.jcc.DB2BaseDataSource.YES (1)`, or the enableRowsetSupport property value is `com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0)` and the useRowsetCursor property value is `com.ibm.db2.jcc.DB2BaseDataSource.YES (1)`.
  - The FETCH operation uses a scrollable cursor.
  
  For forward-only cursors, fetching of multiple rows might occur through DRDA block FETCH. However, this behavior does not utilize the data source's multi-row FETCH capability.

- Second set of conditions:
  - The IBM Data Server Driver for JDBC and SQLJ version is 3.1.
  - The useRowsetCursor property value is `com.ibm.db2.jcc.DB2BaseDataSource.YES (1)`.
  - The FETCH operation uses a scrollable cursor.
  
  For forward-only cursors, fetching of multiple rows might occur through DRDA block FETCH. However, this behavior does not utilize the data source's multi-row FETCH capability.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, the following conditions must be true for multi-row FETCH to be used.

- The IBM Data Server Driver for JDBC and SQLJ version is 3.51 or later.
- The enableRowsetSupport property value is `com.ibm.db2.jcc.DB2BaseDataSource.YES (1)`.
- The FETCH operation uses a scrollable cursor or a forward-only cursor.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, you can control the maximum size of a rowset for each statement by setting the maxRowsetSize property.
Multi-row positioned UPDATE or DELETE

The IBM Data Server Driver for JDBC and SQLJ supports a technique for performing positioned update or delete operations that follows the JDBC 1 standard. That technique involves using the ResultSet.getCursorName method to obtain the name of the cursor for the ResultSet, and defining a positioned UPDATE or positioned DELETE statement of the following form:

```
UPDATE table SET col1=value1,...coln=valueN WHERE CURRENT OF cursorname
DELETE FROM table WHERE CURRENT OF cursorname
```

Multi-row UPDATE or DELETE when useRowsetCursor is set to true: If you use the JDBC 1 technique to update or delete data on a database server that supports multi-row FETCH, and multi-row FETCH is enabled through the useRowsetCursor property, the positioned UPDATE or DELETE statement might update or delete multiple rows, when you expect it to update or delete a single row. To avoid unexpected updates or deletes, you can take one of the following actions:

- Use an updatable ResultSet to retrieve and update one row at a time, as shown in the previous example.
- Set useRowsetCursor to false.

Multi-row UPDATE or DELETE when enableRowsetSupport is set to com.ibm.db2.jcc.DB2BaseDataSource.YES (1): The JDBC 1 technique for updating or deleting data is incompatible with multi-row FETCH that is enabled through the enableRowsetSupport property.

**Recommendation:** If your applications use the JDBC 1 technique, update them to use the JDBC 2.0 ResultSet.updateRow or ResultSet.deleteRow methods for positioned update or delete activity.

Testing whether the current row of a ResultSet is a delete hole or update hole in a JDBC application:

If a ResultSet has the TYPE_SCROLL_SENSITIVE attribute, and the underlying cursor is SENSITIVE STATIC, you need to test for delete holes or update holes before you attempt to retrieve rows of the ResultSet.

After a SENSITIVE STATIC ResultSet is opened, it does not change size. This means that deleted rows are replaced by placeholders, which are also called holes. If updated rows no longer fit the criteria for the ResultSet, those rows also become holes. You cannot retrieve rows that are holes.

To test whether the current row in a ResultSet is a delete hole or update hole, follow these steps:

1. Call the DatabaseMetaData.deletesAreDetected or DatabaseMetaData.updatesAreDetected method with the TYPE_SCROLL_SENSITIVE argument to determine whether the data source creates holes for a TYPE_SCROLL_SENSITIVE ResultSet.
2. If DatabaseMetaData.deletesAreDetected or DatabaseMetaData.updatesAreDetected returns true, which means that the data source can create holes, call the ResultSet.rowDeleted or ResultSet.rowUpdated method to determine whether the current row is a delete or update hole. If the method returns true, the current row is a hole.

The following code tests whether the current row is a delete hole.
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
// Create a Statement object
// for a scrollable, updatable
// ResultSet

ResultSet rs =
stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE FOR UPDATE OF PHONENO");
// Create the ResultSet

DatabaseMetaData dbmd = con.getMetaData();
// Create the DatabaseMetaData object

boolean dbSeesDeletes =
dbmd.deletesAreDetected(ResultSet.TYPESCROLL_SENSITIVE);
// Can the database see delete holes?

rs.afterLast(); // Position the cursor at the end of
// the ResultSet

while (rs.previous()) { // Position the cursor backward
if (dbSeesDeletes) { // If delete holes can be detected
if (!rs.rowDeleted()) // If this row is not a delete hole
{
    s = rs.getString("EMPNO"); // Retrieve the employee number
    System.out.println("Employee number = " + s);
    // Print the column value
}
}
}
rs.close(); // Close the ResultSet
stmt.close(); // Close the Statement

Inserting a row into a ResultSet in a JDBC application:

If a ResultSet has a resultSetConcurrency attribute of CONCUR_UPDATABLE, you
can insert rows into the ResultSet.

To insert a row into a ResultSet, follow these steps:

1. Perform the following steps for each row that you want to insert.
   a. Call the ResultSet.moveToInsertRow method to create the row that you
      want to insert. The row is created in a buffer outside the ResultSet.
      If an insert buffer already exists, all old values are cleared from the buffer.
   b. Call ResultSet.updateXXX methods to assign values to the row that you
      want to insert.
      You need to assign a value to at least one column in the ResultSet. If you do
      not do so, an SQLException is thrown when the row is inserted into the
      ResultSet.
      If you do not assign a value to a column of the ResultSet, when the
      underlying table is updated, the data source inserts the default value for the
      associated table column.
      If you assign a null value to a column that is defined as NOT NULL, the
      JDBC driver throws and SQLException.
   c. Call ResultSet.insertRow to insert the row into the ResultSet.
      After you call ResultSet.insertRow, all values are always cleared from the
      insert buffer, even if ResultSet.insertRow fails.

2. Reposition the cursor within the ResultSet.
To move the cursor from the insert row to the ResultSet, you can invoke any of
the methods that position the cursor at a specific row, such as ResultSet.first,
ResultSet.absolute, or ResultSet.relative. Alternatively, you can call
ResultSet.moveToCurrentRow to move the cursor to the row in the ResultSet
that was the current row before the insert operation occurred.
After you call ResultSet.moveToCurrentRow, all values are cleared from the insert buffer.

Example: The following code illustrates inserting a row into a ResultSet that consists of all rows in the sample DEPARTMENT table. After the row is inserted, the code places the cursor where it was located in the ResultSet before the insert operation. The numbers to the right of selected statements correspond to the previously-described steps.

```java
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                          ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM DEPARTMENT");
rs.moveToInsertRow();
rs.updateString("DEPT_NO", "M13");       // 1a
rs.updateString("DEPTNAME", "TECHNICAL SUPPORT");
rs.updateString("MGRNO", "000010");
rs.updateString("ADMREDEPT", "A00");
rs.insertRow();
rs.moveToCurrentRow();                   // 1b
```

Testing whether the current row was inserted into a ResultSet in a JDBC application:

If a ResultSet is dynamic, you can insert rows into it. After you insert rows into a ResultSet you might need to know which rows were inserted.

To test whether the current row in a ResultSet was inserted, follow these steps:

1. Call the DatabaseMetaData.ownInsertsAreVisible and DatabaseMetaData.othersInsertsAreVisible methods to determine whether inserts can be visible to the given type of ResultSet.
2. If inserts can be visible to the ResultSet, call the DatabaseMetaData.insertsAreDetected method to determine whether the given type of ResultSet can detect inserts.
3. If the ResultSet can detect inserts, call the ResultSet.rowInserted method to determine whether the current row was inserted.

Calling stored procedures in JDBC applications

To call stored procedures, you invoke methods in the CallableStatement class.

The basic steps for calling a stored procedures using standard CallableStatement methods are:

1. Invoke the Connection.prepareCall method with the CALL statement as its argument to create a CallableStatement object.
   
   You can represent parameters with standard parameter markers (?) or named parameter markers. You cannot mix named parameter markers with standard parameter markers in the same CALL statement.

Restriction: The parameter types that are permitted depend on whether the data source supports dynamic execution of the CALL statement. DB2 for z/OS does not support dynamic execution of the CALL statement. For a call to a stored procedure that is on a DB2 for z/OS database server, the parameters can be parameter markers or literals, but not expressions. The following table lists the types of literals that are supported, and the JDBC types to which they map.
2. Invoke the CallableStatement.setXXX methods to pass values to the input parameters (parameters that are defined as IN or INOUT in the CREATE PROCEDURE statement).
   
   This step assumes that you use standard parameter markers or named parameters. Alternatively, if you use named parameter markers, you use IBM Data Server Driver for JDBC and SQLJ-only methods to pass values to the input parameters.

   **Restriction:** If the data source does not support dynamic execution of the CALL statement, you must specify the data types for CALL statement input parameters exactly as they are specified in the stored procedure definition.

3. Invoke the CallableStatement.registerOutParameter method to register parameters that are defined as OUT in the CREATE PROCEDURE statement with specific data types.
   
   This step assumes that you use standard parameter markers. Alternatively, if you use named parameter markers, you use IBM Data Server Driver for JDBC and SQLJ-only methods to register OUT parameters with specific data types.

   **Restriction:** If the data source does not support dynamic execution of the CALL statement, you must specify the data types for CALL statement OUT, IN, or INOUT parameters exactly as they are specified in the stored procedure definition.

4. Invoke one of the following methods to call the stored procedure:

   - **CallableStatement.executeUpdate**
     
     Invoke this method if the stored procedure does not return result sets.

   - **CallableStatement.executeQuery**
     
     Invoke this method if the stored procedure returns one result set.
     
     You can invoke CallableStatement.executeQuery for a stored procedure that returns no result sets if you set property allowNullResultSetForExecuteQuery to DB2BaseDataSource.YES (1). In that case, CallableStatement.executeQuery returns null. This behavior does not conform to the JDBC standard.

   - **CallableStatement.execute**
     
     Invoke this method if the stored procedure returns multiple result sets, or an unknown number of result sets.

     **Restriction:** IBM Informix (IDS) data sources do not support multiple result sets.

5. If the stored procedure returns multiple result sets, retrieve the result sets.

     **Restriction:** IDS data sources do not support multiple result sets.

### Table 12. Supported literal types in parameters in DB2 for z/OS stored procedure calls

<table>
<thead>
<tr>
<th>Literal parameter type</th>
<th>JDBC type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>java.sql.Types.INTEGER</td>
<td>-122, 40022, +27</td>
</tr>
<tr>
<td>Floating-point decimal</td>
<td>java.sql.Types.DOUBLE</td>
<td>23E12, 40022E-4, 2723E+15, 1E+23, 0E0</td>
</tr>
<tr>
<td>Fixed-point decimal</td>
<td>java.sql.Types.DECIMAL</td>
<td>-23.12, 40022.4295, 0.0, 2723.23, 1000000000</td>
</tr>
<tr>
<td>Character</td>
<td>java.sql.Types.VARCHAR</td>
<td>'Grantham Lutz', 'O’Conner', 'ABcde?z?'</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>java.sql.Types.VARBINARY</td>
<td>X'C1C30427', X'00CF18E0'</td>
</tr>
<tr>
<td>Unicode string</td>
<td>java.sql.Types.VARCHAR</td>
<td>UX'0041', UX'0054006500730074'</td>
</tr>
</tbody>
</table>
6. Invoke the CallableStatement.getXXX methods to retrieve values from the OUT parameters or INOUT parameters.

7. Invoke the CallableStatement.close method to close the CallableStatement object when you have finished using that object.

Example: The following code illustrates calling a stored procedure that has one input parameter, four output parameters, and no returned ResultSets. The numbers to the right of selected statements correspond to the previously-described steps.

```java
int ifcaret;
int ifcareas;
int xsbytes;
String errbuff;
Connection con;
CallableStatement cstmt;
ResultSet rs;
...
cstmt = con.prepareStatement("CALL DSN8.DSNBED2(?,?,?,?,?)");  // Create a CallableStatement object
    // Set input parameter (DB2 command)
cstmt.setString(1, "DISPLAY THREAD(*)");
cstmt.registerOutParameter(2, Types.INTEGER);       // Register output parameters
cstmt.registerOutParameter(3, Types.INTEGER);
cstmt.registerOutParameter(4, Types.INTEGER);
cstmt.registerOutParameter(5, Types.VARCHAR);
cstmt.executeUpdate(); // Call the stored procedure
ifcaret = cstmt.getInt(2); // Get the output parameter values
ifcareas = cstmt.getInt(3);
xbytes = cstmt.getInt(4);
errbuff = cstmt.getString(5);
cstmt.close();
```

Using named parameters in CALL statements in JDBC applications

The IBM Data Server Driver for JDBC and SQLJ provides several ways to use named parameters when you call stored procedures. Named parameters use a different syntax from named parameter markers.

You can use named parameters in either or both of the following places in a JDBC application:

- In the CALL statement
  With named parameters, you do not need to specify parameters in the CALL statement in the same order that they appear in the stored procedure definition. In addition, you do not need to specify all parameters in the CALL statement. Unspecified parameters take the default values that are specified in the stored procedure definition.

- In CallableStatement.setXXX, CallableStatement.getXXX, and CallableStatement.registerOutParameter methods
  You can make your programs easier to read by specifying parameter names as they appear in the stored procedure definition, rather than the positions of the parameters in the definition.

To use named parameters with CALL statements, follow these steps.

1. Invoke the Connection.prepareCall method with the CALL statement as its argument to create a CallableStatement object.
   To indicate each parameter, you can use a parameter markers (?), or this syntax:
   ```java
   parameter-name =>?
   ```
parameter-name identifies a parameter in the CREATE PROCEDURE statement. You can explicitly assign the default value or the null value to a named parameter by specifying the DEFAULT keyword or the NULL keyword. For parameters for which a default value is specified in the CREATE PROCEDURE statement, you can implicitly assign the default values to named parameters by omitting those parameters from the CALL statement. You can omit parameters only if all of the omitted parameters have default values in the stored procedure definition.

You cannot mix named parameters and named parameter markers in the same CALL statement.

2. Invoke the CallableStatement.setXXX methods to pass values to the input parameters (parameters that are defined as IN or INOUT in the CREATE PROCEDURE statement).

You can assign values in either of the following ways:
   • By position, using CallableStatement.setXXX(parameterIndex,...)
   • By name, using CallableStatement.setXXX(parameterName,...)

   parameterName is a string that is enclosed in double quotation marks, whose value matches a parameter name in the CREATE PROCEDURE statement.

3. Invoke the CallableStatement.registerOutParameter method to register parameters that are defined as OUT in the CREATE PROCEDURE statement with specific data types.

4. Invoke CallableStatement.executeUpdate, CallableStatement.executeQuery, or CallableStatement.execute to execute the stored procedure.

5. If the stored procedure returns multiple result sets, retrieve those result sets.
   You can register the output parameters in either of the following ways:
   • By position, using CallableStatement.registerOutParameter(parameterIndex,...)
   • By name, using CallableStatement.registerOutParameter(parameterName,...)

   parameterName is a string that is enclosed in double quotation marks, whose value matches a parameter name in the CREATE PROCEDURE statement.

6. Invoke the CallableStatement.getXXX methods to retrieve values from the OUT parameters or INOUT parameters.
   You can retrieve values in either of the following ways:
   • By position, using CallableStatement.getXXX(parameterIndex,...)
   • By name, using CallableStatement.getXXX(parameterName,...)

   parameterName is a string that is enclosed in double quotation marks, whose value matches a parameter name in the CREATE PROCEDURE statement.

7. Invoke the CallableStatement.close method to close the CallableStatement object when you have finished using that object.

The following code illustrates calling a stored procedure that has the following definition:

```
CREATE PROCEDURE SALS (  
    OUT retcode INTEGER,  
    IN lowsal DOUBLE,  
    IN medsal DOUBLE,  
    IN highsal DOUBLE DEFAULT 100000,  
    IN department CHAR(3) DEFAULT '---')  
SPECIFIC JDBC_SALS  
DYNAMIC RESULT SETS 0  
DETERMINISTIC  
LANGUAGE JAVA  
PARAMETER STYLE JAVA  
NO DBINFO
```
The input parameters in the CALL statement are represented by named parameters. The third and fourth parameters are called with the default values for the stored procedure. The numbers to the right of selected statements correspond to the previously-described steps.

```
int hvRetCode; // Host variable for output parameter
Connection con;
CallableStatement cstmt;
ResultSet rs;
...

cstmt = con.prepareCall("CALL SALS(retcode=>?,lowsal=>?,medsal=>?,highsal=>DEFAULT)");  // Prepare the Call statement.
   // Implicitly use the default value for the last parameter
   // by omitting it.

cstmt.setDouble("lowsal", 10000);  // Register output parameter
   cstmt.setDouble("medsal", 50000);
   cstmt.registerOutParameter("retcode", Types.INTEGER);
   cstmt.executeUpdate(); // Call the stored procedure
   hvRetCode = cstmt.getInt("retcode");
   System.out.println("Return code from SALS call: " + hvRetCode);
   cstmt.close();
```

**Retrieving multiple result sets from a stored procedure in a JDBC application**

If you call a stored procedure that returns result sets, you need to include code to retrieve the result sets.

The steps that you take depend on whether you know how many result sets are returned, and whether you know the contents of those result sets.

**Retrieving a known number of result sets from a stored procedure in a JDBC application:**

Retrieving a known number of result sets from a stored procedure is a simpler procedure than retrieving an unknown number of result sets.

To retrieve result sets when you know the number of result sets and their contents, follow these steps:

1. Invoke the Statement.executeUpdate method, the PreparedStatement.executeUpdate method, or the CallableStatement.executeUpdate method to call the stored procedure.

   Use PreparedStatement.executeUpdate if the stored procedure has input parameters.

2. Invoke the getResultSet method to obtain the first result set, which is in a ResultSet object.

3. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.

4. If there are n result sets, repeat the following steps n-1 times:
   a. Invoke the getMoreResults method to close the current result set and point to the next result set.
b. Invoke the getResultSet method to obtain the next result set, which is in a ResultSet object.

c. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.

Example: The following code illustrates retrieving two result sets. The first result set contains an INTEGER column, and the second result set contains a CHAR column. The numbers to the right of selected statements correspond to the previously described steps.

```java
CallableStatement cstmt;
ResultSet rs;
int i;
String s;
...
cstmt.execute(); // Call the stored procedure
rs = cstmt.getResultSet(); // Get the first result set
while (rs.next()) { // Position the cursor
    i = rs.getInt(1); // Retrieve current result set value
    System.out.println("Value from first result set = " + i);
    // Print the value
}
cstmt.getMoreResults(); // Point to the second result set
rs = cstmt.getResultSet(); // and close the first result set
while (rs.next()) { // Get the second result set
    s = rs.getString(1); // Retrieve current result set value
    System.out.println("Value from second result set = " + s);
    // Print the value
}
rs.close(); // Close the result set
stmt.close(); // Close the statement
```

Retrieving an unknown number of result sets from a stored procedure in a JDBC application:

Retrieving an unknown number of result sets from a stored procedure is a more complicated procedure than retrieving a known number of result sets.

To retrieve result sets when you do not know the number of result sets or their contents, you need to retrieve ResultSets, until no more ResultSets are returned. For each ResultSet, use ResultSetMetaData methods to determine its contents.

After you call a stored procedure, follow these basic steps to retrieve the contents of an unknown number of result sets.

1. Check the value that was returned from the execute statement that called the stored procedure.
   - If the returned value is `true`, there is at least one result set, so you need to go to the next step.

2. Repeat the following steps in a loop:
   a. Invoke the getResultSet method to obtain a result set, which is in a ResultSet object. Invoking this method closes the previous result set.
   b. Use ResultSetMetaData methods to determine the contents of the ResultSet, and retrieve data from the ResultSet.
   c. Invoke the getMoreResults method to determine whether there is another result set. If getMoreResults returns `true`, go to step 1 to get the next result set.
**Example:** The following code illustrates retrieving result sets when you do not know the number of result sets or their contents. The numbers to the right of selected statements correspond to the previously described steps.

```java
CallableStatement cst = null;
ResultSet rs = null;
...
boolean resultsAvailable = cst.execute(); // Call the stored procedure
while (resultsAvailable) {
    ResultSet rs = cst.getResultSet(); // Get a result set
    ...
    // Process the ResultSet
    // as you would process
    // a ResultSet from a table

    resultsAvailable = cst.getMoreResults(); // Check for next result set
    // (Also closes the
    // previous result set)
}
```

**Keeping result sets open when retrieving multiple result sets from a stored procedure in a JDBC application:**

The `getMoreResults` method has a form that lets you leave the current ResultSet open when you open the next ResultSet.

To specify whether result sets stay open, follow this process:

When you call `getMoreResults` to check for the next ResultSet, use this form:

```java
CallableStatement.getMoreResults(int current);
```

- To keep the current ResultSet open when you check for the next ResultSet, specify a value of `Statement.KEEP_CURRENT_RESULT` for `current`.
- To close the current ResultSet when you check for the next ResultSet, specify a value of `Statement.CLOSE_CURRENT_RESULT` for `current`.
- To close all ResultSet objects, specify a value of `Statement.CLOSE_ALL_RESULTS` for `current`.

**Example:** The following code keeps all ResultSets open until the final ResultSet has been retrieved, and then closes all ResultSets.

```java
CallableStatement cst = null;
...
boolean resultsAvailable = cst.execute(); // Call the stored procedure
if (resultsAvailable) {
    ResultSet rs1 = cst.getResultSet(); // Get a result set
    ...
    resultsAvailable = cst.getMoreResults(Statement.KEEP_CURRENT_RESULT);
    // Check for next result set
    // but do not close
    // previous result set

    if (resultsAvailable) {
        ResultSet rs2 = cst.getResultSet(); // Get next result set
        ...
        // Process either ResultSet
    }
} resultsAvailable = cst.getMoreResults(Statement.CLOSE_ALL_RESULTS);
// Close the result sets
```

**Learning about stored procedure parameter names using DB2ParameterMetaData methods:**

The `DB2ParameterMetaData.getProcedureParameterName` method lets you retrieve the defined name of a parameter in an SQL CALL statement.
To invoke ParameterMetaData.getProcedureParameterName, you need to perform these basic steps:

1. Invoke the Connection.prepareCall method with the CALL statement as its argument to create a CallableStatement object.
2. Pass values to the input parameters (parameters that are defined as IN or INOUT in the CREATE PROCEDURE statement).
3. Register parameters that are defined as OUT in the CREATE PROCEDURE statement with specific data types.
4. Call the stored procedure.
5. Invoke CallableStatement.getParameterMetaData to retrieve information about the stored procedure parameters.
6. Cast the retrieved ParameterMetaData object as a DB2ParameterMetaData object.
7. Call the DB2ParameterMetaData.getProcedureParameterName method for each CALL statement parameter for which you need to retrieve the parameter name in the CREATE PROCEDURE statement.

The following code demonstrates how to use DB2ParameterMetaData.getProcedureParameterName to determine the names that correspond to standard parameter markers in a stored procedure that is defined like this:

```java
CREATE PROCEDURE SP
  (OUT PARM CHAR(10), IN CHAR(10))
```

The numbers to the right of selected statements correspond to the previously-described steps.

```java
Connection con;
...
CallableStatement cstmt = con.prepareCall("CALL SP(?, ?)"");  
  // Create a CallableStatement object
  // Create a CallableStatement object
    cstmt.setString (2, "INPUT_VALUE");  
    // Set input parameter
    cstmt.registerOutParameter (1, java.sql.Types.CHAR);  
    // Register output parameter
    cstmt.execute();  
    // Call the stored procedure
    DB2ParameterMetaData md =  
      (DB2ParameterMetaData)cstmt.getParameterMetaData ();
    md.getProcedureParameterName(1); // Returns "PARM"
    md.getProcedureParameterName(2); // Returns "2"
```

The following code demonstrates how to use DB2ParameterMetaData.getProcedureParameterName to determine the names that correspond to named parameter markers in a stored procedure that is defined like this:

```java
CREATE PROCEDURE SP
  (OUT PARM CHAR(10), IN CHAR(10))
```

The numbers to the right of selected statements correspond to the previously-described steps.

```java
Connection con;
...
CallableStatement cstmt = con.prepareCall("CALL SP(:output, :input)"");  
  // Create a CallableStatement object
  // Create a CallableStatement object
    ((DB2PreparedStatement)cstmt).setJccStringAtName("input", "INPUT_VALUE");  
    // Set input parameter
    ((DB2CallableStatement)cstmt).registerJccOutParameterAtName  
      ("output", java.sql.Types.CHAR);
```
LOBs in JDBC applications with the IBM Data Server Driver for JDBC and SQLJ

The IBM Data Server Driver for JDBC and SQLJ supports methods for updating and retrieving data from BLOB, CLOB, and DBCLOB columns in a table, and for calling stored procedures or user-defined functions with BLOB or CLOB parameters.

Progressive streaming with the IBM Data Server Driver for JDBC and SQLJ

If the data source supports progressive streaming, also known as dynamic data format, the IBM Data Server Driver for JDBC and SQLJ can use progressive streaming to retrieve data in LOB or XML columns.

DB2 for z/OS Version 9.1 and later supports progressive streaming for LOBs and XML objects. DB2 Database for Linux, UNIX, and Windows Version 9.5 and later, IBM Informix (IDS) Version 11.50 and later, and DB2 for i V6R1 and later support progressive streaming for LOBs.

With progressive streaming, the data source dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects.

Progressive streaming is the default behavior in the following environments:

<table>
<thead>
<tr>
<th>Minimum IBM Data Server Driver for JDBC and SQLJ version</th>
<th>Minimum data server version</th>
<th>Types of objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.53</td>
<td>DB2 for i V6R1</td>
<td>LOB, XML</td>
</tr>
<tr>
<td>3.50</td>
<td>DB2 Database for Linux, UNIX, and Windows Version 9.5</td>
<td>LOB</td>
</tr>
<tr>
<td>3.50</td>
<td>IDS Version 11.50</td>
<td>LOB</td>
</tr>
<tr>
<td>3.2</td>
<td>DB2 for z/OS Version 9</td>
<td>LOB, XML</td>
</tr>
</tbody>
</table>

You set the progressive streaming behavior on new connections using the IBM Data Server Driver for JDBC and SQLJ progressiveStreaming property.

For DB2 for z/OS Version 9.1 and later data sources, or DB2 Database for Linux, UNIX, and Windows Version 9.5 and later data sources, you can set the progressive streaming behavior for existing connections with the `DB2Connection.setDBProgressiveStreaming(DB2BaseDataSource.YES)` method. If you call `DB2Connection.setDBProgressiveStreaming(DB2BaseDataSource.YES)`, all ResultSet objects that are created on the connection use progressive streaming behavior.
When progressive streaming is enabled, you can control when the JDBC driver materializes LOBs with the `streamBufferSize` property. If a LOB or XML object is less than or equal to the `streamBufferSize` value, the object is materialized.

**Important:** With progressive streaming, when you retrieve a LOB or XML value from a ResultSet into an application variable, you can manipulate the contents of that application variable until you move the cursor or close the cursor on the ResultSet. After that, the contents of the application variable are no longer available to you. If you perform any actions on the LOB in the application variable, you receive an SQLException. For example, suppose that progressive streaming is enabled, and you execute statements like this:

```java
... ResultSet rs = stmt.executeQuery("SELECT CLOBCOL FROM MY_TABLE");
rs.next(); // Retrieve the first row of the ResultSet
Clob clobFromRow1 = rs.getClob(1); // Put the CLOB from the first column of
// the first row in an application variable
String substr1Clob = clobFromRow1.getSubString(1,50); // Retrieve the first 50 bytes of the CLOB
rs.next(); // Move the cursor to the next row. // clobFromRow1 is no longer available.
// String substr2Clob = clobFromRow1.getSubString(51,100); // This statement would yield an SQLException
Clob clobFromRow2 = rs.getClob(1); // Put the CLOB from the first column of
// the second row in an application variable
rs.close(); // Close the ResultSet. // clobFromRow2 is also no longer available.
```

After you execute `rs.next()` to position the cursor at the second row of the ResultSet, the CLOB value in `clobFromRow1` is no longer available to you. Similarly, after you execute `rs.close()` to close the ResultSet, the values in `clobFromRow1` and `clobFromRow2` are no longer available.

If you disable progressive streaming, the way in which the IBM Data Server Driver for JDBC and SQLJ handles LOBs depends on the value of the `fullyMaterializeLobData` property.

Use of progressive streaming is the preferred method of LOB or XML data retrieval.

**LOB locators with the IBM Data Server Driver for JDBC and SQLJ**

The IBM Data Server Driver for JDBC and SQLJ can use LOB locators to retrieve data in LOB columns.

To cause JDBC to use LOB locators to retrieve data from LOB columns, you need to set the `fullyMaterializeLobData` property to `false` and set the `progressiveStreaming` property to `NO` (`DB2BaseDataSource.NO` in an application program).

The effect of `fullyMaterializeLobData` depends on whether the data source supports progressive streaming and the value of the `progressiveStreaming` property:

- If the data source does not support progressive locators:

  If the value of `fullyMaterializeLobData` is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If the value is `false`, LOB data is streamed. The driver uses locators internally to retrieve LOB data in chunks on
It is highly recommended that you set this value to \texttt{false} when you retrieve LOBs that contain large amounts of data. The default is \texttt{true}.

- If the data source supports progressive streaming, also known as dynamic data format:
  
  The JDBC driver ignores the value of \texttt{fullyMaterializeLobData} if the \texttt{progressiveStreaming} property is set to \texttt{YES} (\texttt{DB2BaseDataSource.YES} in an application program) or is not set.

\texttt{fullyMaterializeLobData} has no effect on stored procedure parameters.

As in any other language, a LOB locator in a Java application is associated with only one data source. You cannot use a single LOB locator to move data between two different data sources. To move LOB data between two data sources, you need to materialize the LOB data when you retrieve it from a table in the first data source and then insert that data into the table in the second data source.

**LOB operations with the IBM Data Server Driver for JDBC and SQLJ**

The IBM Data Server Driver for JDBC and SQLJ supports methods for updating and retrieving data from BLOB, CLOB, and DBCLOB columns in a table, and for calling stored procedures or user-defined functions with BLOB or CLOB parameters.

Among the operations that you can perform on LOB data under the IBM Data Server Driver for JDBC and SQLJ are:

- Specify a BLOB or column as an argument of the following ResultSet methods to retrieve data from a BLOB or CLOB column:
  
  For BLOB columns:
  - \texttt{getBinaryStream}
  - \texttt{getBlob}
  - \texttt{getBytes}
  
  For CLOB columns:
  - \texttt{getAsciiStream}
  - \texttt{getCharacterStream}
  - \texttt{getClob}
  - \texttt{getString}

- Call the following ResultSet methods to update a BLOB or CLOB column in an updatable ResultSet:
  
  For BLOB columns:
  - \texttt{updateBinaryStream}
  - \texttt{updateBlob}
  
  For CLOB columns:
  - \texttt{updateAsciiStream}
  - \texttt{updateCharacterStream}
  - \texttt{updateClob}

  If you specify \texttt{-1} for the \texttt{length} parameter in any of the previously listed methods, the IBM Data Server Driver for JDBC and SQLJ reads the input data until it is exhausted.

- Use the following PreparedStatement methods to set the values for parameters that correspond to BLOB or CLOB columns:
  
  For BLOB columns:
  - \texttt{setBytes}
  - \texttt{setBlob}
– setBinaryStream
– setObject, where the Object parameter value is an InputStream.

For CLOB columns:
– setString
– setAsciiStream
– setClob
– setCharacterStream
– setObject, where the Object parameter value is a Reader.

If you specify -1 for length, the IBM Data Server Driver for JDBC and SQLJ reads
the input data until it is exhausted.

• Retrieve the value of a JDBC CLOB parameter using the
CallableStatement.getString method.

Restriction: With IBM Data Server Driver for JDBC and SQLJ type 2 connectivity,
you cannot call a stored procedure that has DBCLOB OUT or INOUT parameters.

If you are using the IBM Data Server Driver for JDBC and SQLJ version 4.0 or
later, you can perform the following additional operations:
• Use ResultSet.updateXXX or PreparedStatement.setXXX methods to update a
BLOB or CLOB with a length value of up to 2GB for a BLOB or CLOB. For
example, these methods are defined for BLOBs:
  ResultSet.updateBlob(int columnIndex, InputStream x, long length)
  ResultSet.updateBlob(String columnLabel, InputStream x, long length)
  ResultSet.updateBinaryStream(int columnIndex, InputStream x, long length)
  ResultSet.updateBinaryStream(String columnLabel, InputStream x, long length)
  PreparedStatement.setBlob(int columnIndex, InputStream x, long length)
  PreparedStatement.setBlob(String columnLabel, InputStream x, long length)
  PreparedStatement.setBinaryStream(int columnIndex, InputStream x, long length)
  PreparedStatement.setBinaryStream(String columnLabel, InputStream x, long length)

• Use ResultSet.updateXXX or PreparedStatement.setXXX methods without the
length parameter when you update a BLOB or CLOB, to cause the IBM Data
Server Driver for JDBC and SQLJ to read the input data until it is exhausted. For
example:
  ResultSet.updateBlob(int columnIndex, InputStream x)
  ResultSet.updateBlob(String columnLabel, InputStream x)
  ResultSet.updateBinaryStream(int columnIndex, InputStream x)
  ResultSet.updateBinaryStream(String columnLabel, InputStream x)
  PreparedStatement.setBlob(int columnIndex, InputStream x)
  PreparedStatement.setBlob(String columnLabel, InputStream x)
  PreparedStatement.setBinaryStream(int columnIndex, InputStream x)
  PreparedStatement.setBinaryStream(String columnLabel, InputStream x)

• Create a Blob or Clob object that contains no data, using the
Connection.createBlob or Connection.createClob method.

• Materialize a Blob or Clob object on the client, when progressive streaming or
locators are in use, using the Blob.getBinaryStream or Clob.getCharacterStream
method.

• Free the resources that a Blob or Clob object holds, using the Blob.free or
Clob.free method.

Java data types for retrieving or updating LOB column data in
JDBC applications

When the JDBC driver cannot immediately determine the data type of a parameter
that is used with a LOB column, you need to choose a parameter data type that is
compatible with the LOB data type.
When the deferPrepares property is set to true, and the IBM Data Server Driver for JDBC and SQLJ processes a PreparedStatement.setXXX call, the driver might need to do extra processing to determine data types. This extra processing can impact performance.

**Input parameters for BLOB columns**

For IN parameters for BLOB columns, or INOUT parameters that are used for input to BLOB columns, you can use one of the following techniques:

- Use a java.sql.Blob input variable, which is an exact match for a BLOB column:
  ```java
  cstmt.setBlob(parmIndex, blobData);
  ```
- Use a CallableStatement.setObject call that specifies that the target data type is BLOB:
  ```java
  byte[] byteData = {(byte)0x1a, (byte)0x2b, (byte)0x3c};
  cstmt.setObject(parmInd, byteData, java.sql.Types.BLOB);
  ```
- Use an input parameter of type java.io.ByteArrayInputStream with a CallableStatement.setBinaryStream call. A java.io.ByteArrayInputStream object is compatible with a BLOB data type. For this call, you need to specify the exact length of the input data:
  ```java
  java.io.ByteArrayInputStream byteStream =
      new java.io.ByteArrayInputStream(byteData);
  int numBytes = byteData.length;
  cstmt.setBinaryStream(parmIndex, byteStream, numBytes);
  ```

**Output parameters for BLOB columns**

For OUT parameters for BLOB columns, or INOUT parameters that are used for output from BLOB columns, you can use the following technique:

- Use the CallableStatement.registerOutParameter call to specify that an output parameter is of type BLOB. Then you can retrieve the parameter value into any variable that has a data type that is compatible with a BLOB data type. For example, the following code lets you retrieve a BLOB value into a byte[] variable:
  ```java
  cstmt.registerOutParameter(parmIndex, java.sql.Types.BLOB);
  cstmt.execute();
  byte[] byteData = cstmt.getBytes(parmIndex);
  ```

**Input parameters for CLOB columns**

For IN parameters for CLOB columns, or INOUT parameters that are used for input to CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob input variable, which is an exact match for a CLOB column:
  ```java
  cstmt.setClob(parmIndex, clobData);
  ```
- Use a CallableStatement.setObject call that specifies that the target data type is CLOB:
  ```java
  String charData = "CharacterString";
  cstmt.setObject(parmInd, charData, java.sql.Types.CLOB);
  ```
- Use one of the following types of stream input parameters:
  - A java.io.StringReader input parameter with a cstmt.setCharacterStream call:
    ```java
    java.io.StringReader reader = new java.io.StringReader(charData);
    cstmt.setCharacterStream(parmIndex, reader, charData.length);
    ```
  - A java.io.ByteArrayInputStream parameter with a cstmt.setAsciiStream call, for ASCII data:
Rowid data is a unique value that uniquely identifies a row in a table. A rowid usually contains characteristics of the row such as its creation date and time, and the rowid data type can contain TIMESTAMP data type. It is important to note that rowid data is not unique across different database tables.

ROWIDs in JDBC with the IBM Data Server Driver for JDBC and SQLJ

DB2 for z/OS and DB2 for i support the ROWID data type for a column in a database table. A ROWID is a value that uniquely identifies a row in a table.

Although IBM Informix (IDS) also supports rowids, those rowids have the INTEGER data type. You can select an IDS rowid column into a variable with a four-byte integer data type.

You can use the following ResultSet methods to retrieve data from a ROWID column:
• getRowId (JDBC 4.0 and later)
• getBytes
• getObject
You can use the following ResultSet method to update a ROWID column of an updatable ResultSet:

- `updateRowId (JDBC 4.0 and later)`
  
  *updateRowId is valid only if the target database system supports updating of ROWID columns.*

If you are using JDBC 3.0, for `getObject`, the IBM Data Server Driver for JDBC and SQLJ returns an instance of the IBM Data Server Driver for JDBC and SQLJ-only class `com.ibm.db2.jcc.DB2RowID`.

If you are using JDBC 4.0, for `getObject`, the IBM Data Server Driver for JDBC and SQLJ returns an instance of the class `java.sql.RowId`.

You can use the following PreparedStatement methods to set a value for a parameter that is associated with a ROWID column:

- `setRowId (JDBC 4.0 and later)`
- `setBytes`
- `setObject`

If you are using JDBC 3.0, for `setObject`, use the IBM Data Server Driver for JDBC and SQLJ-only type `com.ibm.db2.jcc.Types.ROWID` or an instance of the `com.ibm.db2.jcc.DB2RowID` class as the target type for the parameter.

If you are using JDBC 4.0, for `setObject`, use the type `java.sql.Types.ROWID` or an instance of the `java.sql.RowId` class as the target type for the parameter.

You can use the following CallableStatement methods to retrieve a ROWID column as an output parameter from a stored procedure call:

- `getRowId (JDBC 4.0 and later)`
- `getObject`

To call a stored procedure that is defined with a ROWID output parameter, register that parameter to be of the `java.sql.Types.ROWID` type.

ROWID values are valid for different periods of time, depending on the data source on which those ROWID values are defined. Use the `DatabaseMetaData.getRowIdLifetime` method to determine the time period for which a ROWID value is valid. The values that are returned for the data sources are listed in the following table.

<table>
<thead>
<tr>
<th>Database server</th>
<th><code>DatabaseMetaData.getRowIdLifetime</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 for z/OS</td>
<td><code>ROWID_VALID_TRANSACTION</code></td>
</tr>
<tr>
<td>DB2 Database for Linux, UNIX, and Windows</td>
<td><code>ROWID_UNSUPPORTED</code></td>
</tr>
<tr>
<td>DB2 for i</td>
<td><code>ROWID_VALID_FOREVER</code></td>
</tr>
<tr>
<td>IDS</td>
<td><code>ROWID_VALID_FOREVER</code></td>
</tr>
</tbody>
</table>

*Example: Using PreparedStatement.setRowId with a java.sql.RowId target type:* Suppose that `rid` is a RowId object. To set parameter 1, use this form of the `setRowId` method:

```java
ps.setRowId(1, rid);
```
Distinct types in JDBC applications

A distinct type is a user-defined data type that is internally represented as a built-in SQL data type. You create a distinct type by executing the SQL statement CREATE DISTINCT TYPE.

In a JDBC program, you can create a distinct type using the executeUpdate method to execute the CREATE DISTINCT TYPE statement. You can also use executeUpdate to create a table that includes a column of that type. When you retrieve data from a column of that type, or update a column of that type, you use Java identifiers with data types that correspond to the built-in types on which the distinct types are based.

The following example creates a distinct type that is based on an INTEGER type, creates a table with a column of that type, inserts a row into the table, and retrieves the row from the table:

```java
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
stmt = con.createStatement(); // Create a Statement object
stmt.executeUpdate("CREATE DISTINCT TYPE SHOESIZE AS INTEGER"); // Create distinct type
stmt.executeUpdate("CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)"); // Create table with distinct type
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
    "VALUES ('000010', 6)"); // Insert a row
rs = stmt.executeQuery("SELECT EMPNO, EMP_SHOE_SIZE FROM EMP_SHOE"); // Create ResultSet for query
while (rs.next()) {
    empNumVar = rs.getString(1); // Get employee number
    shoeSizeVar = rs.getInt(2); // Get shoe size (use int because underlying type of SHOESIZE is INTEGER)
    System.out.println("Employee number = " + empNumVar +
        " Shoe size = " + shoeSizeVar);
}
rs.close(); // Close ResultSet
stmt.close(); // Close Statement
```

Figure 16. Creating and using a distinct type
Invocation of stored procedures with ARRAY parameters in JDBC applications

JDBC applications that run under the IBM Data Server Driver for JDBC and SQLJ can call stored procedures that have ARRAY parameters. ARRAY parameters are supported in stored procedures on DB2 Database for Linux, UNIX, and Windows Version 9.5 and later.

You can use java.sql.Array objects as IN, OUT, or INOUT parameters in a stored procedure.

For IN or INOUT parameters, use the DB2Connection.createArrayOf method (JDBC 3.0 or earlier) or the Connection.createArrayOf method (JDBC 4.0 or later) to create a java.sql.Array object. Use the CallableStatement.setArray method or the CallableStatement.setObject method to assign a java.sql.Array object to an ARRAY stored procedure parameter.

You can register an OUT ARRAY parameter for a stored procedure call by specifying java.sql.Types.ARRAY as the parameter type in a CallableStatement.registerOutParameter call.

There are two ways to retrieve data from an ARRAY output parameter:

- Use the CallableStatement.getArray method to retrieve the data into a java.sql.Array object, and use the java.sql.Array.getArray method to retrieve the contents of the java.sql.Array object into a Java array.
- Use the CallableStatement.getArray method to retrieve the data into a java.sql.Array object. Use the java.sql.Array.getResultSet() method to retrieve the data into a ResultSet object. Use ResultSet methods to retrieve elements of the array. Each row of the ResultSet contains two columns:
  - An index into the array, which starts at 1
  - The array element

Example: Suppose that input and output parameters IN_PHONE and OUT_PHONE in stored procedure GET_EMP_DATA are arrays that are defined like this:

```
CREATE TYPE PHONENUMBERS AS VARCHAR(10) ARRAY[5]
```

Call GET_EMP_DATA with the two parameters.

```
Connection con;
CallableStatement cstmt;
ResultSet rs;
java.sql.Array inPhoneData;
...
stmt = con.prepareCall("CALL GET_EMP_DATA(?,?)");  // Create a CallableStatement object
cstmt.setObject (1, inPhoneData); // Set input parameter
cstmt.registerOutParameter (2, java.sql.Types.ARRAY);  // Register out parameters
cstmt.executeUpdate();  // Call the stored procedure
Array outPhoneData = cstmt.getArray(2);  // Get the output parameter array
System.out.println("Parameter values from GET_EMP_DATA call: ");
String [] outPhoneNums = (String [])outPhoneData.getArray();  // Retrieve output data from the JDBC Array object
// into a Java String array
```
for(int i=0; i<outPhoneNums.length; i++) {
    System.out.print(outPhoneNums[i]);
    System.out.println();
}

**Savepoints in JDBC applications**

An SQL savepoint represents the state of data and schemas at a particular point in time within a unit of work. You can use SQL statements to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents.

The IBM Data Server Driver for JDBC and SQLJ supports the following methods for using savepoints:

- **Connection.setSavepoint() or Connection.setSavepoint(String name)**
  Sets a savepoint. These methods return a Savepoint object that is used in later releaseSavepoint or rollback operations.

  When you execute either of these methods, DB2 executes the form of the SAVEPOINT statement that includes ON ROLLBACK RETAIN CURSORS.

- **Connection.releaseSavepoint(Savepoint savepoint)**
  Releases the specified savepoint, and all subsequently established savepoints.

- **Connection.rollback(Savepoint savepoint)**
  Rolls back work to the specified savepoint.

- **DatabaseMetaData.supportsSavepoints()**
  Indicates whether a data source supports savepoints.

You can indicate whether savepoints are unique by calling the method DB2Connection.setSavePointUniqueOption. If you call this method with a value of true, the application cannot set more than one savepoint with the same name within the same unit of recovery. If you call this method with a value of false (the default), multiple savepoints with the same name can be created within the same unit of recovery, but creation of a savepoint destroys a previously created savepoint with the same name.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.
Retrieval of automatically generated keys in JDBC applications

With the IBM Data Server Driver for JDBC and SQLJ, you can retrieve automatically generated keys (also called auto-generated keys) from a table using JDBC 3.0 methods.

An automatically generated key is any value that is generated by the data server, instead of being specified by the user. One type of automatically generated key is the contents of an identity column. An identity column is a table column that provides a way for the data source to automatically generate a numeric value for each row. You define an identity column in a CREATE TABLE or ALTER TABLE statement by specifying the AS IDENTITY clause when you define a column that has an exact numeric type with a scale of 0 (SMALLINT, INTEGER, BIGINT, DECIMAL with a scale of zero, or a distinct type based on one of these types).

For connections to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows, the IBM Data Server Driver for JDBC and SQLJ supports the return of automatically generated keys for INSERT statements, for searched UPDATE or searched DELETE statements, or for MERGE statements. For UPDATE, DELETE, or MERGE statements, you can identify any columns as automatically generated keys, even if they are not generated by the data server. In this case, the column values that are returned are the column values for the rows that are modified by the UPDATE, DELETE, or MERGE statement.

Restriction: If the Connection or DataSource property atomicMultiRowInsert is set to DB2BaseDataSource.YES (1), you cannot prepare an SQL statement for retrieval of automatically generated keys and use the PreparedStatement object for batch updates. The IBM Data Server Driver for JDBC and SQLJ version 3.50 or later

Figure 17. Setting, rolling back to, and releasing a savepoint in a JDBC application
throws an SQLException when you call the addBatch or executeBatch method on a PreparedStatement object that is prepared to return automatically generated keys.

Retrieving auto-generated keys for an INSERT statement

With the IBM Data Server Driver for JDBC and SQLJ, you can use JDBC 3.0 methods to retrieve the keys that are automatically generated when you execute an INSERT statement.

About this task

To retrieve automatically generated keys that are generated by an INSERT statement, you need to perform these steps.

1. Use one of the following methods to indicate that you want to return automatically generated keys:
   - If you plan to use the PreparedStatement.executeUpdate method to insert rows, invoke one of these forms of the Connection.prepareStatement method to create a PreparedStatement object:
     The following form is valid for a table on any data source that supports identity columns. sql-statement must be a single-row INSERT statement.

     ```java
     Connection.prepareStatement(sql-statement,
                                  Statement.RETURN_GENERATED_KEYS);
     ```

     The following forms are valid only if the data source supports INSERT within SELECT statements. sql-statement can be a single-row INSERT statement or a multiple-row INSERT statement. With the first form, you specify the names of the columns for which you want automatically generated keys. With the second form, you specify the positions in the table of the columns for which you want automatically generated keys.

     ```java
     Connection.prepareStatement(sql-statement, String [] columnNames);
     Connection.prepareStatement(sql-statement, int [] columnIndexes);
     ```

     - If you use the Statement.executeUpdate method to insert rows, invoke one of these forms of the Statement.executeUpdate method:

     The following form is valid for a table on any data source that supports identity columns. sql-statement must be a single-row INSERT statement.

     ```java
     Statement.executeUpdate(sql-statement, Statement.RETURN_GENERATED_KEYS);
     ```

     The following forms are valid only if the data source supports INSERT within SELECT statements. sql-statement can be a single-row INSERT statement or a multiple-row INSERT statement. With the first form, you specify the names of the columns for which you want automatically generated keys. With the second form, you specify the positions in the table of the columns for which you want automatically generated keys.

     ```java
     Statement.executeUpdate(sql-statement, String [] columnNames);
     Statement.executeUpdate(sql-statement, int [] columnIndexes);
     ```

2. Invoke the PreparedStatement.getGeneratedKeys method or the Statement.getGeneratedKeys method to retrieve a ResultSet object that contains the automatically generated key values.
If you include the Statement.RETURN_GENERATED_KEYS parameter, the data type of the automatically generated keys in the ResultSet is DECIMAL, regardless of the data type of the corresponding column.

The following code creates a table with an identity column, inserts a row into the table, and retrieves the automatically generated key value for the identity column. The numbers to the right of selected statements correspond to the previously described steps.

```java
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
java.math.BigDecimal iDColVar;
...
stmt = con.createStatement(); // Create a Statement object
stmt.executeUpdate("CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
                     "IDENTCOL INTEGER GENERATED ALWAYS AS IDENTITY)" );
// Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) " +
                   "VALUES ('000010', '5555')", Statement.RETURN_GENERATED_KEYS);
// Indicate you want automatically generated keys
rs = stmt.getGeneratedKeys(); // Retrieve the automatically generated key value in a ResultSet.
// Only one row is returned.
while (rs.next()) {
    java.math.BigDecimal idColVar = rs.getBigDecimal(1);
    // Get automatically generated key value
    System.out.println("automatically generated key value = " + idColVar);
}
rs.close(); // Close ResultSet
stmt.close(); // Close Statement
```

The following code creates a table with an identity column, inserts two rows into the table using a multiple-row INSERT statement, and retrieves the automatically generated key values for the identity column. The numbers to the right of selected statements correspond to the previously-described steps.

```java
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();
stmt.executeUpdate("CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
                     "IDENTCOL INTEGER GENERATED ALWAYS AS IDENTITY)" );
// Create table with identity column
stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) " +
                   "VALUES ('000010', '5555'), ('000020', '5556')", id_col);
// Indicate you want automatically generated keys
rs = stmt.getGeneratedKeys(); // Retrieve the automatically generated key value in a ResultSet.
// Only one row is returned.
while (rs.next()) {
    java.math.BigDecimal idColVar = rs.getBigDecimal(1);
    // Get automatically generated key value
    System.out.println("automatically generated key value = " + idColVar);
}
rs.close(); // Close ResultSet
stmt.close(); // Close Statement
```

The following code creates a table with an identity column, inserts two rows into the table using a multiple-row INSERT statement, and retrieves the automatically generated key values for the identity column. The numbers to the right of selected statements correspond to the previously-described steps.

```java
import java.sql.*;
import java.math.*;
import com.ibm.db2.jcc.*;

Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();
stmt.executeUpdate("CREATE TABLE EMP_PHONE (EMPNO CHAR(6), PHONENO CHAR(4), " +
                     "IDENTCOL INTEGER GENERATED ALWAYS AS IDENTITY)" );
// Create table with identity column
String[] id_col = {"IDENTCOL");
int updateCount = stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO) " +
                                     "VALUES ('000010', '5555'), ('000020', '5556'), id_col);
// Indicate you want automatically generated keys
```

78 Developing Java Applications
rs = stmt.getGeneratedKeys(); // generated keys
// Retrieve the automatically generated key values in a ResultSet.
// Two rows are returned.
// Create ResultSet for query
while (rs.next()) {
    int idColVar = rs.getInt(1); // Get automatically generated key values
    System.out.println("automatically generated key value = " + idColVar);
}
stmt.close();
con.close();

Retrieving auto-generated keys for an UPDATE, DELETE, or MERGE statement
With the IBM Data Server Driver for JDBC and SQLJ, you can use JDBC 3.0 methods to retrieve the keys that are automatically generated when you execute a searched UPDATE, searched DELETE, or MERGE statement.

About this task
To retrieve automatically generated keys that are generated by an UPDATE, DELETE, or MERGE statement, you need to perform these steps.
1. Construct a String array that contains the names of the columns from which you want to return automatically generated keys.
   The array must be an array of column names, and not column indexes.
2. Set the autocommit mode for the connection to false.
3. Use one of the following methods to indicate that you want to return automatically generated keys:
   • If you plan to use the PreparedStatement.executeUpdate method to update, delete, or merge rows, invoke this form of the Connection.prepareStatement method to create a PreparedStatement object:
     Connection.prepareStatement(sql-statement, String [] columnNames);
   • If you use the Statement.executeUpdate method to update, delete, or merge rows, invoke this form of the Statement.executeUpdate method:
     Statement.executeUpdate(sql-statement, String [] columnNames);
4. Invoke the PreparedStatement.getGeneratedKeys method or the Statement.getGeneratedKeys method to retrieve a ResultSet object that contains the automatically generated key values.

Suppose that a table is defined like this and has thirty rows:

```sql
CREATE TABLE EMP_BONUS
(EMPNO CHAR(6),
 BONUS DECIMAL(9,2))
```

The following code names the EMPNO column as an automatically generated key, updates the thirty rows in the EMP_BONUS table, and retrieves the values of EMPNO for the updated rows. The numbers to the right of selected statements correspond to the previously described steps.

```java
import java.sql.*;
...
Connection conn;
...
String[] agkNames = {"EMPNO"};  // 1
int updateCount = 0;
conn.setAutoCommit(false);  // 2
```
PreparedStatement ps =
    conn.prepareStatement("UPDATE EMP_BONUS SET BONUS = " +
    " BONUS + 300.00",agkNames);
updateCount = ps.executeUpdate();
ResultSet rs = ps.getGeneratedKeys();
while (rs.next()) {
    String agkEmpNo = rs.getString(1);
    // Get automatically generated key value
    System.out.println("Automatically generated key value = " + agkEmpNo);
}
ps.close();
conn.close();

Using named parameter markers in JDBC applications

You can use named parameter markers instead of standard parameter markers in
PreparedStatement and CallableStatement objects to assign values to the input
parameter markers. You can also use named parameter markers instead of
standard parameter markers in CallableStatement objects to register OUT
parameters that have named parameter markers.

SQL strings that contain the following SQL elements can include named parameter
markers:
• CALL
• DELETE
• INSERT
• MERGE
• PL/SQL block
• SELECT
• SET
• UPDATE

Named parameter markers make your JDBC applications more readable. If you
have named parameter markers in an application, set the IBM Data Server Driver
for JDBC and SQLJ Connection or DataSource property
enableNamedParameterMarkers to DB2BaseDataSource.YES (1) to direct the driver
to accept named parameter markers and send them to the data source as standard
parameter markers.

Using named parameter markers with PreparedStatement objects

You can use named parameter markers instead of standard parameter markers in
PreparedStatement objects to assign values to the parameter markers.

To ensure that applications with named parameters work correctly, regardless of
the data server type and version, before you use named parameter markers in your
applications, set the Connection or DataSource property
enableNamedParameterMarkers to DB2BaseDataSource.YES.

To use named parameter markers with PreparedStatement objects, follow these
steps.
1. Execute the Connection.prepareStatement method on an SQL statement string
    that contains named parameter markers. The named parameter markers must
    follow the rules for SQL host variable names.
    You cannot mix named parameter markers and standard parameter markers in
    the same SQL statement string.
    Named parameter markers are case-insensitive.
2. For each named parameter marker, use a DB2PreparedStatement.setJccXXXAtName method to assign a value to each named input parameter.

If you use the same named parameter marker more than once in the same SQL statement string, you need to call a setJccXXXAtName method for that parameter marker only once.

**Recommendation:** Do not use the same named parameter marker more than once in the same SQL statement string if the input to that parameter marker is a stream. Doing so can cause unexpected results.

**Restriction:** You cannot use standard JDBC PreparedStatement.setXXX methods with named parameter markers. Doing so causes an exception to be thrown.

3. Execute the PreparedStatement.

The following code uses named parameter markers to update the phone number to '4657' for the employee with employee number '000010'. The numbers to the right of selected statements correspond to the previously described steps.

```java
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement("UPDATE EMPLOYEE SET PHONENO=:phonenum WHERE EMPNO=:empnum"); // Create a PreparedStatement object
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName("phonenum", "4567"); // Assign a value to phonenum parameter
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName("empnum", "000010"); // Assign a value to empnum parameter
numUpd = pstmt.executeUpdate(); // Perform the update
pstmt.close(); // Close the PreparedStatement object
```

The following code uses named parameter markers to update values in a PL/SQL block. The numbers to the right of selected statements correspond to the previously described steps.

```java
Connection con;
PreparedStatement pstmt;
int numUpd;
...
String sql = "BEGIN "+
  " UPDATE EMPLOYEE SET PHONENO = :phonenum WHERE EMPNO = :empnum; "+
  "END;";
pstmt = con.prepareStatement(sql); // Create a PreparedStatement object
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName("phonenum", "4567"); // Assign a value to phonenum parameter
((com.ibm.db2.jcc.DB2PreparedStatement)pstmt).setJccStringAtName("empnum", "000010"); // Assign a value to empnum parameter
numUpd = pstmt.executeUpdate(); // Perform the update
pstmt.close(); // Close the PreparedStatement object
```

**Using named parameter markers with CallableStatement objects**

You can use named parameter markers instead of standard parameter markers in CallableStatement objects to assign values to IN or INOUT parameters and to register OUT parameters.
To ensure that applications with named parameters work correctly, regardless of the data server type and version, before you use named parameter markers in your applications, set the Connection or DataSource property enableNamedParameterMarkers to DB2BaseDataSource.YES.

To use named parameter markers with CallableStatement objects, follow these steps.

1. Execute the Connection.prepareCall method on an SQL statement string that contains named parameter markers.
   The named parameter markers must follow the rules for SQL host variable names.
   You cannot mix named parameter markers and standard parameter markers in the same SQL statement string.
   Named parameter markers are case-insensitive.

2. If you do not know the names of the named parameter markers in the CALL statement, or the mode of the parameters (IN, OUT, or INOUT):
   a. Call the CallableStatement.getParameterMetaData method to obtain a ParameterMetaData object with information about the parameters.
   b. Call the ParameterMetaData.getParameterMode method to retrieve the parameter mode.
   c. Cast the ParameterMetaData object to a DB2ParameterMetaData object.
   d. Call the DB2ParameterMetaData.getParameterMarkerNames method to retrieve the parameter names.

3. For each named parameter marker that represents an OUT parameter, use a DB2CallableStatement.registerJccOutParameterAtName method to register the OUT parameter with a data type.
   If you use the same named parameter marker more than once in the same SQL statement string, you need to call a registerJccOutParameterAtName method for that parameter marker only once. All parameters with the same name are registered as the same data type.

   **Restriction:** You cannot use standard JDBC CallableStatement.registerOutParameter methods with named parameter markers. Doing so causes an exception to be thrown.

4. For each named parameter marker for an input parameter, use a DB2CallableStatement.setJccXXXAtName method to assign a value to each named input parameter.
   setJccXXXAtName methods are inherited from DB2PreparedStatement.
   If you use the same named parameter marker more than once in the same SQL statement string, you need to call a setJccXXXAtName method for that parameter marker only once.

   **Recommendation:** Do not use the same named parameter marker more than once in the same SQL statement string if the input to that parameter marker is a stream. Doing so can cause unexpected results.

5. Execute the CallableStatement.

6. Call CallableStatement.getXXX methods or DB2CallableStatement.getJccXXXAtName methods to retrieve output parameter values.

The following code illustrates calling a stored procedure that has one input VARCHAR parameter and one output INTEGER parameter, which are represented
by named parameter markers. The numbers to the right of selected statements correspond to the previously described steps.

```java
CallableStatement cstmt =
    con.prepareCall("CALL MYSP(:inparm,:outparm)");
    // Create a CallableStatement object
(c.com.ibm.db2.jcc.DB2CallableStatement)cstmt).
    registerJccOutParameterAtName("outparm", java.sql.Types.INTEGER);
    // Register OUT parameter data type
(c.com.ibm.db2.jcc.DB2CallableStatement)cstmt).setJccStringAtName("inparm", "4567");
    // Assign a value to inparm parameter
```

```java
cstmt.executeUpdate(); // Call the stored procedure
int outssid = cstmt.getInt(2); // Get the output parameter value
```

Providing extended client information to the data source with IBM Data Server Driver for JDBC and SQLJ-only methods

A set of IBM Data Server Driver for JDBC and SQLJ-only methods provide extra information about the client to the server. This information can be used for accounting, workload management, or debugging.

Extended client information is sent to the database server when the application performs an action that accesses the server, such as executing SQL.

In the IBM Data Server Driver for JDBC and SQLJ version 4.0 or later, the IBM Data Server Driver for JDBC and SQLJ-only methods are deprecated. You should use java.sql.Connection.setClientInfo instead.

The IBM Data Server Driver for JDBC and SQLJ-only methods are listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Information provided</th>
</tr>
</thead>
<tbody>
<tr>
<td>setDB2ClientAccountingInformation</td>
<td>Accounting information</td>
</tr>
<tr>
<td>setDB2ClientApplicationInformation</td>
<td>Name of the application that is working with a connection</td>
</tr>
<tr>
<td>setDB2ClientDebugInfo</td>
<td>The CLIENT DEBUGINFO connection attribute for the Unified debugger</td>
</tr>
<tr>
<td>setDB2ClientProgramId</td>
<td>A caller-specified string that helps the caller identify which program is associated with a particular SQL statement. setDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.</td>
</tr>
<tr>
<td>setDB2ClientUser</td>
<td>User name for a connection</td>
</tr>
<tr>
<td>setDB2ClientWorkstation</td>
<td>Client workstation name for a connection</td>
</tr>
</tbody>
</table>

To set the extended client information, follow these steps:

1. Create a Connection.
2. Cast the java.sql.Connection object to a com.ibm.db2.jcc.DB2Connection.
3. Call any of the methods shown in Table 14.
4. Execute an SQL statement to cause the information to be sent to the DB2 server.
The following code performs the previous steps to pass a user name and a workstation name to the DB2 server. The numbers to the right of selected statements correspond to the previously-described steps.

```java
public class ClientInfoTest {
    public static void main(String[] args) {
        String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            String user = "db2adm";
            String password = "db2adm";
            Connection conn = DriverManager.getConnection(url, user, password);
            if (conn instanceof DB2Connection) {
                DB2Connection db2conn = (DB2Connection) conn;
                db2conn.setDB2ClientUser("Michael L Thompson");
                db2conn.setDB2ClientWorkstation("sjwkstn1");
                // Execute SQL to force extended client information to be sent to the server
                conn.prepareStatement("SELECT * FROM SYSIBM.SYSDummy1" + "WHERE 0 = 1").executeQuery();
            }
        } catch (Throwable e) {
            e.printStackTrace();
        }
    }
}
```

Figure 18. Example of passing extended client information to a DB2 server

Providing extended client information to the data source with client info properties

The IBM Data Server Driver for JDBC and SQLJ version 4.0 supports JDBC 4.0 client info properties, which you can use to provide extra information about the client to the server. This information can be used for accounting, workload management, or debugging.

Extended client information is sent to the database server when the application performs an action that accesses the server, such as executing SQL.

The application can also use the Connection.getClientInfo method to retrieve client information from the database server, or execute the DatabaseMetaData.getClientInfoProperties method to determine which client information the driver supports.

The JDBC 4.0 client info properties should be used instead IBM Data Server Driver for JDBC and SQLJ-only methods, which are deprecated.

To set client info properties, follow these steps:
1. Create a Connection.
2. Call the java.sql.Connection.setClientInfo method to set any of the client info properties that the database server supports.
3. Execute an SQL statement to cause the information to be sent to the database server.

The following code performs the previous steps to pass a client's user name and host name to the DB2 server. The numbers to the right of selected statements correspond to the previously-described steps.
public class ClientInfoTest {
    public static void main(String[] args) {
        String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            String user = "db2adm";
            String password = "db2adm";
            Connection conn = DriverManager.getConnection(url, 1
                user, password);
            conn.setClientInfo("ClientUser", "Michael L Thompson"); 2
            conn.setClientInfo("ClientHostname", "sjwkstn1");
            // Execute SQL to force extended client information to be sent
            // to the server
            conn.prepareStatement("SELECT * FROM SYSIBM.SYSDUMMY1" + "WHERE 0 = 1").executeQuery(); 3
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }
    }

Figure 19. Example of passing extended client information to a DB2 server

Client info properties support by the IBM Data Server Driver for JDBC and SQLJ

JDBC 4.0 includes client info properties, which contain information about a connection to a data source. The DatabaseMetaData.getClientInfoProperties method returns a list of client info properties that the IBM Data Server Driver for JDBC and SQLJ supports.

When you call DatabaseMetaData.getClientInfoProperties, a result set is returned that contains the following columns:
- NAME
- MAX_LEN
- DEFAULT_VALUE
- DESCRIPTION

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for DB2 Database for Linux, UNIX, and Windows and for DB2 for i.

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>255</td>
<td>Empty string</td>
<td>The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>255</td>
<td>Empty string</td>
<td>The value of the accounting string that is specified for the connection. This value is stored in DB2 special register CURRENT CLIENT_ACCTNG.</td>
</tr>
</tbody>
</table>
Table 15. Client info property values for DB2 Database for Linux, UNIX, and Windows and for DB2 for i (continued)

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientHostname</td>
<td>255</td>
<td>The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host.</td>
<td></td>
</tr>
<tr>
<td>ClientUser</td>
<td>255</td>
<td>Empty string</td>
<td>The name of the user on whose behalf the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_USERID.</td>
</tr>
</tbody>
</table>

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for DB2 for z/OS when the connection uses type 4 connectivity.

Table 16. Client info property values for type 4 connectivity to DB2 for z/OS

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>32</td>
<td>clientProgramName property value, if set. &quot;db2jcc_application&quot; otherwise.</td>
<td>The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>200</td>
<td>A string that is the concatenation of the following values:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&quot;JCCnnnnn&quot;, where nnnnn is the driver level, such as 04000.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The value that is set by DB2Connection.setDB2ClientWorkstation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>If the value is not set, the default is the host name of the local host.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>applicationName property value, if set. 20 blanks otherwise.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>clientUser property value, if set. Eight blanks otherwise.</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>18</td>
<td>The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host.</td>
<td></td>
</tr>
<tr>
<td>ClientUser</td>
<td>16</td>
<td>The value that is set by DB2Connection.setDB2ClientUser. If the value is not set, the default is the current user ID that is used to connect to the database.</td>
<td></td>
</tr>
</tbody>
</table>

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for DB2 for z/OS when the connection uses type 2 connectivity.
### Table 17. Client info property values for type 2 connectivity to DB2 for z/OS

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>32</td>
<td>Empty string</td>
<td>The name of the application that is currently using the connection. This value is stored in DB2 special register CURRENT CLIENT_APPLNAME.</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>200</td>
<td>Empty string</td>
<td>The value of the accounting string from the client information that is specified for the connection. This value is stored in DB2 special register CURRENT CLIENT_ACCTNG.</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>18</td>
<td>Empty string</td>
<td>The host name of the computer on which the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_WRKSTNNAME.</td>
</tr>
<tr>
<td>ClientUser</td>
<td>16</td>
<td>Empty string</td>
<td>The name of the user on whose behalf the application that is using the connection is running. This value is stored in DB2 special register CURRENT CLIENT_USERID.</td>
</tr>
</tbody>
</table>

The following table lists the client info property values that the IBM Data Server Driver for JDBC and SQLJ returns for IBM Informix

### Table 18. Client info property values for IBM Informix

<table>
<thead>
<tr>
<th>NAME</th>
<th>MAX_LEN (bytes)</th>
<th>DEFAULT_VALUE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplicationName</td>
<td>20</td>
<td>Empty string</td>
<td>The name of the application that is currently using the connection.</td>
</tr>
<tr>
<td>ClientAccountingInformation</td>
<td>199</td>
<td>Empty string</td>
<td>The value of the accounting string from the client information that is specified for the connection.</td>
</tr>
<tr>
<td>ClientHostname</td>
<td>20</td>
<td>The value that is set by DB2Connection.setDB2ClientWorkstation. If the value is not set, the default is the host name of the local host.</td>
<td>The host name of the computer on which the application that is using the connection is running.</td>
</tr>
<tr>
<td>ClientUser</td>
<td>1024</td>
<td>Empty string</td>
<td>The name of the user on whose behalf the application that is using the connection is running.</td>
</tr>
</tbody>
</table>

### Optimistic locking in JDBC applications

You can write JDBC applications to take advantage of optimistic locking on a data source.

*Optimistic locking* is a technique that applications can use to release locks between SELECT and UPDATE or DELETE operations. If the selected rows change before that application updates or deletes them, the UPDATE or DELETE operation fails. Optimistic locking minimizes the time during which a given resource is unavailable for use by other transactions.

For connections to a DB2 for i data source, use of optimistic locking requires DB2 for i V6R1 or later.
In general, an application performs these steps to use optimistic locking:
1. Select rows from a table.
2. Release locks on the table.
3. Update the selected rows, if they have not changed.

To check whether the row has changed, the application queries the row change token. The row change token is not always a completely accurate indicator of whether the row has changed. If you create a table with a row change timestamp column, the row change token is completely accurate. If you create the table without a row change timestamp column, or alter a table to add a row change timestamp column, the row change token might not accurately reflect updates to a row. This means that the row change token might indicate that a row has changed, even though it has not. This condition is called a *false negative* condition.

When you write a JDBC application to perform optimistic locking you follow similar steps:
1. Prepare and execute a query.
   Indicate whether you want optimistic locking information, and whether that information can include false negatives.
2. Determine whether the ResultSet has optimistic locking information, and whether that information can produce false negatives.
   Based on the type of optimistic locking information, you can decide whether to continue with optimistic locking.
4. Update the selected rows, if the row change token indicates that they have not changed.

The following code demonstrates how a JDBC application can perform optimistic locking. The numbers in the example correspond to the previously listed steps.

```java
com.ibm.db2.jcc.DB2Statement s1 =
  (com.ibm.db2.jcc.DB2Statement)conn.createStatement();
ResultSet rs =
  (com.ibm.db2.jcc.DB2Statement)s1.executeDB2OptimisticLockingQuery
  ("SELECT EMPNO, SALARY FROM EMP WHERE EMP.LASTNAME = 'HAAS'",
   com.ibm.db2.jcc.DB2Statement.RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES); 1
  // Indicate that you plan to do
  // optimistic locking, and that you
  // want optimistic locking information
  // that does not generate
  // false negatives
ResultSetMetaData rsmd = rs.getMetaData();
int optColumns =
  ((com.ibm.db2.jcc.ResultSetMetaData)rsmd).getDB2OptimisticLockingColumns(); 2
  // Retrieve the optimistic locking
  // information.
boolean optColumnsReturned = false;
if (optColumns == 0) { // If optimistic locking information is not
  // returned, do not attempt to do
  // optimistic locking.
  else if (optColumns == 1) { // A value of 1 is never returned if
    // RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES
    // is specified, because 1 indicates
    // that there could be false negatives.
    else if (optColumns == 2) { // If optimistic locking information is
      // returned, and false negatives will not
      // occur, try optimistic locking.
        optColumnsReturned = true;
    }
    rs.next(); // Retrieve the contents of the ResultSet
    int emp_id = rs.getInt(1);
    double salary = rs.getDouble(2);
    long rowChangeToken = 0;
    Object rid = null;
```

88 Developing Java Applications
int type = -1;

if (optColumnsReturned) {
    rowChangeToken = ((com.ibm.db2.jcc.DB2ResultSet)rs).getDB2RowChangeToken();
    rid = ((com.ibm.db2.jcc.DB2ResultSet)rs).getDB2RID();
    // Get the RID, which uniquely identifies the row.
    int type = ((com.ibm.db2.jcc.DB2ResultSet)rs).getDB2RIDType();
    // Get the data type of the RID.
}

// ***************************************************
// Release the locks or disconnect from the database.
// Perform some work on the retrieved data.
// Reconnect to the data source.
// ***************************************************
...

PreparedStatement s2 =
    conn.prepareStatement("UPDATE EMP SET SALARY=?" +
    "WHERE EMPNO = ? AND ROW CHANGE TOKEN FOR EMP = ? and " +
    "RID_BIT(EMP) = ?");
    // Statement for updating the previously selected rows that have not changed.
    s2.setDouble(1, salary+10000);
    s2.setInt(2, emp_id);
    // Set the new row values.
    s2.setLong(3, rowChangeToken);
    // Set the row change token of the previously retrieved row.
    if (type == java.sql.Types.BIGINT)
        s2.setLong (4, ((Long)rid).longValue());
    else if (type == java.sql.Types.VARBINARY)
        s2.setBytes (4, (byte[])rid);
        // Set the RID of the previously retrieved row.
        // Use the correct setXXX method for the data type of the RID.
    int updateCount = s2.executeUpdate();
    // Perform the update.
    if (updateCount == 1) { // Update is successful.
        else // Update failed.
        ...

XML data in JDBC applications

In JDBC applications, you can store data in XML columns and retrieve data from XML columns.

In database tables, the XML built-in data type is used to store XML data in a column as a structured set of nodes in a tree format.

JDBC applications can send XML data to the data server or retrieve XML data from the data server as textual XML data.

In JDBC applications, you can:
- Store an entire XML document in an XML column using setXXX methods.
- Retrieve an entire XML document from an XML column using getXXX methods.
- Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function to retrieve the sequence into a serialized sequence in the database, and then using getXXX methods to retrieve the data into an application variable.
- Retrieve a sequence from a document in an XML column by using an XQuery expression, prepended with the string 'XQUERY', to retrieve the elements of the sequence into a result table in the database, in which each row of the result table represents an item in the sequence. Then use getXXX methods to retrieve the data into application variables.
- Retrieve a sequence from a document in an XML column as a user-defined table by using the SQL XMLTABLE function to define the result table and retrieve it. Then use getXXX methods to retrieve the data from the result table into application variables.

JDBC 4.0 `java.sql.SQLXML` objects can be used to retrieve and update data in XML columns. Invocations of metadata methods, such as `ResultSetMetaData.getColumnName` return the integer value `java.sql.Types.SQLXML` for an XML column type.

**XML column updates in JDBC applications**

When you update or insert data into XML columns of a database table, the input data in your JDBC applications must be in the textual format.

The following table lists the methods and corresponding input data types that you can use to put data in XML columns.

<table>
<thead>
<tr>
<th>Method</th>
<th>Input data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreparedStatement.setAsciiStream</td>
<td>InputStream</td>
</tr>
<tr>
<td>PreparedStatement.setBinaryStream</td>
<td>InputStream</td>
</tr>
<tr>
<td>PreparedStatement.setBlob</td>
<td>Blob</td>
</tr>
<tr>
<td>PreparedStatement.setBytes</td>
<td>byte[]</td>
</tr>
<tr>
<td>PreparedStatement.setCharacterStream</td>
<td>Reader</td>
</tr>
<tr>
<td>PreparedStatement.setClob</td>
<td>Clob</td>
</tr>
<tr>
<td>PreparedStatement.setObject</td>
<td>byte[], Blob, Clob, SQLXML, DB2Xml (deprecated), InputStream, Reader, String</td>
</tr>
<tr>
<td>PreparedStatement.setSQLXML[]</td>
<td>SQLXML</td>
</tr>
<tr>
<td>PreparedStatement.setString</td>
<td>String</td>
</tr>
</tbody>
</table>

Note:
1. This method requires JDBC 4.0 or later.

The encoding of XML data can be derived from the data itself, which is known as *internally encoded* data, or from external sources, which is known as *externally encoded* data. XML data that is sent to the database server as binary data is treated as internally encoded data. XML data that is sent to the data source as character data is treated as externally encoded data.

External encoding for Java applications is always Unicode encoding.

Externally encoded data can have internal encoding. That is, the data might be sent to the data source as character data, but the data contains encoding information. The data source handles incompatibilities between internal and external encoding as follows:

- If the data source is DB2 Database for Linux, UNIX, and Windows, the database source generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the database source ignores the internal encoding.
- If the database source is DB2 for z/OS, the database source ignores the internal encoding.
Data in XML columns is stored in UTF-8 encoding. The database source handles conversion of the data from its internal or external encoding to UTF-8.

**Example:** The following example demonstrates inserting data from an SQLXML object into an XML column. The data is String data, so the database source treats the data as externally encoded.

```java
public void insertSQLXML() {
    Connection con = DriverManager.getConnection(url);
    SQLXML info = con.createSQLXML();
    PreparedStatement insertStmt = null;
    String infoData = "<customerinfo xmlns="http://posample.org" Cid="1000">...</customerinfo>";
    info.setString(infoData);
    // Populate the SQLXML object
    int cid = 1000;
    try {
        sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
        insertStmt = con.prepareStatement(sqls);
        insertStmt.setInt(1, cid);
        insertStmt.setSQLXML(2, info);
        // Assign the SQLXML object value
        // to an input parameter
        if (insertStmt.executeUpdate() != 1) {
            System.out.println("insertSQLXML: No record inserted.\n");
        }
    } catch (IOException ioe) {
        ioe.printStackTrace();
    } catch (SQLException sqle) {
        System.out.println("insertSQLXML: SQL Exception: " + sqle.getMessage());
        System.out.println("insertSQLXML: SQL State: " + sqle.getSQLState());
        System.out.println("insertSQLXML: SQL Error Code: " + sqle.getErrorCode());
    }
}
```

**Example:** The following example demonstrates inserting data from a file into an XML column. The data is inserted as binary data, so the database server honors the internal encoding.

```java
public void insertBinStream(Connection conn) {
    PreparedStatement insertStmt = null;
    String sqls = null;
    int cid = 0;
    Statement stmt=null;
    try {
        sqls = "INSERT INTO CUSTOMER (CID, INFO) VALUES (?, ?)";
        insertStmt = conn.prepareStatement(sqls);
        insertStmt.setInt(1, cid);
        File file = new File(fn);
        insertStmt.setBinaryStream(2, new FileInputStream(file), (int)file.length());
        if (insertStmt.executeUpdate() != 1) {
            System.out.println("insertBinStream: No record inserted.\n");
        }
    } catch (IOException ioe) {
    }
    catch (SQLException sqle) {
        System.out.println("insertBinStream: SQL Exception: " + sqle.getMessage());
        System.out.println("insertBinStream: SQL State: " + sqle.getSQLState());
    }
}
```
XML data retrieval in JDBC applications

In JDBC applications, you use ResultSet.getXXX or ResultSet.getObject methods to retrieve data from XML columns.

When you retrieve data from XML columns of a DB2 table, the output data is in the serialized string format. This is true whether you retrieve the entire contents of an XML column or a sequence from the column.

You can use one of the following techniques to retrieve XML data:

- Use the ResultSet.getSQLXML method to retrieve the data. Then use a SQLXML.getXXX method to retrieve the data into a compatible output data type. This technique requires JDBC 4.0 or later.
  For example, you can retrieve data by using the SQLXML.getBinaryStream method or the SQLXML.getSource method.
- Use a ResultSet.getXXX method other than ResultSet.getObject to retrieve the data into a compatible data type.
- Use the ResultSet.getObject method to retrieve the data, and then cast it to the DB2Xml type and assign it to a DB2Xml object. Then use a DB2Xml.getDB2XXX or DB2Xml.getDB2XmlXXX method to retrieve the data into a compatible output data type.
  You need to use this technique if you are not using a version of the IBM Data Server Driver for JDBC and SQLJ that supports JDBC 4.0.

The following table lists the ResultSet methods and corresponding output data types for retrieving XML data.

<table>
<thead>
<tr>
<th>Method</th>
<th>Output data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResultSet.getAsciiStream</td>
<td>InputStream</td>
</tr>
<tr>
<td>ResultSet.getBinaryStream</td>
<td>InputStream</td>
</tr>
<tr>
<td>ResultSet.getBytes</td>
<td>byte[]</td>
</tr>
<tr>
<td>ResultSet.getCharacterStream</td>
<td>Reader</td>
</tr>
<tr>
<td>ResultSet.getObject</td>
<td>Object</td>
</tr>
<tr>
<td>ResultSet.getSQLXML</td>
<td>SQLXML</td>
</tr>
<tr>
<td>ResultSet.getString</td>
<td>String</td>
</tr>
</tbody>
</table>

The following table lists the methods that you can call to retrieve data from a java.sql.SQLXML or a com.ibm.db2.jcc.DB2Xml object, and the corresponding output data types and type of encoding in the XML declarations.
### Table 21. SQLXML and DB2Xml methods, data types, and added encoding specifications

<table>
<thead>
<tr>
<th>Method</th>
<th>Output data type</th>
<th>Type of XML internal encoding declaration added</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLXML.getBinaryStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getCharacterStream</td>
<td>Reader</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getSource</td>
<td>Source</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getString</td>
<td>String</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2AsciiStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2BinaryStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2Bytes</td>
<td>byte[]</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2CharacterStream</td>
<td>Reader</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2String</td>
<td>String</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlAsciiStream</td>
<td>InputStream</td>
<td>US-ASCII</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlBinaryStream</td>
<td>InputStream</td>
<td>Specified by getDB2XmlBinaryStream <em>targetEncoding</em> parameter</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlBytes</td>
<td>byte[]</td>
<td>Specified by DB2Xml.getDB2XmlBytes <em>targetEncoding</em> parameter</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlCharacter</td>
<td>Reader</td>
<td>ISO-10646-UCS-2</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlString</td>
<td>String</td>
<td>ISO-10646-UCS-2</td>
</tr>
</tbody>
</table>

**Note:**
1. The class that is returned is specified by the invoker of getSource, but the class must extend javax.xml.transform.Source.

If the application executes the XMLSERIALIZE function on the data that is to be returned, after execution of the function, the data has the data type that is specified in the XMLSERIALIZE function, not the XML data type. Therefore, the driver handles the data as the specified type and ignores any internal encoding declarations.

**Example:** The following example demonstrates retrieving data from an XML column into an SQLXML object, and then using the SQLXML.getString method to retrieve the data into a string.

```java
public void fetchToSQLXML(long cid, java.sql.Connection conn)
{
    System.out.println(">> fetchToSQLXML: Get XML data as an SQLXML object + using getSQLXML");
    PreparedStatement selectStmt = null;
    String sqls = null, stringDoc = null;
    try{
        sqls = "SELECT info FROM customer WHERE cid = " + cid;
        selectStmt = conn.prepareStatement(sqls);
        rs = selectStmt.executeQuery();
        while (rs.next()) {
            // Get metadata
            // Column type for XML column is the integer java.sql.Types.OTHER
            ResultSetMetaData meta = rs.getMetaData();
            System.out.println("fetchToSQLXML: Column type = " + meta.getColumnType(1));
            // Retrieve the XML data with getSQLXML.
            // Then write it to a string with
            // explicit internal ISO-10646-UCS-2 encoding.
        }
    }
```
java.sql.SQLXML xml = rs.getSQLXML(1);
System.out.println (xml.getString());
}
rs.close();
}
catch (SQLException sqle) {
  System.out.println("fetchToSQLXML: SQL Exception: " +
  sqle.getMessage());
  System.out.println("fetchToSQLXML: SQL State: " +
  sqle.getSQLState());
  System.out.println("fetchToSQLXML: SQL Error Code: " +
  sqle.getErrorCode());
}
}

Example: The following example demonstrates retrieving data from an XML column into a String variable.

public void fetchToString(long cid, java.sql.Connection conn) {
  System.out.println(">> fetchToString: Get XML data " +
  "using getString");
  PreparedStatement selectStmt = null;
  String sqls = null, stringDoc = null;
  ResultSet rs = null;
  try{
    sqls = "SELECT info FROM customer WHERE cid = " + cid;
    selectStmt = conn.prepareStatement(sqls);
    rs = selectStmt.executeQuery();
    // Get metadata
    // Column type for XML column is the integer java.sql.Types.OTHER
    ResultSetMetaData meta = rs.getMetaData();
    int colType = meta.getColumnType(1);
    System.out.println("fetchToString: Column type = " + colType);
    while (rs.next()) {
      stringDoc = rs.getString(1);
      System.out.println("Document contents:");
      System.out.println(stringDoc);
    }
  } catch (SQLException sqle) {
    System.out.println("fetchToString: SQL Exception: " +
    sqle.getMessage());
    System.out.println("fetchToString: SQL State: " +
    sqle.getSQLState());
    System.out.println("fetchToString: SQL Error Code: " +
    sqle.getErrorCode());
  }
}

Example: The following example demonstrates retrieving data from an XML column into a DB2Xml object, and then using the DB2Xml.getDB2XmlString method to retrieve the data into a string with an added XML declaration with an ISO-10646-UCS-2 encoding specification.

public void fetchToDB2Xml(long cid, java.sql.Connection conn) {
  System.out.println(">> fetchToDB2Xml: Get XML data as a DB2XML object " +
  "using getDB2Xml");
  PreparedStatement selectStmt = null;
  String sqls = null, stringDoc = null;
  ResultSet rs = null;
  try{
    sqls = "SELECT info FROM customer WHERE cid = " + cid;
    selectStmt = conn.prepareStatement(sqls);
    rs = selectStmt.executeQuery();
    // Get metadata
    // Column type for XML column is the integer java.sql.Types.OTHER
    ResultSetMetaData meta = rs.getMetaData();
    int colType = meta.getColumnType(1);
    System.out.println("fetchToDB2Xml: Column type = " + colType);
    while (rs.next()) {
      JSONObject result = rs.getDB2Xml(1);
      System.out.println("Result: ");
      System.out.println(result.toString());
    }
  } catch (SQLException sqle) {
    System.out.println("fetchToDB2Xml: SQL Exception: " +
    sqle.getMessage());
    System.out.println("fetchToDB2Xml: SQL State: " +
    sqle.getSQLState());
    System.out.println("fetchToDB2Xml: SQL Error Code: " +
    sqle.getErrorCode());
  }
}
Invocation of routines with XML parameters in Java applications

Java applications can call stored procedures at DB2 Database for Linux, UNIX, and Windows data sources that have XML parameters.

For native SQL procedures, XML parameters in the stored procedure definition have the XML type. For external stored procedures and user-defined functions on DB2 Database for Linux, UNIX, and Windows data sources, XML parameters in the routine definition have the XML AS CLOB type. When you call a stored procedure or user-defined function that has XML parameters, you need to use a compatible data type in the invoking statement.

To call a routine with XML input parameters from a JDBC program, use parameters of the java.sql.SQLXML or com.ibm.db2.jcc.DB2Xml type. To register XML output parameters, register the parameters as the java.sql.Types.SQLXML or com.ibm.db2.jcc.DB2Types.XML type. (The com.ibm.db2.jcc.DB2Types.XML types are deprecated.)

**Example:** JDBC program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter. This example requires JDBC 4.0.

```java
java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML inout_xml = xmlvar;

// Declare an input, output, and
// INOUT XML parameter

Connection con;
CallableStatement cstmt;
ResultSet rs;

... cstmt = con.prepareCall("CALL SP_xml(?,?,?)");

// Create a CallableStatement object

// Set input parameter
```
cstmt.setObject (3, inout_xml); // Set inout parameter
cstmt.registerOutParameter (2, java.sql.Types.SQLXML);
// Register out and input parameters
cstmt.registerOutParameter (3, java.sql.Types.SQLXML);
cstmt.executeUpdate(); // Call the stored procedure
out_xml = cstmt.getSQLXML(2); // Get the OUT parameter value
inout_xml = cstmt.getSQLXML(3); // Get the INOUT parameter value
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value");
MyUtilities.printString(out_xml.getString());
    // Use the SQLXML.getString
    // method to convert the out_xml
    // value to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.
System.out.println("INOUT parameter value ");
MyUtilities.printString(inout_xml.getString());
    // Use the SQLXML.getString
    // method to convert the inout_xml
    // value to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.

To call a routine with XML parameters from an SQLJ program, use parameters of
the java.sql.SQLXML or com.ibm.db2.jcc.DB2Xml type.

Example: SQLJ program that calls a stored procedure that takes three XML
parameters: an IN parameter, an OUT parameter, and an INOUT parameter. This
element requires JDBC 4.0.
java.sql.SQLXML in_xml = xmlvar;
java.sql.SQLXML out_xml = null;
java.sql.SQLXML inout_xml = xmlvar;
    // Declare an input, output, and
    // INOUT XML parameter
...
#sql [myConnCtx] {CALL SP_xml(:IN in_xml,
    :OUT out_xml,
    :INOUT inout_xml)};
    // Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value");
MyUtilities.printString(out_xml.getString());
    // Use the SQLXML.getString
    // method to convert the out_xml value
    // to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.
System.out.println("INOUT parameter value ");
MyUtilities.printString(inout_xml.getString());
    // Use the SQLXML.getString
    // method to convert the inout_xml
    // value to a string for printing.
    // Call a user-defined method called
    // printString (not shown) to print
    // the value.

Java support for XML schema registration and removal
The IBM Data Server Driver for JDBC and SQLJ provides methods that let you
write Java application programs to register and remove XML schemas and their
components.
The methods are:

**DB2Connection.registerDB2XMLSchema**

Registers an XML schema in DB2, using one or more XML schema documents. There are two forms of this method: one form for XML schema documents that are input from InputStream objects, and one form for XML schema documents that are in a String.

**DB2Connection.deregisterDB2XMLObject**

Removes an XML schema definition from DB2.

**DB2Connection.updateDB2XmlSchema**

Replaces the XML schema documents in a registered XML schema with the XML schema documents from another registered XML schema. Optionally drops the XML schema whose contents are copied. This method is available only for connections to DB2 Database for Linux, UNIX, and Windows.

Before you can invoke these methods, the stored procedures that support these methods must be installed on the DB2 database server.

**Example: Registration of an XML schema:** The following example demonstrates the use of registerDB2XmlSchema to register an XML schema in DB2 using a single XML schema document (customer.xsd) that is read from an input stream. The SQL schema name for the registered schema is SYSXSR. No additional properties are registered.

```java
public static void registerSchema(
    Connection con,
    String schemaName)
    throws SQLException {
    // Define the registerDB2XmlSchema parameters
    String[] xmlSchemaNameQualifiers = new String[1];
    String[] xmlSchemaNames = new String[1];
    String[] xmlSchemaLocations = new String[1];
    InputStream[] xmlSchemaDocuments = new InputStream[1];
    int[] xmlSchemaDocumentsLengths = new int[1];
    java.io.InputStream[] xmlSchemaDocumentsProperties = new InputStream[1];
    int[] xmlSchemaDocumentsPropertiesLengths = new int[1];
    InputStream xmlSchemaProperties;
    int xmlSchemaPropertiesLength;
    //Set the parameter values
    xmlSchemaLocations[0] = "";
    FileInputStream fi = null;
    xmlSchemaNameQualifiers[0] = "SYSXSR";
    xmlSchemaNames[0] = schemaName;
    try {
        fi = new FileInputStream("customer.xsd");
        xmlSchemaDocuments[0] = new BufferedInputStream(fi);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    try {
        xmlSchemaDocumentsLengths[0] = (int) fi.getChannel().size();
        System.out.println(xmlSchemaDocumentsLengths[0]);
    } catch (IOException e1) {
        e1.printStackTrace();
    }
    xmlSchemaDocumentsProperties[0] = null;
    xmlSchemaDocumentsPropertiesLengths[0] = 0;
    xmlSchemaProperties = null;
    xmlSchemaPropertiesLength = 0;
    DB2Connection ds = (DB2Connection) con;
    // Invoke registerDB2XmlSchema
    ds.registerDB2XmlSchema(
        xmlSchemaNameQualifiers,
        xmlSchemaNames,
        xmlSchemaLocations,
        xmlSchemaDocuments,
        xmlSchemaDocumentsLengths,
        xmlSchemaDocumentsProperties,
        xmlSchemaDocumentsPropertiesLengths,
        xmlSchemaProperties,
        xmlSchemaPropertiesLength)
    ;
}
```
Example: Removal of an XML schema: The following example demonstrates the use of deregisterDB2XmlObject to remove an XML schema from DB2. The SQL schema name for the registered schema is SYSXSR.

```java
public static void deregisterSchema(
    Connection con,
    String schemaName)
    throws SQLException {
    // Define and assign values to the deregisterDB2XmlObject parameters
    String xmlSchemaNameQualifier = "SYSXSR";
    String xmlSchemaName = schemaName;
    DB2Connection ds = (DB2Connection) con;
    // Invoke deregisterDB2XmlObject
    ds.deregisterDB2XmlObject(
        xmlSchemaNameQualifier,
        xmlSchemaName);
}
```

Example: Update of an XML schema: The following example applies only to connections to DB2 Database for Linux, UNIX, and Windows. It demonstrates the use of updateDB2XmlSchema to update the contents of an XML schema with the contents of another XML schema. The schema that is copied is kept in the repository. The SQL schema name for both registered schemas is SYSXSR.

```java
public static void updateSchema(
    Connection con,
    String schemaNameTarget,
    String schemaNameSource)
    throws SQLException {
    // Define and assign values to the updateDB2XmlSchema parameters
    String xmlSchemaNameQualifierTarget = "SYSXSR";
    String xmlSchemaNameQualifierSource = "SYSXSR";
    String xmlSchemaNameTarget = schemaNameTarget;
    String xmlSchemaNameSource = schemaNameSource;
    boolean dropSourceSchema = false;
    DB2Connection ds = (DB2Connection) con;
    // Invoke updateDB2XmlSchema
    ds.updateDB2XmlSchema(
        xmlSchemaNameQualifierTarget,
        xmlSchemaNameTarget,
        xmlSchemaNameQualifierSource,
        xmlSchemaNameSource, 
        dropSourceSchema);
}
```

**Transaction control in JDBC applications**

In JDBC applications, as in other types of SQL applications, transaction control involves explicitly or implicitly committing and rolling back transactions, and setting the isolation level for transactions.
IBM Data Server Driver for JDBC and SQLJ isolation levels

The IBM Data Server Driver for JDBC and SQLJ supports a number of isolation levels, which correspond to database server isolation levels.

JDBC isolation levels can be set for a unit of work within a JDBC program, using the `Connection.setTransactionIsolation` method. The default isolation level can be set with the `defaultIsolationLevel` property.

The following table shows the values of `level` that you can specify in the `Connection.setTransactionIsolation` method and their DB2 database server equivalents.

<table>
<thead>
<tr>
<th>JDBC value</th>
<th>DB2 isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code></td>
<td>Repeatable read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_REPEATABLE_READ</code></td>
<td>Read stability</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_COMMITTED</code></td>
<td>Cursor stability</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</code></td>
<td>Uncommitted read</td>
</tr>
</tbody>
</table>

The following table shows the values of `level` that you can specify in the `Connection.setTransactionIsolation` method and their IBM Informix (IDS) equivalents.

<table>
<thead>
<tr>
<th>IDS value</th>
<th>IDS isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code></td>
<td>Repeatable read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_REPEATABLE_READ</code></td>
<td>Repeatable read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_COMMITTED</code></td>
<td>Committed read</td>
</tr>
<tr>
<td><code>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</code></td>
<td>Dirty read</td>
</tr>
<tr>
<td><code>com.ibm.db2.jcc.DB2Connection.TRANSACTION_IDS_CURSOR_STABILITY</code></td>
<td>IDS cursor stability</td>
</tr>
<tr>
<td><code>com.ibm.db2.jcc.DB2Connection.TRANSACTION_IDS_LAST_COMMITTED</code></td>
<td>Committed read, last committed</td>
</tr>
</tbody>
</table>

Committing or rolling back JDBC transactions

In JDBC, to commit or rollback transactions explicitly, use the commit or rollback methods.

For example:
```java
Connection con;
...
con.commit();
```

If autocommit mode is on, the database manager performs a commit operation after every SQL statement completes. To set autocommit mode on, invoke the `Connection.setAutoCommit(true)` method. To set autocommit mode off, invoke the `Connection.setAutoCommit(false)` method. To determine whether autocommit mode is on, invoke the `Connection.getAutoCommit` method.

Connections that participate in distributed transactions cannot invoke the `setAutoCommit(true)` method.
When you change the autocommit state, the database manager executes a commit operation, if the application is not already on a transaction boundary.

While a connection is participating in a distributed transaction, the associated application cannot issue the commit or rollback methods.

**Default JDBC autocommit modes**

The default autocommit mode depends on the data source to which the JDBC application connects.

**Autocommit default for DB2 data sources**

For connections to DB2 data sources, the default autocommit mode is `true`.

**Autocommit default for IDS data sources**

For connections to IDS data sources, the default autocommit mode depends on the type of data source. The following table shows the defaults.

<table>
<thead>
<tr>
<th>Type of data source</th>
<th>Default autocommit mode for local transactions</th>
<th>Default autocommit mode for global transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI-compliant database</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>Non-ANSI-compliant database without logging</td>
<td>false</td>
<td>not applicable</td>
</tr>
<tr>
<td>Non-ANSI-compliant database with logging</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

**Exceptions and warnings under the IBM Data Server Driver for JDBC and SQLJ**

In JDBC applications, SQL errors throw exceptions, which you handle using `try/catch` blocks. SQL warnings do not throw exceptions, so you need to invoke methods to check whether warnings occurred after you execute SQL statements.

The IBM Data Server Driver for JDBC and SQLJ provides the following classes and interfaces, which provide information about errors and warnings.

**SQLException**

The `SQLException` class for handling errors. All JDBC methods throw an instance of `SQLException` when an error occurs during their execution. According to the JDBC specification, an `SQLException` object contains the following information:

- An `int` value that contains an error code. `SQLException.getErrorCode` retrieves this value.
- A `String` object that contains the SQLSTATE, or null. `SQLException.getSQLState` retrieves this value.
- A `String` object that contains a description of the error, or null. `SQLException.getMessage` retrieves this value.
- A pointer to the next `SQLException`, or null. `SQLException.getNextException` retrieves this value.
When a JDBC method throws a single SQLException, that SQLException might be caused by an underlying Java exception that occurred when the IBM Data Server Driver for JDBC and SQLJ processed the method. In this case, the SQLException wraps the underlying exception, and you can use the SQLException.getCause method to retrieve information about the error.

**DB2Diagnosable**

The IBM Data Server Driver for JDBC and SQLJ-only interface `com.ibm.db2.jcc.DB2Diagnosable` extends the SQLException class. The DB2Diagnosable interface gives you more information about errors that occur when the data source is accessed. If the JDBC driver detects an error, DB2Diagnosable gives you the same information as the standard SQLException class. However, if the database server detects the error, DB2Diagnosable adds the following methods, which give you additional information about the error:

- `getSqlca`
  - Returns an `DB2Sqlca` object with the following information:
    - An SQL error code
    - The SQLERRMC values
    - The SQLERRP value
    - The SQLERRD values
    - The SQLWARN values
    - The SQLSTATE

- `getThrowable`
  - Returns a `java.lang.Throwable` object that caused the SQLException, or null, if no such object exists.

- `printTrace`
  - Prints diagnostic information.

**SQLException subclasses**

If you are using JDBC 4.0 or later, you can obtain more specific information than an SQLException provides by catching the following exception classes:

- `SQLNonTransientException`
  - An SQLNonTransientException is thrown when an SQL operation that failed previously cannot succeed when the operation is retried, unless some corrective action is taken. The SQLNonTransientException class has these subclasses:
    - SQLFeatureNotSupportedException
    - SQLNonTransientConnectionException
    - SQLDataException
    - SQLIntegrityConstraintViolationException
    - SQLInvalidAuthorizationSpecException
    - SQLSyntaxException

- `SQLTransientException`
  - An SQLTransientException is thrown when an SQL operation that failed previously might succeed when the operation is retried, without intervention from the application. A connection is still valid after an SQLTransientException is thrown. The SQLTransientException class has these subclasses:
    - SQLTransientConnectionException
    - SQLTransientRollbackException
    - SQLTimeoutException

- `SQLRecoverableException`
An SQLRecoverableException is thrown when an operation that failed previously might succeed if the application performs some recovery steps, and retries the transaction. A connection is no longer valid after an SQLRecoverableException is thrown.

- SQLClientInfoException
  A SQLClientInfoException is thrown by the Connection.setClientInfo method when one or more client properties cannot be set. The SQLClientInfoException indicates which properties cannot be set.

**BatchUpdateException**

A BatchUpdateException object contains the following items about an error that occurs during execution of a batch of SQL statements:
- A String object that contains a description of the error, or null.
- A String object that contains the SQLSTATE for the failing SQL statement, or null.
- An integer value that contains the error code, or zero.
- An integer array of update counts for SQL statements in the batch, or null.
- A pointer to an SQLException object, or null.

One BatchUpdateException is thrown for the entire batch. At least one SQLException object is chained to the BatchUpdateException object. The SQLException objects are chained in the same order as the corresponding statements were added to the batch. To help you match SQLException objects to statements in the batch, the error description field for each SQLException object begins with this string:

```
Error for batch element #n:
```

\( n \) is the number of the statement in the batch.

SQL warnings during batch execution do not throw BatchUpdateExceptions. To obtain information about warnings, use the Statement.getWarnings method on the object on which you ran the executeBatch method. You can then retrieve an error description, SQLSTATE, and error code for each SQLWarning object.

**SQLWarning**

The IBM Data Server Driver for JDBC and SQLJ accumulates warnings when SQL statements return positive SQLCODEs, and when SQL statements return 0 SQLCODEs with non-zero SQLSTATEs.

Calling getWarnings retrieves an SQLWarning object.

**Important:** When a call to Statement.executeUpdate or PreparedStatement.executeUpdate affects no rows, the IBM Data Server Driver for JDBC and SQLJ generates an SQLWarning with error code +100.

When a call to ResultSet.next returns no rows, the IBM Data Server Driver for JDBC and SQLJ does not generate an SQLWarning.

A generic SQLWarning object contains the following information:
- A String object that contains a description of the warning, or null.
- A String object that contains the SQLSTATE, or null.
- An int value that contains an error code.
- A pointer to the next SQLWarning, or null.
Under the IBM Data Server Driver for JDBC and SQLJ, like an SQLException object, an SQLWarning object can also contain DB2-specific information. The DB2-specific information for an SQLWarning object is the same as the DB2-specific information for an SQLException object.

**Handling an SQLException under the IBM Data Server Driver for JDBC and SQLJ**

As in all Java programs, error handling for JDBC applications is done using try/catch blocks. Methods throw exceptions when an error occurs, and the code in the catch block handles those exceptions.

The basic steps for handling an SQLException in a JDBC program that runs under the IBM Data Server Driver for JDBC and SQLJ are:

1. Give the program access to the com.ibm.db2.jcc.DB2Diagnosable interface and the com.ibm.db2.jcc.DB2Sqlca class. You can fully qualify all references to them, or you can import them:
   ```java
   import com.ibm.db2.jcc.DB2Diagnosable;
   import com.ibm.db2.jcc.DB2Sqlca;
   ```

2. Optional: During a connection to a data server, set the retrieveMessagesFromServerOnGetMessage property to true if you want full message text from an SQLException.getMessage call.

3. Optional: During a IBM Data Server Driver for JDBC and SQLJ type 2 connectivity connection to a DB2 for z/OS data source, set the extendedDiagnosticLevel property to EXTENDED_DIAG_MESSAGE_TEXT (241) if you want extended diagnostic information similar to the information that is provided by the SQL GET DIAGNOSTICS statement from an SQLException.getMessage call.

4. Put code that can generate an SQLException in a try block.

5. In the catch block, perform the following steps in a loop:
   a. Test whether you have retrieved the last SQLException. If not, continue to the next step.
   b. Optional: For an SQL statement that executes on an IDS data source, execute the com.ibm.db2.jcc.DB2Statement.getIDSSQLStatementOffSet method to determine which columns have syntax errors. DB2Statement.getIDSSQLStatementOffSet returns the offset into the SQL statement of the first syntax error.
   c. Optional: For an SQL statement that executes on an IDS data source, execute the SQLException.getCause method to retrieve any ISAM error messages.
      1) If the Throwable that is returned by SQLException.getCause is not null, perform one of the following sets of steps:
         • Issue SQLException.printStackTrace to print an error message that includes the ISAM error message text. The ISAM error message text is preceded by the string "Caused by:"
         • Retrieve the error code and message text for the ISAM message:
            a) Test whether the Throwable is an instance of an SQLException. If so, retrieve the SQL error code from that SQLException.
            b) Execute the Throwable.getMessage method to retrieve the text of the ISAM message.
      d. Check whether any IBM Data Server Driver for JDBC and SQLJ-only information exists by testing whether the SQLException is an instance of DB2Diagnosable. If so:
1) Cast the object to a DB2Diagnosable object.
2) Optional: Invoke the DB2Diagnosable.printTrace method to write all SQLException information to a java.io.PrintWriter object.
3) Invoke the DB2Diagnosable.getThrowable method to determine whether an underlying java.lang.Throwable caused the SQLException.
4) Invoke the DB2Diagnosable.getSqlca method to retrieve the DB2Sqlca object.
5) Invoke the DB2Sqlca.getSqlCode method to retrieve an SQL error code value.
6) Invoke the DB2Sqlca.getSqlErrmc method to retrieve a string that contains all SQLERRMC values, or invoke the DB2Sqlca.getSqlErrmcTokens method to retrieve the SQLERRMC values in an array.
7) Invoke the DB2Sqlca.getSqlErrp method to retrieve the SQLERRP value.
8) Invoke the DB2Sqlca.getSqlErrd method to retrieve the SQLERRD values in an array.
9) Invoke the DB2Sqlca.getSqlWarn method to retrieve the SQLWARN values in an array.
10) Invoke the DB2Sqlca.getSqlState method to retrieve the SQLSTATE value.
11) Invoke the DB2Sqlca.getMessage method to retrieve error message text from the data source.
   e. Invoke the SQLException.getNextException method to retrieve the next SQLException.

The following code demonstrates how to obtain IBM Data Server Driver for JDBC and SQLJ-specific information from an SQLException that is provided with the IBM Data Server Driver for JDBC and SQLJ. The numbers to the right of selected statements correspond to the previously-described steps.

```java
import java.sql.*; // Import JDBC API package
import com.ibm.db2.jcc.DB2Diagnosable; // Import packages for DB2
import com.ibm.db2.jcc.DB2Sqlca; // SQLException support
java.io.PrintWriter printWriter; // For dumping all SQLException information

String url = "jdbc:db2://myhost:9999/myDB:" + 
    "retrieveMessagesFromServerOnGetMessage=true;";
    // Set properties to retrieve full message text

String user = "db2adm";
String password = "db2adm";
java.sql.Connection con = 
    java.sql.DriverManager.getConnection (url, user, password) 
    // Connect to a DB2 for z/OS data source

... try { 
    // Code that could generate SQLExceptions
...
} catch(SQLException sqle) { 
    while(sqle != null) { 
        // Check whether there are more SQLExceptions to process
```

*Figure 20. Processing an SQLException under the IBM Data Server Driver for JDBC and SQLJ*
Optional IBM Data Server Driver for JDBC and SQLJ-only

```java
if (sqle instanceof DB2Diagnosable) {
    // Check if IBM Data Server Driver for JDBC and SQLJ-only
    // information exists
    com.ibm.db2.jcc.DB2Diagnosable diagnosable =
        (com.ibm.db2.jcc.DB2Diagnosable)sqle;
    diagnosable.printTrace (printWriter, "");
    java.lang.Throwable throwable =
        diagnosable.getThrowable();
    if (throwable != null) {
        // Extract java.lang.Throwable information
        // such as message or stack trace.
        ...
    }
    DB2Sqlca sqlca = diagnosable.getSqlca();
    // Get DB2Sqlca object
    if (sqlca != null) {
        // Check that DB2Sqlca is not null
        int sqlCode = sqlca.getSqlCode(); // Get the SQL error code
        String sqlErrmc = sqlca.getSqlErrmc(); // Get the entire SQLERRMC
        String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
        // You can also retrieve the
        // individual SQLERRMC tokens
        String sqlErrp = sqlca.getSqlErrp(); // Get the SQLERRP
        int[] sqlErrd = sqlca.getSqlErrd(); // Get SQLERRD fields
        char[] sqlWarn = sqlca.getSqlWarn(); // Get SQLWARN fields
        String sqlState = sqlca.getSqlState(); // Get SQLSTATE
        String errMessage = sqlca.getMessage();
        System.err.println ("Server error message: " + errMessage);
        System.err.println ("--------------- SQLCA ---------------");
        System.err.println ("Error code: " + sqlCode);
        System.err.println ("SQLERRMC: " + sqlErrmc);
        if (sqlErrmcTokens != null) {
            for (int i=0; i< sqlErrmcTokens.length; i++) {
                System.err.println (" token " + i + ": " + sqlErrmcTokens[i]);
            }
        }
        System.err.println ("SQLERRP: " + sqlErrp);
        System.err.println ("SQLERRD(1): " + sqlErrd[0] + "\n" +
            "SQLERRD(2): " + sqlErrd[1] + "\n" +
        );
        System.err.println ("SQLWARN1: " + sqlWarn[0] + "\n" +
            "SQLWARN2: " + sqlWarn[1] + "\n" +
            "SQLWARN4: " + sqlWarn[3] + "\n" +
            "SQLWARN5: " + sqlWarn[4] + "\n" +
            "SQLWARN6: " + sqlWarn[5] + "\n" +
            "SQLWARN7: " + sqlWarn[6] + "\n" +
            "SQLWARN8: " + sqlWarn[7] + "\n" +
            "SQLWARN9: " + sqlWarn[8] + "\n" +
            "SQLWARN10: " + sqlWarn[9] +
        );
        System.err.println ("SQLSTATE: " + sqlState);
        // portion of SQLException
    }
}
```
Handling an SQLWarning under the IBM Data Server Driver for JDBC and SQLJ

Unlike SQL errors, SQL warnings do not cause JDBC methods to throw exceptions. Instead, the Connection, Statement, PreparedStatement, CallableStatement, and ResultSet classes contain getWarnings methods, which you need to invoke after you execute SQL statements to determine whether any SQL warnings were generated.

The basic steps for retrieving SQL warning information are:

1. Optional: During connection to the database server, set properties that affect SQLWarning objects.
   If you want full message text from a data server when you execute SQLWarning.getMessage calls, set the retrieveMessagesFromServerOnGetMessage property to true.
   If you are using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 for z/OS data source, and you want extended diagnostic information that is similar to the information that is provided by the SQL GET DIAGNOSTICS statement when you execute SQLWarning.getMessage calls, set the extendedDiagnosticLevel property to EXTENDED_DIAG_MESSAGE_TEXT (241).

2. Immediately after invoking a method that connects to a database server or executes an SQL statement, invoke the getWarnings method to retrieve an SQLWarning object.

3. Perform the following steps in a loop:
   a. Test whether the SQLWarning object is null. If not, continue to the next step.
   b. Invoke the SQLWarning.getMessage method to retrieve the warning description.
   c. Invoke the SQLWarning.getSQLState method to retrieve the SQLSTATE value.
   d. Invoke the SQLWarning.getErrorCode method to retrieve the error code value.
   e. If you want DB2-specific warning information, perform the same steps that you perform to get DB2-specific information for an SQLException.
   f. Invoke the SQLWarning.getNextWarning method to retrieve the next SQLWarning.

The following code illustrates how to obtain generic SQLWarning information. The numbers to the right of selected statements correspond to the previously-described steps.
Retrieving information from a BatchUpdateException

When an error occurs during execution of a statement in a batch, processing continues. However, executeBatch throws a BatchUpdateException.

To retrieve information from the BatchUpdateException, follow these steps:

1. Use the BatchUpdateException.getUpdateCounts method to determine the number of rows that each SQL statement in the batch updated before the exception was thrown.
   
   getUpdateCount returns an array with an element for each statement in the batch. An element has one of the following values:
   
   - **n**  
     The number of rows that the statement updated.

   - **Statement.SUCCESS_NO_INFO**  
     This value is returned if the number of updated rows cannot be determined. The number of updated rows cannot be determined if the following conditions are true:
     
     - The application is connected to a subsystem that is in DB2 for z/OS Version 8 new-function mode, or later.
     - The application is using Version 3.1 or later of the IBM Data Server Driver for JDBC and SQLJ.
     - The IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT operations to execute batch updates.

   - **Statement.EXECUTE_FAILED**  
     This value is returned if the statement did not execute successfully.

2. If the batched statement can return automatically generated keys:
   a. Cast the BatchUpdateException to a
      
      com.ibm.db2.jcc.DBBatchUpdateException.
b. Call the DDBatchUpdateException.getDBGeneratedKeys method to retrieve an array of ResultSet objects that contains the automatically generated keys for each execution of the batched SQL statement.

c. Test whether each ResultSet in the array is null.
   Each ResultSet contains:
   • If the ResultSet is not null, it contains the automatically generated keys for an execution of the batched SQL statement.
   • If the ResultSet is null, execution of the batched statement failed.

3. Use SQLException methods getMessage, getSQLState, and getErrorCode to retrieve the description of the error, the SQLSTATE, and the error code for the first error.

4. Use the BatchUpdateException.getNextException method to get a chained SQLException.

5. In a loop, execute the getMessage, getSQLState, getErrorCode, and getNextException method calls to obtain information about an SQLException and get the next SQLException.

The following code fragment demonstrates how to obtain the fields of a BatchUpdateException and the chained SQLException objects for a batched statement that returns automatically generated keys. The example assumes that there is only one column in the automatically generated key, and that there is always exactly one key value, whose data type is numeric. The numbers to the right of selected statements correspond to the previously-described steps.

```java
try {
    // Batch updates
} catch(BatchUpdateException buex) {
    System.err.println("Contents of BatchUpdateException:");
    System.err.println(" Update counts: ");
    int [] updateCounts = buex.getUpdateCounts();
    for (int i = 0; i < updateCounts.length; i++) {
        System.err.println(" Statement "+i+": " + updateCounts[i]);
    }
    ResultSet[] resultList = ((DDBatchUpdateException)buex).getDBGeneratedKeys();
    for (i = 0; i < resultList.length; i++)
    {
        if (resultList[i] == null)
            continue; // Skip the ResultSet for which there was a failure
        else
            { rs.next();
                java.math.BigDecimal idColVar = rs.getBigDecimal(1);
                // Get automatically generated key value
                System.out.println("Automatically generated key value = " + idColVar);
            }
    } System.err.println(" Message: " + buex.getMessage());
    System.err.println(" SQLSTATE: "+ buex.getSQLState());
    System.err.println(" Error code: " + buex.getErrorCode());
    SQLException ex = buex.getNextException();
    while (ex != null) {
        System.err.println("SQL exception:");
        System.err.println(" Message: " + ex.getMessage());
        System.err.println(" SQLSTATE: "+ ex.getSQLState());
        System.err.println(" Error code: " + ex.getErrorCode());
        ex = ex.getNextException();
    }
}
```
Handling an SQLException under the DB2 JDBC Type 2 Driver
(deprecated)

As in all Java programs, error handling under the DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows is done using try/catch blocks. Methods throw exceptions when an error occurs, and the code in the catch block handles those exceptions.

The DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows uses the SQLException class for handling errors. All JDBC methods throw an instance of SQLException when an error occurs during their execution. According to the JDBC specification, an SQLException object contains the following information:

- A String object that contains a description of the error, or null
- A String object that contains the SQLSTATE, or null
- An int value that contains an error code
- A pointer to the next SQLException, or null

The basic steps for handling an SQLException in a JDBC program that runs under the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) are:

1. Put code that can generate an SQLException in a try block.
2. In the catch block, perform the following steps in a loop:
   - a. Test whether you have retrieved the last SQLException. If not, continue to the next step.
   - b. Retrieve error information from the SQLException.
   - c. Invoke the SQLException.getNextException method to retrieve the next SQLException.

The following code illustrates a catch block that uses the DB2 version of SQLException that is provided with the DB2 JDBC Type 2 Driver. The numbers to the right of selected statements correspond to the previously-described steps.

```java
import java.sql.*; // Import JDBC API package
...
try {
    // Code that could generate SQLExceptions
    ...
} catch(SQLException sqle) {
    while(sqle != null) { // Check whether there are more
        System.out.println("Message: " + sqle.getMessage());  // 1
        System.out.println("SQLSTATE: " + sqle.getSQLState());  // 2a
        System.out.println("SQL error code: " + sqle.getErrorCode());  // 2b
        sqle=sqle.getNextException(); // Retrieve next SQLException  // 2c
    }
}
```

Figure 22. Processing an SQLException under the IBM Data Server Driver for JDBC and SQLJ

Handling an SQLWarning under the DB2 JDBC Type 2 Driver

Unlike SQL errors, SQL warnings do not cause JDBC methods to throw exceptions. Instead, the Connection, Statement, PreparedStatement, CallableStatement, and ResultSet classes contain getWarnings methods, which you need to invoke after you execute SQL statements to determine whether any SQL warnings were generated.

Calling getWarnings retrieves an SQLWarning object.
The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) generates generic SQLWarning objects. A generic SQLWarning object contains the following information:

- A String object that contains a description of the warning, or null
- A String object that contains the SQLSTATE, or null
- An int value that contains an error code
- A pointer to the next SQLWarning, or null

The basic steps for retrieving SQL warning information are:

1. Immediately after invoking a method that executes an SQL statement, invoke the getWarnings method to retrieve an SQLWarning object.
2. Perform the following steps in a loop:
   a. Test whether the SQLWarning object is null. If not, continue to the next step.
   b. Invoke the SQLWarning.getMessage method to retrieve the warning description.
   c. Invoke the SQLWarning.getSQLState method to retrieve the SQLSTATE value.
   d. Invoke the SQLWarning.getErrorCode method to retrieve the error code value.
   e. Invoke the SQLWarning.getNextWarning method to retrieve the next SQLWarning.

The following code illustrates how to obtain generic SQLWarning information. The numbers to the right of selected statements correspond to the previously-described steps.

```java
Connection con;
Statement stmt;
ResultSet rs;
SQLWarning sqlwarn;
...
stmt = con.createStatement(); // Create a Statement object
rs = stmt.executeQuery("SELECT * FROM EMPLOYEE"); // Get the result table from the query
sqlwarn = stmt.getWarnings(); // Get any warnings generated
while (sqlwarn != null) { // While there are warnings, get and print warning information
    System.out.println ("Warning description: " + sqlwarn.getMessage()); // 2b
    System.out.println ("SQLSTATE: " + sqlwarn.getSQLState()); // 2c
    System.out.println ("Error code: " + sqlwarn.getErrorCode()); // 2d
    sqlwarn=sqlwarn.getNextWarning(); // Get next SQLWarning // 2e
}
```

**Figure 23. Processing an SQLWarning**

---

**Disconnecting from data sources in JDBC applications**

When you have finished with a connection to a data source, it is **essential** that you close the connection to the data source. Doing this releases the Connection object’s database and JDBC resources immediately.

To close the connection to the data source, use the close method. For example:

```java
Connection con;
...
con.close();
```
For a connection to a DB2 data source, if autocommit mode is not on, the connection needs to be on a unit-of-work boundary before you close the connection.

For a connection to an IBM Informix database, if the database supports logging, and autocommit mode is not on, the connection needs to be on a unit-of-work boundary before you close the connection.
Chapter 4. SQLJ application programming

Writing an SQLJ application has much in common with writing an SQL application in any other language.

In general, you need to do the following things:
• Import the Java packages that contain SQLJ and JDBC methods.
• Declare variables for sending data to or retrieving data from DB2 tables.
• Connect to a data source.
• Execute SQL statements.
• Handle SQL errors and warnings.
• Disconnect from the data source.

Although the tasks that you need to perform are similar to those in other languages, the way that you execute those tasks, and the order in which you execute those tasks, is somewhat different.

Example of a simple SQLJ application

A simple SQLJ application demonstrates the basic elements that JDBC applications need to include.

Figure 24. Simple SQLJ application

```java
import sqlj.runtime.*;
import java.sql.*;

#sql context EzSqljCtx;
#sql iterator EzSqljNameIter (String LASTNAME);

public class EzSqlj {
    public static void main(String args[]) throws SQLException {
        EzSqljCtx ctx = null;
        String URLprefix = "jdbc:db2:"
        String url = new String(URLprefix + args[0]);
        // Location name is an input parameter
        String hvmgr="000010";
        String hvdeptno="A00";
        try {
            Class.forName("com.ibm.db2.jcc.DB2Driver");
        } catch (Exception e) {
            throw new SQLException("Error in EzSqlj: Could not load the driver");
        }
        try {
            System.out.println("About to connect using url: " + url);
            Connection con0 = DriverManager.getConnection(url);
            con0.setAutoCommit(false); // Create a JDBC Connection
            con0.setAutoCommit(false); // set autocommit OFF
            ctx = new EzSqljCtx(con0);
        }
    }
```
EzSqljNameIter iter;
int count=0;

#sql [ctx] iter =
{SELECT LASTNAME FROM EMPLOYEE};  \[4b\] // Create result table of the SELECT
while (iter.next()) {
  System.out.println(iter.LASTNAME());  \[4c\] // Retrieve rows from result table
  count++;
}
System.out.println("Retrieved "+ count + " rows of data");
iter.close();  \[\] // Close the iterator
}
catch( SQLException e )  \[5\]
{
  System.out.println ("**** SELECT SQLException...");
  while(e!=null) {
    System.out.println ("Error msg: "+ e.getMessage());
    System.out.println ("SQLSTATE: "+ e.getSQLState());
    System.out.println ("Error code: "+ e.getErrorCode());
    e = e.getNextException();  // Check for chained exceptions
  }
}
catch( Exception e )
{
  System.out.println("**** NON-SQL exception =" + e);
  e.printStackTrace();
}
try
{
  #sql [ctx]
  {UPDATE DEPARTMENT SET MGRNO=:hvmgr
   WHERE DEPTNO=:hvdeptno};  \[\] // Update data for one department
  #sql [ctx] {COMMIT};  \[\] // Commit the update
}
catch( SQLException e )
{
  System.out.println ("**** UPDATE SQLException...");
  System.out.println ("Error msg: "+ e.getMessage() + ". SQLSTATE=" +
    e.getSQLState() + ". Error code=" + e.getErrorCode());
  e.printStackTrace();
}
catch( Exception e )
{
  System.out.println("**** NON-SQL exception =" + e);
  e.printStackTrace();
}
ctx.close();  \[\]  \[7\]
catch(SQLException e)
{
  System.out.println ("**** SQLException ...");
  System.out.println ("Error msg: "+ e.getMessage() + ". SQLSTATE=" +
    e.getSQLState() + ". Error code=" + e.getErrorCode());
  e.printStackTrace();
}
catch(Exception e)
{
  System.out.println("**** NON-SQL exception =" + e);
  e.printStackTrace();
}
}

Notes to Figure 24 on page 113
Connecting to a data source using SQLJ

In an SQLJ application, as in any other DB2 application, you must be connected to a data source before you can execute SQL statements.

You can use one of six techniques to connect to a data source in an SQLJ program. Two use the JDBC DriverManager interface, two use the JDBC DataSource interface, one uses a previously created connection context, and one uses the default connection.

SQLJ connection technique 1: JDBC DriverManager interface

SQLJ connection technique 1 uses the JDBC DriverManager interface as the underlying means for creating the connection.

To use SQLJ connection technique 1, follow these steps:

1. Execute an SQLJ connection declaration clause.
   Doing this generates a connection context class. The simplest form of the connection declaration clause is:
   
   ```sql
   #sql context context-class-name;
   ```
   The name of the generated connection context class is `context-class-name`.

2. Load a JDBC driver by invoking the Class.forName method.
   - For the IBM Data Server Driver for JDBC and SQLJ, invoke Class.forName this way:
     ```java
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```
     This step is unnecessary if you use the JDBC 4.0 driver.
For the DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows, which is deprecated, invoke Class.forName this way:

```java
Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
```

3. **Invoke the constructor for the connection context class that you created in step 1 on page 115.**

Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in one of the following forms:

```
connection-context-class connection-context-object =
    new connection-context-class(String url, boolean autocommit);
```

```
connection-context-class connection-context-object =
    new connection-context-class(String url, String user,
        String password, boolean autocommit);
```

```
connection-context-class connection-context-object =
    new connection-context-class(String url, Properties info,
        boolean autocommit);
```

The meanings of the parameters are:

- **url** A string that specifies the location name that is associated with the data source. That argument has one of the forms that are specified in "Connect to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ". The form depends on which JDBC driver you are using.

- **user and password** Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

- **info** Specifies an object of type java.util.Properties that contains a set of driver properties for the connection. For the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver), you should specify only the user and password properties. For the IBM Data Server Driver for JDBC and SQLJ, you can specify any of the properties listed in "Properties for the IBM Data Server Driver for JDBC and SQLJ".

- **autocommit** Specifies whether you want the database manager to issue a COMMIT after every statement. Possible values are true or false. If you specify false, you need to do explicit commit operations.

The following code uses connection technique 1 to create a connection to location NEWYORK. The connection requires a user ID and password, and does not require autocommit. The numbers to the right of selected statements correspond to the previously-described steps.
SQLJ connection technique 2: JDBC DriverManager interface

SQLJ connection technique 2 uses the JDBC DriverManager interface as the underlying means for creating the connection.

To use SQLJ connection technique 2, follow these steps:

1. Execute an SQLJ connection declaration clause.
   - Doing this generates a connection context class. The simplest form of the connection declaration clause is:
     ```
     #sql context context-class-name;
     ```
   - The name of the generated connection context class is `context-class-name`.

2. Load a JDBC driver by invoking the Class.forName method.
   - For the IBM Data Server Driver for JDBC and SQLJ, invoke Class.forName this way:
     ```java
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```
   - This step is unnecessary if you use the JDBC 4.0 driver.
   - For the DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows, which is deprecated, invoke Class.forName this way:
     ```java
     Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
     ```

3. Invoke the JDBC DriverManager.getConnection method.
   - Doing this creates a JDBC connection object for the connection to the data source. You can use any of the forms of getConnection that are specified in "Connect to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ".
   - The meanings of the `url`, `user`, and `password` parameters are:
     - `url` A string that specifies the location name that is associated with the data source. That argument has one of the forms that are specified in "Connect to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ". The form depends on which JDBC driver you are using.
user and password

Specify a user ID and password for connection to the data source, if the
data source to which you are connecting requires them.

4. Invoke the constructor for the connection context class that you created in step
   1 on page 117

Doing this creates a connection context object that you specify in each SQL
statement that you execute at the associated data source. The constructor
invocation statement needs to be in the following form:

```
connection-context-class connection-context-object =
   new connection-context-class (Connection JDBC-connection-object);
```

The JDBC-connection-object parameter is the Connection object that you created
in step 3 on page 117

The following code uses connection technique 2 to create a connection to location
NEWYORK. The connection requires a user ID and password, and does not require
autocommit. The numbers to the right of selected statements correspond to the
previously-described steps.

```
#sql context Ctx; // Create connection context class Ctx
String userid="dbadm"; // Declare variables for user ID and password
String password="dbadm";
String empname; // Declare a host variable
...
try {
   // Load the JDBC driver
   Class.forName("com.ibm.db2.jcc.DB2Driver");  
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Connection jdbccon=
   DriverManager.getConnection("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
   userid,password); // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx myConnCtx=new Ctx(jdbccon); // Create connection context object myConnCtx
   // for the connection to NEWYORK
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'}; // Use myConnCtx for executing an SQL statement
```

Figure 26. Using connection technique 2 to connect to a data source

**SQLJ connection technique 3: JDBC DataSource interface**

SQLJ connection technique 3 uses the JDBC DataSource as the underlying means
for creating the connection.

To use SQLJ connection technique 3, follow these steps:

1. Execute an SQLJ connection declaration clause.

   Doing this generates a connection context class. The simplest form of the
   connection declaration clause is:

   ```
   #sql context context-class-name;
   ```

   The name of the generated connection context class is context-class-name.

2. If your system administrator created a DataSource object in a different
   program, follow these steps. Otherwise, create a DataSource object and assign
   properties to it.

   a. Obtain the logical name of the data source to which you need to connect.
b. Create a context to use in the next step.

c. In your application program, use the Java Naming and Directory Interface (JNDI) to get the DataSource object that is associated with the logical data source name.

3. Invoke the JDBC DataSource.getConnection method.
   Doing this creates a JDBC connection object for the connection to the data source. You can use one of the following forms of getConnection:

   `getConnection();`
   `getConnection(user, password);`

   The meanings of the `user` and `password` parameters are:

   - `user` and `password`
     Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

4. If the default autocommit mode is not appropriate, invoke the JDBC Connection.setAutoCommit method.
   Doing this indicates whether you want the database manager to issue a COMMIT after every statement. The form of this method is:

   `setAutoCommit(boolean autocommit);`

5. Invoke the constructor for the connection context class that you created in step 1 on page 118
   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in the following form:

   `connection-context-class connection-context-object=
    new connection-context-class(Connection JDBC-connection-object);`

   The `JDBC-connection-object` parameter is the Connection object that you created in step 3.

The following code uses connection technique 3 to create a connection to a location with logical name `jdbc/sampledb`. This example assumes that the system administrator created and deployed a DataSource object that is available through JNDI lookup. The numbers to the right of selected statements correspond to the previously-described steps.

```java
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
#sql context CtxSqlj; // Create connection context class CtxSqlj 1
Context ctx=new InitialContext();
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
Connection con=ds.getConnection(); 2c
String empname; // Declare a host variable
...
con.setAutoCommit(false); // Do not autocommit
CtxSqlj myConnCtx=new CtxSqlj(con); 5
#sql [myConnCtx] [SELECT LASTNAME INTO :empname FROM EMPLOYEE WHERE EMPNO='000010'];
   // Use myConnCtx for executing an SQL statement
```

*Figure 27. Using connection technique 3 to connect to a data source*
**SQLJ connection technique 4: JDBC DataSource interface**

SQLJ connection technique 4 uses the JDBC DataSource as the underlying means for creating the connection. This technique **requires** that the DataSource is registered with JNDI.

To use SQLJ connection technique 4, follow these steps:

1. From your system administrator, obtain the logical name of the data source to which you need to connect.

2. Execute an SQLJ connection declaration clause.
   
   For this type of connection, the connection declaration clause needs to be of this form:
   
   ```sql
   #sql public static context context-class-name
   with (dataSource="logical-name");
   ```

   The connection context must be declared as public and static. *logical-name* is the data source name that you obtained in step 1.

3. Invoke the constructor for the connection context class that you created in step 2.
   
   Doing this creates a connection context object that you specify in each SQL statement that you execute at the associated data source. The constructor invocation statement needs to be in one of the following forms:

   ```java
   connection-context-class connection-context-object=
   new connection-context-class();
   ```

   ```java
   connection-context-class connection-context-object=
   new connection-context-class (String user,
   String password);
   ```

   The meanings of the *user* and *password* parameters are:

   **user and password**
   
   Specify a user ID and password for connection to the data source, if the data source to which you are connecting requires them.

The following code uses connection technique 4 to create a connection to a location with logical name `jdbc/sampledb`. The connection requires a user ID and password.

```sql
#sql public static context Ctx
   with (dataSource="jdbc/sampledb");  // Create connection context class Ctx
String userid="dbadm";  // Declare variables for user ID and password
String password="dbadm";

String empname;  // Declare a host variable
...
Ctx myConnCtx=new Ctx(userid, password);  // Create connection context object myConnCtx
   // for the connection to jdbc/sampledb
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};  // Use myConnCtx for executing an SQL statement
```

**Figure 28. Using connection technique 4 to connect to a data source**

**SQLJ connection technique 5: Use a previously created connection context**

SQLJ connection technique 5 uses a previously created connection context to connect to the data source.
In general, one program declares a connection context class, creates connection contexts, and passes them as parameters to other programs. A program that uses the connection context invokes a constructor with the passed connection context object as its argument.

Program CtxGen.sqlj declares connection context Ctx and creates instance oldCtx:

```sql
#sql context Ctx;
```

```java
// Create connection context object oldCtx
```

Program test.sqlj receives oldCtx as a parameter and uses oldCtx as the argument of its connection context constructor:

```java
void useContext(sqlj.runtime.ConnectionContext oldCtx)
```

```java
// oldCtx was created in CtxGen.sqlj
{
    Ctx myConnCtx=
        new Ctx(oldCtx); // Create connection context object myConnCtx
    // from oldCtx
    #sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
                      WHERE EMPNO='000010';} // Use myConnCtx for executing an SQL statement
    ...
}
```

**SQLJ connection technique 6: Use the default connection**

SQLJ connection technique 6 uses the default connection to connect to the data source. It should be used only in situations where the database thread is controlled by another resource manager, such as the Java stored procedure environment.

You use the default connection by specifying your SQL statements without a connection context object. When you use this technique, you do not need to load a JDBC driver unless you explicitly use JDBC interfaces in your program.

The default connection context can be:

- The connection context that is associated with the data source that is bound to the logical name jdbc/defaultDataSource
- An explicitly created connection context that has been set as the default connection context with the ConnectionContext.setDefaultContext method. This method of creating a default connection context is not recommended.

The following SQLJ execution clause does not have a connection context, so it uses the default connection context.

```sql
#sql {SELECT LASTNAME INTO :empname FROM EMPLOYEE
      WHERE EMPNO='000010';} // Use default connection for executing an SQL statement
```

**Java packages for SQLJ support**

Before you can execute SQLJ statements or invoke JDBC methods in your SQLJ program, you need to be able to access all or parts of various Java packages that contain support for those statements.

You can do that either by importing the packages or specific classes, or by using fully-qualified class names. You might need the following packages or classes for your SQLJ program:
Variables in SQLJ applications

In DB2 programs in other languages, you use host variables to pass data between the application program and DB2. In SQLJ programs, you can use host variables or host expressions.

A host expression begins with a colon (:). The colon is followed by an optional parameter mode identifier (IN, OUT, or INOUT), which is followed by a parenthesized expression clause.

Host variables and host expressions are case sensitive.

A complex expression is an array element or Java expression that evaluates to a single value. A complex expression in an SQLJ clause must be surrounded by parentheses.

The following examples demonstrate how to use host expressions.

Example: Declaring a Java identifier and using it in a SELECT statement:

In this example, the statement that begins with #sql has the same function as a SELECT statement in other languages. This statement assigns the last name of the employee with employee number 000010 to Java identifier empname.

```java
String empname;
...
#sql [ctxt]
{SELECT LASTNAME INTO :empname FROM EMPLOYEE WHERE EMPNO='000010'};
```

Example: Declaring a Java identifier and using it in a stored procedure call:

In this example, the statement that begins with #sql has the same function as an SQL CALL statement in other languages. This statement uses Java identifier empno as an input parameter to stored procedure A. The keyword IN, which precedes empno, specifies that empno is an input parameter. For a parameter in a CALL statement, IN is the default. The explicit or default qualifier that indicates how the parameter is used (IN, OUT, or INOUT) must match the corresponding value in the parameter definition that you specified in the CREATE PROCEDURE statement for the stored procedure.

```java
String empno = "0000010";
...
#sql [ctxt] {CALL A (:IN empno)};
```
Example: Using a complex expression as a host identifier:

This example uses complex expression (((int)yearsEmployed++/5)*500) as a host expression.

```sql
#sql [ctxt] (UPDATE EMPLOYEE
   SET BONUS=:(((int)yearsEmployed++/5)*500) WHERE EMPNO=:empID);
```

SQLJ performs the following actions when it processes a complex host expression:

- Evaluates each of the host expressions in the statement, from left to right, before assigning their respective values to the database.
- Evaluates side effects, such as operations with postfix operators, according to normal Java rules. All host expressions are fully evaluated before any of their values are passed to DB2.
- Uses Java rules for rounding and truncation.

Therefore, if the value of yearsEmployed is 6 before the UPDATE statement is executed, the value that is assigned to column BONUS by the UPDATE statement is ((int)6/5)*500, or 500. After 500 is assigned to BONUS, the value of yearsEmployed is incremented.

Restrictions on variable names: Two strings have special meanings in SQLJ programs. Observe the following restrictions when you use these strings in your SQLJ programs:

- The string __sJT_ is a reserved prefix for variable names that are generated by SQLJ. Do not begin the following types of names with __sJT_:  
  - Host expression names
  - Java variable names that are declared in blocks that include executable SQL statements
  - Names of parameters for methods that contain executable SQL statements
  - Names of fields in classes that contain executable SQL statements, or in classes with subclasses or enclosed classes that contain executable SQL statements

- The string _SJ is a reserved suffix for resource files and classes that are generated by SQLJ. Avoid using the string _SJ in class names and input source file names.

Comments in an SQLJ application

To document your SQLJ program, you need to include comments. To do that, use Java comments. Java comments are denoted by /* */ or //.

You can include Java comments outside SQLJ clauses, wherever the Java language permits them. Within an SQLJ clause, you can use Java comments in the following places:

- Within a host expression (/* */ or //).
- Within an SQL statement in an executable clause, if the data source supports a comment within the SQL statement (/* */ or --).

/* and */ pairs in an SQL statement can be nested.
SQL statement execution in SQLJ applications

You execute SQL statements in a traditional SQL program to create tables, update data in tables, retrieve data from the tables, call stored procedures, or commit or rollback transactions. In an SQLJ program, you also execute these statements, within SQLJ executable clauses.

An executable clause can have one of the following general forms:

```
#sql [connection-context] {sql-statement};
#sql [connection-context,execution-context] {sql-statement};
#sql [execution-context] {sql-statement};
```

execution-context specification
In an executable clause, you should always specify an explicit connection context, with one exception: you do not specify an explicit connection context for a FETCH statement. You include an execution context only for specific cases. See "Control the execution of SQL statements in SQLJ" for information about when you need an execution context.

connection-context specification
In an executable clause, if you do not explicitly specify a connection context, the executable clause uses the default connection context.

Creating and modifying DB2 objects in an SQLJ application

Use SQLJ executable clauses to execute data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE) or to execute INSERT, searched or positioned UPDATE, and searched or positioned DELETE statements.

The following executable statements demonstrate an INSERT, a searched UPDATE, and a searched DELETE:

```
#sql [myConnCtx] {INSERT INTO DEPARTMENT VALUES
("X00","Operations 2","000030","E01",NULL)};
#sql [myConnCtx] {UPDATE DEPARTMENT
SET MGRNO="000090" WHERE MGRNO="000030"};
#sql [myConnCtx] {DELETE FROM DEPARTMENT
WHERE DEPTNO="X00"};
```

Performing positioned UPDATE and DELETE operations in an SQLJ application

As in DB2 applications in other languages, performing positioned UPDATEs and DELETEs with SQLJ is an extension of retrieving rows from a result table.

The basic steps are:
1. Declare the iterator.
   The iterator can be positioned or named. For positioned UPDATE or DELETE operations, declare the iterator as updatable, using one or both of the following clauses:

   implements sqlj.runtime.ForUpdate
   This clause causes the generated iterator class to include methods for using updatable iterators. This clause is required for programs with positioned UPDATE or DELETE operations.

   with (updateColumns="column-list")
   This clause specifies a comma-separated list of the columns of the result table that the iterator will update. This clause is optional.
You need to declare the iterator as public, so you need to follow the rules for declaring and using public iterators in the same file or different files.

If you declare the iterator in a file by itself, any SQLJ source file that has addressability to the iterator and imports the generated class can retrieve data and execute positioned UPDATE or DELETE statements using the iterator.

The authorization ID under which a positioned UPDATE or DELETE statement executes depends on whether the statement executes statically or dynamically. If the statement executes statically, the authorization ID is the owner of the plan or package that includes the statement. If the statement executes dynamically the authorization ID is determined by the DYNAMICRULES behavior that is in effect. For the IBM Data Server Driver for JDBC and SQLJ, the behavior is always DYNAMICRULES BIND.

2. Disable autocommit mode for the connection.
   If autocommit mode is enabled, a COMMIT operation occurs every time the positioned UPDATE statement executes, which causes the iterator to be destroyed unless the iterator has the with (holdability=true) attribute. Therefore, you need to turn autocommit off to prevent COMMIT operations until you have finished using the iterator. If you want a COMMIT to occur after every update operation, an alternative way to keep the iterator from being destroyed after each COMMIT operation is to declare the iterator with (holdability=true).

3. Create an instance of the iterator class.
   This is the same step as for a non-updatable iterator.

4. Assign the result table of a SELECT to an instance of the iterator.
   This is the same step as for a non-updatable iterator. The SELECT statement must not include a FOR UPDATE clause.

5. Retrieve and update rows.
   For a positioned iterator, do this by performing the following actions in a loop:
   a. Execute a FETCH statement in an executable clause to obtain the current row.
   b. Test whether the iterator is pointing to a row of the result table by invoking the PositionedIterator.endFetch method.
   c. If the iterator is pointing to a row of the result table, execute an SQL UPDATE... WHERE CURRENT OF :iterator-object statement in an executable clause to update the columns in the current row. Execute an SQL DELETE... WHERE CURRENT OF :iterator-object statement in an executable clause to delete the current row.

   For a named iterator, do this by performing the following actions in a loop:
   a. Invoke the next method to move the iterator forward.
   b. Test whether the iterator is pointing to a row of the result table by checking whether next returns true.
   c. Execute an SQL UPDATE... WHERE CURRENT OF iterator-object statement in an executable clause to update the columns in the current row. Execute an SQL DELETE... WHERE CURRENT OF iterator-object statement in an executable clause to delete the current row.

6. Close the iterator.
   Use the close method to do this.

The following code shows how to declare a positioned iterator and use it for positioned UPDATEs. The numbers to the right of selected statements correspond to the previously described steps.
First, in one file, declare positioned iterator `UpdByPos`, specifying that you want to use the iterator to update column `SALARY`:

```java
import java.math.*; // Import this class for BigDecimal data type
#sql public iterator UpdByPos implements sqlj.runtime.ForUpdate with(updateColumns="SALARY") (String, BigDecimal);
```

**Figure 29. Example of declaring a positioned iterator for a positioned UPDATE**

Then, in another file, use `UpdByPos` for a positioned UPDATE, as shown in the following code fragment:

```java
import sqlj.runtime.*; // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*; // Import this class for BigDecimal data type
import UpdByPos; // Import the generated iterator class that
// was created by the iterator declaration clause
// for UpdByName in another file

#sql context HSCtx; // Create a context class HSCtx
public static void main (String args[]) {  
try {  
Class.forName("com.ibm.db2.jcc.DB2Driver");
} catch (ClassNotFoundException e) {  
e.printStackTrace();
}
Connection HSjdbccon=
DriverManager.getConnection("jdbc:db2:SANJOSE");  
// Create a JDBC connection object
HSjdbccon.setAutoCommit(false);  
// Set autocommit off so automatic commits
// do not destroy the cursor between updates
HSCtx myConnCtx=new HSCtx(HSjdbccon);  
// Create a connection context object
UpdByPos upditer; // Declare iterator object of UpdByPos class
String empnum; // Declares host variable to receive EMPNO
BigDecimal sal; // and SALARY column values

#sql [myConnCtx]
upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
WHERE WORKDEPT='D11'};
// Assign result table to iterator object
#sql {FETCH :upditer INTO :empnum,:sal};  
// Move cursor to next row
while (!upditer.endFetch()) {  
// Check if on a row
{
#sql [myConnCtx] {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
WHERE CURRENT OF :upditer};  
// Perform positioned update
System.out.println("Updating row for " + empnum);
#sql {FETCH :upditer INTO :empnum,:sal};  
// Move cursor to next row
}
}
upditer.close(); // Close the iterator
#sql [myConnCtx] {COMMIT};  
// Commit the changes
myConnCtx.close(); // Close the connection context
}
```

**Figure 30. Example of performing a positioned UPDATE with a positioned iterator**

The following code shows how to declare a named iterator and use it for positioned UPDATEs. The numbers to the right of selected statements correspond to the previously described steps.
First, in one file, declare named iterator `UpdByName`, specifying that you want to use the iterator to update column `SALARY`:

```java
import java.math.*; // Import this class for BigDecimal data type
#sql public iterator UpdByName implements sqlj.runtime.ForUpdate
    with(updateColumns="SALARY") (String EmpNo, BigDecimal Salary);
```

*Figure 31. Example of declaring a named iterator for a positioned UPDATE*

Then, in another file, use `UpdByName` for a positioned UPDATE, as shown in the following code fragment:

```java
import sqlj.runtime.*; // Import files for SQLJ and JDBC APIs
import java.sql.*; // Import the generated iterator class that
import java.math.*; // was created by the iterator declaration clause
import UpdByName; // for UpdByName in another file
#sql context HSCtx; // Create a connection context class HSCtx
public static void main (String args[])
{
    try {
        Class.forName("com.ibm.db2.jcc.DB2Driver");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    Connection HSjdbccon=
        DriverManager.getConnection("jdbc:db2:SANJOSE");
    // Create a JDBC connection object
    HSjdbccon.setAutoCommit(false);
    // Set autocommit off so automatic commits do not destroy the cursor between updates
    HSCtx myConnCtx=new HSCtx(HSjdbccon);
    // Create a connection context object
    UpdByName upditer;
    // Declare iterator object of UpdByName class
    String empnum;
    // Declare host variable to receive EmpNo column values
    #sql [myConnCtx]
        upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE
            WHERE WORKDEPT='D11'};
    // Assign result table to iterator object
    while (upditer.next())
    { // Move cursor to next row and check if on a row
        empnum = upditer.EmpNo(); // Get employee number from current row
        #sql [myConnCtx]
            {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
                WHERE CURRENT OF :upditer};
        // Perform positioned update
        System.out.println("Updating row for " + empnum);
    }
    upditer.close(); // Close the iterator
    #sql [myConnCtx] {COMMIT}; // Commit the changes
    myConnCtx.close(); // Close the connection context
}
```

*Figure 32. Example of performing a positioned UPDATE with a named iterator*

**Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application**

SQLJ allows iterators to be passed between methods as variables.
An iterator that is used for a positioned UPDATE or DELETE statement can be identified only at runtime. The same SQLJ positioned UPDATE or DELETE statement can be used with different iterators at runtime. If you specify a value of YES for -staticpositioned when you customize your SQLJ application as part of the program preparation process, the SQLJ customizer prepares positioned UPDATE or DELETE statements to execute statically. In this case, the customizer must determine which iterators belong with which positioned UPDATE or DELETE statements. The SQLJ customizer does this by matching iterator data types to data types in the UPDATE or DELETE statements. However, if there is not a unique mapping of tables in UPDATE or DELETE statements to iterator classes, the SQLJ customizer cannot determine exactly which iterators and UPDATE or DELETE statements go together. The SQLJ customizer must arbitrarily pair iterators with UPDATE or DELETE statements, which can sometimes result in SQL errors. The following code fragments illustrate this point.

```
#sql iterator GeneralIter implements sqlj.runtime.ForUpdate
    ( String );

public static void main ( String args[] )
{
    ...
    GeneralIter iter1 = null;
    #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };
    GeneralIter iter2 = null;
    #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };
    ...
    doUpdate ( iter1 );
}

public static void doUpdate ( GeneralIter iter )
{
    #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
}
```

Figure 33. Static positioned UPDATE that fails

In this example, only one iterator is declared. Two instances of that iterator are declared, and each is associated with a different SELECT statement that retrieves data from a different table. During customization and binding with -staticpositioned YES, SQLJ creates two DECLARE CURSOR statements, one for each SELECT statement, and attempts to bind an UPDATE statement for each cursor. However, the bind process fails with SQLCODE -509 when UPDATE TABLE1 ...
... WHERE CURRENT OF :iter is bound for the cursor for SELECT CHAR_COL2 FROM TABLE2 because the table for the UPDATE does not match the table for the cursor.

You can avoid a bind time error for a program like the one in Figure 33 by specifying the bind option SQLERROR(CONTINUE). However, this technique has the drawback that it causes the DB2 database manager to build a package, regardless of the SQL errors that are in the program. A better technique is to write the program so that there is a one-to-one mapping between tables in positioned UPDATE or DELETE statements and iterator classes. Figure 34 on page 129 shows an example of how to do this.
With this method of coding, each iterator class is associated with only one table. Therefore, the DB2 bind process can always associate the positioned UPDATE statement with a valid iterator.

Making batch updates in SQLJ applications

The IBM Data Server Driver for JDBC and SQLJ supports batch updates in SQLJ. With batch updates, instead of updating rows of a table one at a time, you can direct SQLJ to execute a group of updates at the same time.

You can include the following types of statements in a batch update:
- Searched INSERT, UPDATE, or DELETE, or MERGE statements
- CREATE, ALTER, DROP, GRANT, or REVOKE statements
- CALL statements with input parameters only

Unlike JDBC, SQLJ allows heterogeneous batches that contain statements with input parameters or host expressions. You can therefore combine any of the following items in an SQLJ batch:
- Instances of the same statement
- Different statements
- Statements with different numbers of input parameters or host expressions
- Statements with different data types for input parameters or host expressions
- Statements with no input parameters or host expressions

For all cases except homogeneous batches of INSERT statements, when an error occurs during execution of a statement in a batch, the remaining statements are executed, and a BatchUpdateException is thrown after all the statements in the batch have executed.

For homogeneous batches of INSERT statements, the behavior is as follows:

Figure 34. Static positioned UPDATE that succeeds
• If you set atomicMultiRowInsert to DB2BaseDataSource.YES (1) when you run db2sqljcustomize, and the target data server is DB2 for z/OS, when an error occurs during execution of an INSERT statement in a batch, the remaining statements are not executed, and a BatchUpdateException is thrown.

• If you do not set atomicMultiRowInsert to DB2BaseDataSource.YES (1) when you run db2sqljcustomize, or the target data server is not DB2 for z/OS, when an error occurs during execution of an INSERT statement in a batch, the remaining statements are executed, and a BatchUpdateException is thrown after all the statements in the batch have executed.

To obtain information about warnings, use the ExecutionContext.getWarnings method on the ExecutionContext that you used to submit statements to be batched. You can then retrieve an error description, SQLSTATE, and error code for each SQLWarning object.

When a batch is executed implicitly because the program contains a statement that cannot be added to the batch, the batch is executed before the new statement is processed. If an error occurs during execution of the batch, the statement that caused the batch to execute does not execute.

The basic steps for creating, executing, and deleting a batch of statements are:

1. Disable AutoCommit for the connection.
   Do this so that you can control whether to commit changes to already-executed statements when an error occurs during batch execution.

2. Acquire an execution context.
   All statements that execute in a batch must use this execution context.

3. Invoke the ExecutionContext.setBatching(true) method to create a batch.
   Subsequent batchable statements that are associated with the execution context that you created in step 2 are added to the batch for later execution.

   If you want to batch sets of statements that are not batch compatible in parallel, you need to create an execution context for each set of batch compatible statements.

4. Include SQLJ executable clauses for SQL statements that you want to batch.
   These clauses must include the execution context that you created in step 2.

   If an SQLJ executable clause has input parameters or host expressions, you can include the statement in the batch multiple times with different values for the input parameters or host expressions.

   To determine whether a statement was added to an existing batch, was the first statement in a new batch, or was executed inside or outside a batch, invoke the ExecutionContext.getUpdateCount method. This method returns one of the following values:

   **ExecutionContext.ADD_BATCH_COUNT**
   This is a constant that is returned if the statement was added to an existing batch.

   **ExecutionContext.NEW_BATCH_COUNT**
   This is a constant that is returned if the statement was the first statement in a new batch.

   **ExecutionContext.EXEC_BATCH_COUNT**
   This is a constant that is returned if the statement was part of a batch, and the batch was executed.
Other integer
This value is the number of rows that were updated by the statement. This value is returned if the statement was executed rather than added to a batch.

5. Execute the batch explicitly or implicitly.
   • Invoke the ExecutionContext.executeBatch method to execute the batch explicitly.
     executeBatch returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch.
   • Alternatively, a batch executes implicitly under the following circumstances:
     – You include a batchable statement in your program that is not compatible with statements that are already in the batch. In this case, SQLJ executes the statements that are already in the batch and creates a new batch that includes the incompatible statement.
     – You include a statement in your program that is not batchable. In this case, SQLJ executes the statements that are already in the batch. SQLJ also executes the statement that is not batchable.
     – After you invoke the ExecutionContext.setBatchLimit(n) method, you add a statement to the batch that brings the number of statements in the batch to n or greater. n can have one of the following values:

   ExecutionContext.UNLIMITED_BATCH
   This constant indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking setBatchLimit.

   ExecutionContext.AUTO_BATCH
   This constant indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

Positive integer
When this number of statements have been added to the batch, SQLJ executes the batch implicitly. However, the batch might be executed before this many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

To determine the number of rows that were updated by a batch that was executed implicitly, invoke the ExecutionContext.getBatchUpdateCounts method. getBatchUpdateCounts returns an integer array that contains the number of rows that were updated by each statement in the batch. The order of the elements in the array corresponds to the order in which you added statements to the batch. Each array element can be one of the following values:

-2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.

-3 This value indicates that the SQL statement failed.

Other integer
This value is the number of rows that were updated by the statement.

6. Optionally, when all statements have been added to the batch, disable batching. Do this by invoking the ExecutionContext.setBatching(false) method. When you disable batching, you can still execute the batch implicitly or explicitly, but no
more statements are added to the batch. Disabling batching is useful when a batch already exists, and you want to execute a batch compatible statement, rather than adding it to the batch.

If you want to clear a batch without executing it, invoke the ExecutionContext.cancel method.

7. If batch execution was implicit, perform a final, explicit executeBatch to ensure that all statements have been executed.

Example

The following example demonstrates batching of UPDATE statements. The numbers to the right of selected statements correspond to the previously described steps.

```java
#sql iterator GetMgr(String); // Declare positioned iterator
...
GetMgr deptiter; // Declare object of GetMgr class
String mgrnum = null; // Declare host variable for manager number
int raise = 400; // Declare raise amount
int currentSalary; // Declare current salary
String url, username, password; // Declare url, user ID, password
...
TestContext c1 = new TestContext (url, username, password, false);
ExecutionContext ec = new ExecutionContext();
ec.setBatching(true);

#sql [c1] deptiter =
  {SELECT MGRNO FROM DEPARTMENT}; // Assign the result table of the SELECT to iterator object deptiter
#sql {FETCH :deptiter INTO :mgrnum}; // Retrieve the first manager number
while (!deptiter.endFetch()) { // Check whether the FETCH returned a row
  #sql [c1]
    {SELECT SALARY INTO :currentSalary FROM EMPLOYEE
     WHERE EMPNO=:mgrnum};
  #sql [c1, ec]
    {UPDATE EMPLOYEE SET SALARY=:(currentSalary+raise)
     WHERE EMPNO=:mgrnum};
  #sql {FETCH :deptiter INTO :mgrnum }; // Fetch the next row
}
ec.executeBatch();
ec.setBatching(false);
#sql [c1] {COMMIT};
depiter.close(); // Close the iterator
c1.close(); // Close the connection
}
```

The following example demonstrates batching of INSERT statements. Suppose that ATOMICTBL is defined like this:

CREATE TABLE ATOMICTBL(
  INTCOL INTEGER NOT NULL UNIQUE,
  CHARCOL VARCHAR(10))

Also suppose that the table already has a row with the values 2 and "val2". Because of the uniqueness constraint on INTCOL, when the following code is executed, the second INSERT statement in the batch fails.

If the target data server is DB2 for z/OS, and this application is customized without atomicMultiRowInsert set to DB2BaseDataSource.YES, the batch INSERT is
non-atomic, so the first set of values is inserted in the table. However, if the
application is customized with atomicMultiRowInsert set to
DB2BaseDataSource.YES, the batch INSERT is atomic, so the first set of values is
not inserted.

The numbers to the right of selected statements correspond to the previously
described steps.

```java
...  
TestContext ctx = new TestContext (url, username, password, false);  1
ctx.getExecutionContext().setBatching(true);  2,3
try {
  for (int i = 1; i <= 2 ; ++i) {
    if (i == 1) {
      intVar = 3;
      strVar = "val1";
    }
    if (i == 2) {
      intVar = 1;
      strVar = "val2";
    }
    #sql [ctx] {INSERT INTO ATOMICTBL values(:intVar, :strVar)};  4
  }
  int[] counts = ctx.getExecutionContext().executeBatch();  5
  for (int i = 0; i<counts.length; ++i) {
    System.out.println(" count[" + i + "]: " + counts[i]);
  }
}
catch (SQLException e) {
  System.out.println(" Exception Caught: " + e.getMessage());
  SQLException excp = null;
  if (e instanceof SQLException) {
    System.out.println(" SQLCode: " + ((SQLException)e).getErrorCode() + "
      Message: " + e.getMessage());
    excp = ((SQLException)e).getNextException();
    while ( excp != null ) {
      System.out.println(" SQLCode: " + ((SQLException)excp).getErrorCode() + 
        " Message: " + excp.getMessage());
      excp = excp.getNextException();
    }
  }
}
```

### Data retrieval in SQLJ applications

SQLJ applications use a result set iterator to retrieve result sets. Like a cursor, a
result set iterator can be non-scrollable or scrollable.

Just as in DB2 applications in other languages, if you want to retrieve a single row
from a table in an SQLJ application, you can write a SELECT INTO statement with
a WHERE clause that defines a result table that contains only that row:

```sql
#sql [myConnCtx] {SELECT DEPTNO INTO :hvdeptno
  FROM DEPARTMENT WHERE DEPTNAME="OPERATIONS"};
```

However, most SELECT statements that you use create result tables that contain
many rows. In DB2 applications in other languages, you use a cursor to select the
individual rows from the result table. That cursor can be non-scrollable, which
means that when you use it to fetch rows, you move the cursor serially, from the
beginning of the result table to the end. Alternatively, the cursor can be scrollable,
which means that when you use it to fetch rows, you can move the cursor
forward, backward, or to any row in the result table.
This topic discusses how to use non-scrollable iterators. For information on using scrollable iterators, see "Use scrollable iterators in an SQLJ application".

A result set iterator is a Java object that you use to retrieve rows from a result table. Unlike a cursor, a result set iterator can be passed as a parameter to a method.

The basic steps in using a result set iterator are:
1. Declare the iterator, which results in an iterator class
2. Define an instance of the iterator class.
3. Assign the result table of a SELECT to an instance of the iterator.
4. Retrieve rows.
5. Close the iterator.

There are two types of iterators: positioned iterators and named iterators. Positioned iterators extend the interface sqlj.runtime.PositionedIterator. Positioned iterators identify the columns of a result table by their position in the result table. Named iterators extend the interface sqlj.runtime.NamedIterator. Named iterators identify the columns of the result table by result table column names.

**Using a named iterator in an SQLJ application**
Use a named iterator to refer to each of the columns in a result table by name.

The steps in using a named iterator are:
1. Declare the iterator.
   
   You declare any result set iterator using an *iterator declaration clause*. This causes an iterator class to be created that has the same name as the iterator. For a named iterator, the iterator declaration clause specifies the following information:
   
   • The name of the iterator
   • A list of column names and Java data types
   • Information for a Java class declaration, such as whether the iterator is public or static
   • A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

   When you declare a named iterator for a query, you specify names for each of the iterator columns. Those names must match the names of columns in the result table for the query. An iterator column name and a result table column name that differ only in case are considered to be matching names. The named iterator class that results from the iterator declaration clause contains *accessor methods*. There is one accessor method for each column of the iterator. Each accessor method name is the same as the corresponding iterator column name. You use the accessor methods to retrieve data from columns of the result table.

   You need to specify Java data types in the iterators that closely match the corresponding DB2 column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and DB2 data types.

   You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a *with-clause* must be declared as public. Therefore, if an iterator needs to be public, it can be declared only where a public class is allowed. The following list describes some alternative methods of declaring an iterator:
   
   • As public, in a source file by itself
This method lets you use the iterator declaration in other code modules, and provides an iterator that works for all SQLJ applications. In addition, there are no concerns about having other top-level classes or public classes in the same source file.

- As a top-level class in a source file that contains other top-level class definitions
  Java allows only one public, top-level class in a code module. Therefore, if you need to declare the iterator as public, such as when the iterator includes a with-clause, no other classes in the code module can be declared as public.

- As a nested static class within another class
  Using this alternative lets you combine the iterator declaration with other class declarations in the same source file, declare the iterator and other classes as public, and make the iterator class visible to other code modules or packages. However, when you reference the iterator from outside the nesting class, you must fully-qualify the iterator name with the name of the nesting class.

- As an inner class within another class
  When you declare an iterator in this way, you can instantiate it only within an instance of the nesting class. However, you can declare the iterator and other classes in the file as public.
  You cannot cast a JDBC ResultSet to an iterator if the iterator is declared as an inner class. This restriction does not apply to an iterator that is declared as a static nested class. See "Use SQLJ and JDBC in the same application" for more information on casting a ResultSet to a iterator.

2. Create an instance of the iterator class.
   You declare an object of the named iterator class to retrieve rows from a result table.

3. Assign the result table of a SELECT to an instance of the iterator.
   To assign the result table of a SELECT to an iterator, you use an SQLJ assignment clause. The format of the assignment clause for a named iterator is:
   
   ```sql
   #sql context-clause iterator-object={select-statement};
   ```

   See "SQLJ assignment-clause" and "SQLJ context-clause" for more information.

4. Retrieve rows.
   Do this by invoking accessor methods in a loop. Accessor methods have the same names as the corresponding columns in the iterator, and have no parameters. An accessor method returns the value from the corresponding column of the current row in the result table. Use the NamedIterator.next() method to move the cursor forward through the result table.
   To test whether you have retrieved all rows, check the value that is returned when you invoke the next method. next returns a boolean with a value of false if there is no next row.

5. Close the iterator.
   Use the NamedIterator.close method to do this.

The following code demonstrates how to declare and use a named iterator. The numbers to the right of selected statements correspond to the previously-described steps.
Using a positioned iterator in an SQLJ application

Use a positioned iterator to refer to columns in a result table by their position in the result set.

The steps in using a positioned iterator are:

1. Declare the iterator.

   You declare any result set iterator using an iterator declaration clause. This causes an iterator class to be created that has the same name and attributes as the iterator. For a positioned iterator, the iterator declaration clause specifies the following information:
   - The name of the iterator
   - A list of Java data types
   - Information for a Java class declaration, such as whether the iterator is public or static
   - A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

   The data type declarations represent columns in the result table and are referred to as columns of the result set iterator. The columns of the result set iterator correspond to the columns of the result table, in left-to-right order. For example, if an iterator declaration clause has two data type declarations, the first data type declaration corresponds to the first column in the result table, and the second data type declaration corresponds to the second column in the result table.

   You need to specify Java data types in the iterators that closely match the corresponding DB2 column data types. See “Java, JDBC, and SQL data types” for a list of the best mappings between Java data types and DB2 data types.

2. You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a with-clause must be declared as public. Therefore, if an iterator needs to be public, it can be declared only where a public class is allowed. The following list describes some alternative methods of declaring an iterator:
   - As public, in a source file by itself

---

Figure 35. Example of using a named iterator

```sql
#sql iterator ByName(String LastName, Date HireDate);

ByName nameiter; // Declare object of ByName class
#sql [ctxt]
nameiter=(SELECT LASTNAME, HIREDATE FROM EMPLOYEE);

while (nameiter.next()) // Move the iterator through the result table and test whether all rows retrieved
{
    System.out.println( nameiter.LastName() + " was hired on " + nameiter.HireDate()); // Use accessor methods LastName and HireDate to retrieve column values
}
nameiter.close(); // Close the iterator
```

---

Using a positioned iterator in an SQLJ application

Use a positioned iterator to refer to columns in a result table by their position in the result set.

The steps in using a positioned iterator are:

1. Declare the iterator.

   You declare any result set iterator using an iterator declaration clause. This causes an iterator class to be created that has the same name and attributes as the iterator. For a positioned iterator, the iterator declaration clause specifies the following information:
   - The name of the iterator
   - A list of Java data types
   - Information for a Java class declaration, such as whether the iterator is public or static
   - A set of attributes, such as whether the iterator is holdable, or whether its columns can be updated

   The data type declarations represent columns in the result table and are referred to as columns of the result set iterator. The columns of the result set iterator correspond to the columns of the result table, in left-to-right order. For example, if an iterator declaration clause has two data type declarations, the first data type declaration corresponds to the first column in the result table, and the second data type declaration corresponds to the second column in the result table.

   You need to specify Java data types in the iterators that closely match the corresponding DB2 column data types. See "Java, JDBC, and SQL data types" for a list of the best mappings between Java data types and DB2 data types.

   You can declare an iterator in a number of ways. However, because a Java class underlies each iterator, you need to ensure that when you declare an iterator, the underlying class obeys Java rules. For example, iterators that contain a with-clause must be declared as public. Therefore, if an iterator needs to be public, it can be declared only where a public class is allowed. The following list describes some alternative methods of declaring an iterator:
   - As public, in a source file by itself
This is the most versatile method of declaring an iterator. This method lets you use the iterator declaration in other code modules, and provides an iterator that works for all SQLJ applications. In addition, there are no concerns about having other top-level classes or public classes in the same source file.

- As a top-level class in a source file that contains other top-level class definitions
  Java allows only one public, top-level class in a code module. Therefore, if you need to declare the iterator as public, such as when the iterator includes a with-clause, no other classes in the code module can be declared as public.

- As a nested static class within another class
  Using this alternative lets you combine the iterator declaration with other class declarations in the same source file, declare the iterator and other classes as public, and make the iterator class visible from other code modules or packages. However, when you reference the iterator from outside the nesting class, you must fully-qualify the iterator name with the name of the nesting class.

- As an inner class within another class
  When you declare an iterator in this way, you can instantiate it only within an instance of the nesting class. However, you can declare the iterator and other classes in the file as public.
  You cannot cast a JDBC ResultSet to an iterator if the iterator is declared as an inner class. This restriction does not apply to an iterator that is declared as a static nested class. See "Use SQLJ and JDBC in the same application" for more information on casting a ResultSet to a iterator.

2. Create an instance of the iterator class.
   You declare an object of the positioned iterator class to retrieve rows from a result table.

3. Assign the result table of a SELECT to an instance of the iterator.
   To assign the result table of a SELECT to an iterator, you use an SQLJ assignment clause. The format of the assignment clause for a positioned iterator is:
   
   ```
   #sql
   context-clause iterator-object={select-statement};
   ```

4. Retrieve rows.
   Do this by executing FETCH statements in executable clauses in a loop. The FETCH statements looks the same as a FETCH statements in other languages.
   To test whether you have retrieved all rows, invoke the PositionedIterator.endFetch method after each FETCH. endFetch returns a boolean with the value true if the FETCH failed because there are no rows to retrieve.

5. Close the iterator.
   Use the PositionedIterator.close method to do this.

The following code demonstrates how to declare and use a positioned iterator. The numbers to the right of selected statements correspond to the previously-described steps.
Multiple open iterators for the same SQL statement in an SQLJ application

With the IBM Data Server Driver for JDBC and SQLJ, your application can have multiple concurrently open iterators for a single SQL statement in an SQLJ application. With this capability, you can perform one operation on a table using one iterator while you perform a different operation on the same table using another iterator.

When you use concurrently open iterators in an application, you should close iterators when you no longer need them to prevent excessive storage consumption in the Java heap.

The following examples demonstrate how to perform the same operations on a table without concurrently open iterators on a single SQL statement and with concurrently open iterators on a single SQL statement. These examples use the following iterator declaration:

```java
import java.math.*;
#sql public iterator MultiIter(String EmpNo, BigDecimal Salary);
```

Without the capability for multiple, concurrently open iterators for a single SQL statement, if you want to select employee and salary values for a specific employee number, you need to define a different SQL statement for each employee number, as shown in Figure 37 on page 139.
Figure 38 demonstrates how you can perform the same operations when you have the capability for multiple, concurrently open iterators for a single SQL statement.

Multiple open instances of an iterator in an SQLJ application

Multiple instances of an iterator can be open concurrently in a single SQLJ application. One application for this ability is to open several instances of an iterator that uses host expressions. Each instance can use a different set of host expression values.

The following example shows an application with two concurrently open instances of an iterator.

```java
MultiIter iter1 = null; // Iterator instance for retrieving
// data for first employee
String EmpNo1 = "000100"; // Employee number for first employee
#sql [ctx] iter1 =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo1}; // Assign result table to first iterator
MultiIter iter2 = null; // Iterator instance for retrieving
// data for second employee
String EmpNo2 = "000200"; // Employee number for second employee
#sql [ctx] iter2 =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo2}; // Assign result table to second iterator

// Process with iter1
// Process with iter2
iter1.close(); // Close the iterators
iter2.close();
```

Figure 38. Example of concurrent table operations using iterators with different SQL statements

```java
MultiIter iter1 = null; // Iterator instance for retrieving
// data for first employee
String EmpNo1 = "000100"; // Employee number for first employee
#sql [ctx] iter1 =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo1}; // Assign result table to first iterator
MultiIter iter2 = null; // Iterator instance for retrieving
// data for second employee
String EmpNo2 = "000200"; // Employee number for second employee
#sql [ctx] iter2 =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo2}; // Assign result table to second iterator

// Process with iter1
// Process with iter2
iter1.close(); // Close the iterators
iter2.close();

public MultiIter openIter(String EmpNo) // Method to assign a result table
    // to an iterator instance
{
    MultiIter iter;
    #sql [ctx] iter =
        {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo}; // Method returns an iterator instance
    return iter;
}
```

Figure 38. Example of concurrent table operations using iterators with the same SQL statement
As with any other iterator, you need to remember to close this iterator after the last time you use it to prevent excessive storage consumption.

Using scrollable iterators in an SQLJ application

In addition to moving forward, one row at a time, through a result table, you might want to move backward or go directly to a specific row. The IBM Data Server Driver for JDBC and SQLJ provides this capability.

An iterator in which you can move forward, backward, or to a specific row is called a scrollable iterator. A scrollable iterator in SQLJ is equivalent to the result table of a database cursor that is declared as SCROLL.

Like a scrollable cursor, a scrollable iterator can be insensitive or sensitive. A sensitive scrollable iterator can be static or dynamic. Insensitive means that changes to the underlying table after the iterator is opened are not visible to the iterator. Insensitive iterators are read-only. Sensitive means that changes that the iterator or other processes make to the underlying table are visible to the iterator. Asensitive means that if the cursor is a read-only cursor, it behaves as an insensitive cursor. If it is not a read-only cursor, it behaves as a sensitive cursor.

If a scrollable iterator is static, the size of the result table and the order of the rows in the result table do not change after the iterator is opened. This means that you cannot insert into result tables, and if you delete a row of a result table, a delete hole occurs. If you update a row of the result table so that the row no longer qualifies for the result table, an update hole occurs. Fetching from a hole results in an SQLException.

If a scrollable iterator is dynamic, the size of the result table and the order of the rows in the result table can change after the iterator is opened. Rows that are inserted or deleted with INSERT and DELETE statements that are executed by the same application process are immediately visible. Rows that are inserted or deleted with INSERT and DELETE statements that are executed by other application processes are visible after the changes are committed.

Important: DB2 Database for Linux, UNIX, and Windows servers do not support dynamic scrollable cursors. You can use dynamic scrollable iterators in your SQLJ applications only if those applications access data on DB2 for z/OS servers, at Version 9 or later.

Important:

To create and use a scrollable iterator, you need to follow these steps:

```java
ResultSet myFunc(String empid) // Method to open an iterator and get a resultSet
{
    MyIter iter;
    #sql iter = {SELECT * FROM EMPLOYEE WHERE EMPNO = :empid};
    return iter.getResultSet();
}

// An application can call this method to get a resultSet for each
// employee ID. The application can process each resultSet separately.
...
ResultSet rs1 = myFunc("000100"); // Get employee record for employee ID 000100
...
ResultSet rs2 = myFunc("000200"); // Get employee record for employee ID 000200
```

Figure 39. Example of opening more than one instance of an iterator in a single application
1. Specify an iterator declaration clause that includes the following clauses:
   - `implements sqlj.runtime.Scrollable`
     This indicates that the iterator is scrollable.
   - `with (sensitivity=INSENSITIVE|SENSITIVE|ASENSITIVE) or with (sensitivity=SENSITIVE, dynamic=true|false)`
     `sensitivity=INSENSITIVE|SENSITIVE|ASENSITIVE` indicates whether update or delete operations on the underlying table can be visible to the iterator. The default sensitivity is `INSENSITIVE`.
     `dynamic=true|false` indicates whether the size of the result table or the order of the rows in the result table can change after the iterator is opened. The default value of dynamic is `false`.

   The iterator can be a named or positioned iterator.

   **Example:** The following iterator declaration clause declares a positioned, sensitive, dynamic, scrollable iterator:
   ```sql
   #sql public iterator ByPos
   implements sqlj.runtime.Scrollable
   with (sensitivity=SENSITIVE, dynamic=true) (String);
   ```

   **Example:** The following iterator declaration clause declares a named, insensitive, scrollable iterator:
   ```sql
   #sql public iterator ByName
   implements sqlj.runtime.Scrollable
   with (sensitivity=INSENSITIVE) (String EmpNo);
   ```

   **Restriction:** You cannot use a scrollable iterator to select columns with the following data types from a table on a DB2 Database for Linux, UNIX, and Windows server:
   - `LONG VARCHAR`
   - `LONG VARGRAPHIC`
   - `BLOB`
   - `CLOB`
   - `XML`
   - A distinct type that is based on any of the previous data types in this list
   - A structured type

2. Create an iterator object, which is an instance of your iterator class.

3. If you want to give the SQLJ runtime environment a hint about the initial fetch direction, use the `setFetchDirection(int direction)` method. `direction` can be `FETCH_FORWARD` or `FETCH_REVERSE`. If you do not invoke `setFetchDirection`, the fetch direction is `FETCH_FORWARD`.

4. For each row that you want to access:
   - For a named iterator, perform the following steps:
     a. Position the cursor using one of the methods listed in the following table.

   **Table 25. sqlj.runtime.Scrollable methods for positioning a scrollable cursor**

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>first</code></td>
<td>On the first row of the result table</td>
</tr>
<tr>
<td><code>last</code></td>
<td>On the last row of the result table</td>
</tr>
<tr>
<td><code>previous</code></td>
<td>On the previous row of the result table</td>
</tr>
<tr>
<td><code>next</code></td>
<td>On the next row of the result table</td>
</tr>
<tr>
<td><code>absolute(int n)</code></td>
<td>If <code>n&gt;0</code>, on row <code>n</code> of the result table. If <code>n&lt;0</code>, and <code>m</code> is the number of rows in the result table, on row <code>m+n+1</code> of the result table.</td>
</tr>
</tbody>
</table>
Table 25. sqlj.runtime.Scrollable methods for positioning a scrollable cursor (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>relative(int n)</td>
<td>If ( n &gt; 0 ), on the row that is ( n ) rows after the current row.</td>
</tr>
<tr>
<td></td>
<td>If ( n &lt; 0 ), on the row that is ( n ) rows before the current row.</td>
</tr>
<tr>
<td></td>
<td>If ( n = 0 ), on the current row.</td>
</tr>
<tr>
<td>afterLast</td>
<td>After the last row in the result table</td>
</tr>
<tr>
<td>beforeFirst</td>
<td>Before the first row in the result table</td>
</tr>
</tbody>
</table>

Notes:
1. This method does not apply to connections to IBM Informix.
2. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
3. If the absolute value of \( n \) is greater than the number of rows in the result table, this method positions the cursor after the last row if \( n \) is positive, or before the first row if \( n \) is negative.
4. Suppose that \( m \) is the number of rows in the result table and \( x \) is the current row number in the result table. If \( n > 0 \) and \( x + n > m \), the iterator is positioned after the last row. If \( n < 0 \) and \( x + n < 1 \), the iterator is positioned before the first row.

b. If you need to know the current cursor position, use thegetRow, isFirst, isLast, isBeforeFirst, or isAfterLast method to obtain this information.
   If you need to know the current fetch direction, invoke the getFetchDirection method.

c. Use accessor methods to retrieve the current row of the result table.
d. If update or delete operations by the iterator or by other means are visible in the result table, invoke the getWarnings method to check whether the current row is a hole.

For a positioned iterator, perform the following steps:
a. Use a FETCH statement with a fetch orientation clause to position the iterator and retrieve the current row of the result table. Table 26 lists the clauses that you can use to position the cursor.

Table 26. FETCH clauses for positioning a scrollable cursor

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIRST</td>
<td>On the first row of the result table</td>
</tr>
<tr>
<td>LAST</td>
<td>On the last row of the result table</td>
</tr>
<tr>
<td>PRIOR</td>
<td>On the previous row of the result table</td>
</tr>
<tr>
<td>NEXT</td>
<td>On the next row of the result table</td>
</tr>
<tr>
<td>ABSOLUTE(n)</td>
<td>If ( n &gt; 0 ), on row ( n ) of the result table. If ( n &lt; 0 ), and ( m ) is the number of rows in the result table, on row ( m + n + 1 ) of the result table.</td>
</tr>
<tr>
<td>RELATIVE(n)</td>
<td>If ( n &gt; 0 ), on the row that is ( n ) rows after the current row. If ( n &lt; 0 ), on the row that is ( n ) rows before the current row. If ( n = 0 ), on the current row.</td>
</tr>
<tr>
<td>AFTER</td>
<td>After the last row in the result table</td>
</tr>
<tr>
<td>BEFORE</td>
<td>Before the first row in the result table</td>
</tr>
</tbody>
</table>
Table 26. FETCH clauses for positioning a scrollable cursor (continued)

<table>
<thead>
<tr>
<th>Method</th>
<th>Positions the cursor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>This value is not supported for connections to IBM Informix</td>
</tr>
<tr>
<td>2.</td>
<td>If the cursor is after the last row of the result table, this method positions the cursor on the last row.</td>
</tr>
<tr>
<td>3.</td>
<td>If the absolute value of ( n ) is greater than the number of rows in the result table, this method positions the cursor after the last row if ( n ) is positive, or before the first row if ( n ) is negative.</td>
</tr>
<tr>
<td>4.</td>
<td>Suppose that ( m ) is the number of rows in the result table and ( x ) is the current row number in the result table. If ( n &gt; 0 ) and ( x + n &gt; m ), the iterator is positioned after the last row. If ( n &lt; 0 ) and ( x + n &lt; 1 ), the iterator is positioned before the first row.</td>
</tr>
<tr>
<td>5.</td>
<td>Values are not assigned to host expressions.</td>
</tr>
</tbody>
</table>

b. If update or delete operations by the iterator or by other means are visible in the result table, invoke the getWarnings method to check whether the current row is a hole.

5. Invoke the close method to close the iterator.

The following code demonstrates how to use a named iterator to retrieve the employee number and last name from all rows from the employee table in reverse order. The numbers to the right of selected statements correspond to the previously-described steps.

```java
#sql context Ctx; // Create connection context class Ctx
#sql iterator ScrollIter implements sqlj.runtime.Scrollable
(String EmpNo, String LastName);
{
    ... 
    Ctx ctxt =
        new Ctx("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
            userid,password,false); // Create connection context object ctxt
        // for the connection to NEW YORK
    ScrollIter scrlIter;
    #sql [ctxt]
    scrlIter(SELECT EMPNO, LASTNAME FROM EMPLOYEE);
    scrlIter.afterLast();
    while (scrlIter.previous())
    {
        System.out.println(scrlIter.EmpNo() + **
           + scrlIter.LastName());
    }
    scrlIter.close();
}
```

**Calling stored procedures in SQLJ applications**

To call a stored procedure, you use an executable clause that contains an SQL CALL statement.

You can execute the CALL statement with host identifier parameters. You can execute the CALL statement with literal parameters only if the DB2 server on which the CALL statement runs supports execution of the CALL statement dynamically.

The basic steps in calling a stored procedure are:
1. Assign values to input (IN or INOUT) parameters.
2. Call the stored procedure.
3. Process output (OUT or INOUT) parameters.
4. If the stored procedure returns multiple result sets, retrieve those result sets.

The following code illustrates calling a stored procedure that has three input parameters and three output parameters. The numbers to the right of selected statements correspond to the previously-described steps.

```java
String FirstName="TOM"; // Input parameters
String LastName="NARISINST";
String Address="IBM";
int CustNo; // Output parameters
String Mark;
String MarkErrorText;
...
```

```
#sql [myConnCtx] {CALL ADD_CUSTOMER(:IN FirstName,
    :IN LastName,
    :IN Address,
    :OUT CustNo,
    :OUT Mark,
    :OUT MarkErrorText});
// Call the stored procedure
System.out.println("Output parameters from ADD_CUSTOMER call: ");
System.out.println("Customer number for " + LastName + ": " + CustNo);
System.out.println("Error messages:" + MarkErrorText);
```

Figure 40. Example of calling a stored procedure in an SQLJ application

**Retrieving multiple result sets from a stored procedure in an SQLJ application**

Some stored procedures return one or more result sets to the calling program by including the DYNAMIC RESULT SETS \( n \) clause in the definition, with \( n>0 \), and opening cursors that are defined with the WITH RETURN clause. The calling program needs to retrieve the contents of those result sets.

To retrieve the rows from those result sets, you execute these steps:
1. Acquire an execution context for retrieving the result set from the stored procedure.
2. Associate the execution context with the CALL statement for the stored procedure.
   Do not use this execution context for any other purpose until you have retrieved and processed the last result set.
3. For each result set:
   a. Use the ExecutionContext method getNextResultSet to retrieve the result set.
   b. If you do not know the contents of the result set, use ResultSetMetaData methods to retrieve this information.
   c. Use an SQLJ result set iterator or JDBC ResultSet to retrieve the rows from the result set.

Result sets are returned to the calling program in the same order that their cursors are opened in the stored procedure. When there are no more result sets to retrieve, getNextResultSet returns a null value.

getNextResultSet has two forms:
`getNextResultSet();`  `getNextResultSet(int current);`
When you invoke the first form of getNextResultSet, SQLJ closes the currently-open result set and advances to the next result set. When you invoke the second form of getNextResultSet, the value of current indicates what SQLJ does with the currently-open result set before it advances to the next result set:

**java.sql.Statement.CLOSE_CURRENT_RESULT**
- Specifies that the current ResultSet object is closed when the next ResultSet object is returned.

**java.sql.Statement.KEEP_CURRENT_RESULT**
- Specifies that the current ResultSet object stays open when the next ResultSet object is returned.

**java.sql.Statement.CLOSE_ALL_RESULTS**
- Specifies that all open ResultSet objects are closed when the next ResultSet object is returned.

The following code calls a stored procedure that returns multiple result sets. For this example, it is assumed that the caller does not know the number of result sets to be returned or the contents of those result sets. It is also assumed that autoCommit is false. The numbers to the right of selected statements correspond to the previously-described steps.

```java
ExecutionContext execCtx=myConnCtx.getExecutionContext();
#sql [myConnCtx, execCtx] {CALL MULTRSSP();}
// MULTRSSP returns multiple result sets
ResultSet rs;
while ((rs = execCtx.getNextResultSet()) != null) {
    ResultSetMetaData rsmeta=rs.getMetaData();
    int numcols=rsmeta.getColumnCount();
    while (rs.next()) {
        for (int i=1; i<=numcols; i++) {
            String colval=rs.getString(i);
            System.out.println("Column " + i + " value is " + colval);
        }
    }
}
```

*Figure 41. Retrieving result sets from a stored procedure*

### LOBs in SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, you can retrieve LOB data into `Clob` or `Blob` host expressions or update CLOB, BLOB, or DBCLOB columns from `Clob` or `Blob` host expressions. You can also declare iterators with `Clob` or `Blob` data types to retrieve data from CLOB, BLOB, or DBCLOB columns.

**Retrieving or updating LOB data:** To retrieve data from a BLOB column, declare an iterator that includes a data type of `Blob` or `byte[]`. To retrieve data from a CLOB or DBCLOB column, declare an iterator in which the corresponding column has a `Clob` data type.

To update data in a BLOB column, use a host expression with data type `Blob`. To update data in a CLOB or DBCLOB column, use a host expression with data type `Clob`. 
**Progressive streaming or LOB locators:** In SQLJ applications, you can use progressive streaming, also known as dynamic data format, or LOB locators in the same way that you use them in JDBC applications.

**Java data types for retrieving or updating LOB column data in SQLJ applications**

When the deferPrepares property is set to true, and the IBM Data Server Driver for JDBC and SQLJ processes an uncustomized SQLJ statement that includes host expressions, the driver might need to do extra processing to determine data types. This extra processing can impact performance.

When the JDBC driver cannot immediately determine the data type of a parameter that is used with a LOB column, you need to choose a parameter data type that is compatible with the LOB data type.

**Input parameters for BLOB columns**

For input parameters for BLOB columns, you can use either of the following techniques:

- Use a java.sql.Blob input variable, which is an exact match for a BLOB column:

  ```java
  java.sql.Blob blobData;
  #sql {CALL STORPROC(:IN blobData)};
  ```

  Before you can use a java.sql.Blob input variable, you need to create a java.sql.Blob object, and then populate that object.

- Use an input parameter of type of sqlj.runtime.BinaryStream. A sqlj.runtime.BinaryStream object is compatible with a BLOB data type. For example:

  ```java
  java.io.ByteArrayInputStream byteStream =
      new java.io.ByteArrayInputStream(byteData);
  int numBytes = byteData.length;
  sqlj.runtime.BinaryStream binStream =
      new sqlj.runtime.BinaryStream(byteStream, numBytes);
  #sql {CALL STORPROC(:IN binStream)};
  ```

  You cannot use this technique for INOUT parameters.

**Output parameters for BLOB columns**

For output or INOUT parameters for BLOB columns, you can use the following technique:

- Declare the output parameter or INOUT variable with a java.sql.Blob data type:

  ```java
  java.sql.Blob blobData = null;
  #sql CALL STORPROC (:OUT blobData)};
  ```

  ```java
  java.sql.Blob blobData = null;
  #sql CALL STORPROC (:INOUT blobData)};
  ```

**Input parameters for CLOB columns**

For input parameters for CLOB columns, you can use one of the following techniques:

- Use a java.sql.Clob input variable, which is an exact match for a CLOB column:

  ```java
  java.sql.Clob clobData;
  #sql {CALL STORPROC(:IN clobData)};
  ```

  Before you can use a java.sql.Clob input variable, you need to create a java.sql.Clob object, and then populate that object.
• Use one of the following types of stream IN parameters:
  – A sqlj.runtime.CharacterStream input parameter:
    ```java
    java.lang.String charData;
    java.io.StringReader reader = new java.io.StringReader(charData);
    sqlj.runtime.CharacterStream charStream =
    new sqlj.runtime.CharacterStream(reader, charData.length);
    #sql {CALL STORPROC(:IN charStream)};
    ```
  – A sqlj.runtime.UnicodeStream parameter, for Unicode UTF-16 data:
    ```java
    byte[] charDataBytes = charData.getBytes("UnicodeBigUnmarked");
    java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(charDataBytes);
    sqlj.runtime.UnicodeStream uniStream =
    new sqlj.runtime.UnicodeStream(byteStream, charDataBytes.length);
    #sql {CALL STORPROC(:IN uniStream)};
    ```
  – A sqlj.runtime.AsciiStream parameter, for ASCII data:
    ```java
    byte[] charDataBytes = charData.getBytes("US-ASCII");
    java.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(charDataBytes);
    sqlj.runtime.AsciiStream asciiStream =
    new sqlj.runtime.AsciiStream(byteStream, charDataBytes.length);
    #sql {CALL STORPROC(:IN asciiStream)};
    ```
  For these calls, you need to specify the exact length of the input data. You
  cannot use this technique for INOUT parameters.
• Use a java.lang.String input parameter:
  ```java
  java.lang.String charData;
  #sql {CALL STORPROC(:IN charData)};
  ```

### Output parameters for CLOB columns

For output or INOUT parameters for CLOB columns, you can use one of the
following techniques:
• Use a java.sql.Clob output variable, which is an exact match for a CLOB column:
  ```java
  java.sql.Clob clobData = null;
  #sql {CALL STORPROC(:OUT clobData)};
  ```
• Use a java.lang.String output variable:
  ```java
  java.lang.String charData = null;
  #sql {CALL STORPROC(:OUT charData)};
  ```
  This technique should be used only if you know that the length of the retrieved
data is less than or equal to 32KB. Otherwise, the data is truncated.

### Output parameters for DBCLOB columns

DBCLOB output or INOUT parameters for stored procedures are not supported.

### SQLJ and JDBC in the same application

You can combine SQLJ clauses and JDBC calls in a single program.

To do this effectively, you need to be able to do the following things:
• Use a JDBC Connection to build an SQLJ ConnectionContext, or obtain a JDBC
  Connection from an SQLJ ConnectionContext.
• Use an SQLJ iterator to retrieve data from a JDBC ResultSet or generate a JDBC
  ResultSet from an SQLJ iterator.

**Building an SQLJ ConnectionContext from a JDBC Connection:** To do that:
1. Execute an SQLJ connection declaration clause to create a ConnectionContext class.
2. Load the driver or obtain a DataSource instance.
3. Invoke the SQLJ DriverManager.getConnection or DataSource.getConnection method to obtain a JDBC Connection.
4. Invoke the ConnectionContext constructor with the Connection as its argument to create the ConnectionContext object.

**Obtaining a JDBC Connection from an SQLJ ConnectionContext:** To do this,
1. Execute an SQLJ connection declaration clause to create a ConnectionContext class.
2. Load the driver or obtain a DataSource instance.
3. Invoke the ConnectionContext constructor with the URL of the driver and any other necessary parameters as its arguments to create the ConnectionContext object.
4. Invoke the JDBC ConnectionContext.getConnection method to create the JDBC Connection object.

See "Connect to a data source using SQLJ" for more information on SQLJ connections.

**Retrieving JDBC result sets using SQLJ iterators:** Use the iterator conversion statement to manipulate a JDBC result set as an SQLJ iterator. The general form of an iterator conversion statement is:

```
#sql iterator={CAST :result-set};
```

Before you can successfully cast a result set to an iterator, the iterator must conform to the following rules:
- The iterator must be declared as public.
- If the iterator is a positioned iterator, the number of columns in the result set must match the number of columns in the iterator. In addition, the data type of each column in the result set must match the data type of the corresponding column in the iterator.
- If the iterator is a named iterator, the name of each accessor method must match the name of a column in the result set. In addition, the data type of the object that an accessor method returns must match the data type of the corresponding column in the result set.

The code in Figure 42 on page 149 builds and executes a query using a JDBC call, executes an iterator conversion statement to convert the JDBC result set to an SQLJ iterator, and retrieves rows from the result table using the iterator.
#sql public iterator ByName(String LastName, Date HireDate);

public void HireDates(ConnectionContext connCtx, String whereClause)
{
    ByName nameiter; // Declare object of ByName class
    Connection conn = connCtx.getConnection(); // Create JDBC connection
    Statement stmt = conn.createStatement();
    String query = "SELECT LASTNAME, HIREDATE FROM EMPLOYEE";
    query += whereClause; // Build the query
    ResultSet rs = stmt.executeQuery(query);
    //sql [connCtx] nameiter = {CAST :rs}; // This iterator conversion clause converts the JDBC ResultSet rs to SQLJ iterator nameiter.
    while (nameiter.next())
    {
        System.out.println( nameiter.LastName() + " was hired on " + nameiter.HireDate());
    }
    nameiter.close();
    stmt.close();
}

Figure 42. Converting a JDBC result set to an SQLJ iterator

Notes to Figure 42

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This SQLJ clause creates the named iterator class ByName, which has accessor methods LastName() and HireDate() that return the data from result table columns LASTNAME and HIREDATE.</td>
</tr>
<tr>
<td>2</td>
<td>This statement and the following two statements build and prepare a query for dynamic execution using JDBC.</td>
</tr>
<tr>
<td>3</td>
<td>This JDBC statement executes the SELECT statement and assigns the result table to result set rs.</td>
</tr>
<tr>
<td>4</td>
<td>This iterator conversion clause converts the JDBC ResultSet rs to SQLJ iterator nameiter, and the following statements use nameiter to retrieve values from the result table.</td>
</tr>
<tr>
<td>5</td>
<td>The nameiter.close() method closes the SQLJ iterator and JDBC ResultSet rs.</td>
</tr>
</tbody>
</table>

Generating JDBC ResultSets from SQLJ iterators: Use the ResultSet method to generate a JDBC ResultSet from an SQLJ iterator. Every SQLJ iterator has a ResultSet method. After you access the ResultSet that underlies an iterator, you need to fetch rows using only the ResultSet.

The code in Figure 43 on page 150 generates a positioned iterator for a query, converts the iterator to a result set, and uses JDBC methods to fetch rows from the table.
Notes to Figure 43:

<table>
<thead>
<tr>
<th>Note</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This SQLJ clause executes the SELECT statement, constructs an iterator object that contains the result table for the SELECT statement, and assigns the iterator object to variable iter.</td>
</tr>
<tr>
<td>2</td>
<td>The ResultSet method accesses the ResultSet that underlies iterator iter.</td>
</tr>
<tr>
<td>3</td>
<td>The JDBC getString() and getDate() methods retrieve values from the ResultSet. The next() method moves the cursor to the next row in the ResultSet.</td>
</tr>
<tr>
<td>4</td>
<td>The rs.close() method closes the SQLJ iterator as well as the ResultSet.</td>
</tr>
</tbody>
</table>

Rules and restrictions for using JDBC ResultSets in SQLJ applications: When you write SQLJ applications that include JDBC result sets, observe the following rules and restrictions:

- You cannot cast a ResultSet to an SQLJ iterator if the ResultSet and the iterator have different holdability attributes.
  
  A JDBC ResultSet or an SQLJ iterator can remain open after a COMMIT operation. For a JDBC ResultSet, this characteristic is controlled by the IBM Data Server Driver for JDBC and SQLJ property resultSetHoldability. For an SQLJ iterator, this characteristic is controlled by the with holdability parameter of the iterator declaration. Casting a ResultSet that has holdability to an SQLJ iterator that does not, or casting a ResultSet that does not have holdability to an SQLJ iterator that does, is not supported.

- Close the iterator or the underlying ResultSet object as soon as the program no longer uses the iterator or ResultSet, and before the end of the program. Closing the iterator also closes the ResultSet object. Closing the ResultSet object also closes the iterator object. In general, it is best to close the object that is used last.

- For the IBM Data Server Driver for JDBC and SQLJ, which supports scrollable iterators and scrollable and updatable ResultSet objects, the following restrictions apply:
  - Scrollable iterators have the same restrictions as their underlying JDBC ResultSet objects.
  - You cannot cast a JDBC ResultSet that is not updatable to an SQLJ iterator that is updatable.

Controlling the execution of SQL statements in SQLJ

You can use selected methods of the SQLJ ExecutionContext class to control or monitor the execution of SQL statements.
To use ExecutionContext methods, follow these steps:

1. Acquire the default execution context from the connection context.

   There are two ways to acquire an execution context:
   - Acquire the default execution context from the connection context. For example:
     ```java
     ExecutionContext execCtx = connCtx.getExecutionContext();
     ```
   - Create a new execution context by invoking the constructor for ExecutionContext. For example:
     ```java
     ExecutionContext execCtx = new ExecutionContext();
     ```

2. Associate the execution context with an SQL statement.

   To do that, specify an execution context after the connection context in the execution clause that contains the SQL statement.

3. Invoke ExecutionContext methods.

   Some ExecutionContext methods are applicable before the associated SQL statement is executed, and some are applicable only after their associated SQL statement is executed.

   For example, you can use method getUpdateCount to count the number of rows that are deleted by a DELETE statement after you execute the DELETE statement.

   The following code demonstrates how to acquire an execution context, and then use the getUpdateCount method on that execution context to determine the number of rows that were deleted by a DELETE statement. The numbers to the right of selected statements correspond to the previously-described steps.

   ```java
   ExecutionContext execCtx = new ExecutionContext();
   \#sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};
   System.out.println("Deleted " + execCtx.getUpdateCount() + " rows");
   ```

**ROWIDs in SQLJ with the IBM Data Server Driver for JDBC and SQLJ**

DB2 for z/OS and DB2 for i support the ROWID data type for a column in a table. A ROWID is a value that uniquely identifies a row in a table.

Although IBM Informix (IDS) also supports rowids, those rowids have the INTEGER data type. You can select an IDS rowid column into a variable with a four-byte integer data type.

If you use columns with the ROWID data type in SQLJ programs, you need to customize those programs.

JDBC 4.0 includes interface java.sql.RowId that you can use in iterators and in CALL statement parameters. If you do not have JDBC 4.0, you can use the IBM Data Server Driver for JDBC and SQLJ-only class com.ibm.db2.jcc.DB2RowID. For an iterator, you can also use the byte[] object type to retrieve ROWID values.

The following code shows an example of an iterator that is used to select values from a ROWID column:
The following code shows an example of calling a stored procedure that takes three ROWID parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```
#sql iterator PosIter(int,String,java.sql.RowId);
   // Declare positioned iterator
   // for retrieving ITEM_ID (INTEGER),
   // ITEM_FORMAT (VARCHAR), and ITEM_ROWID (ROWID)
   // values from table ROWIDTAB
{
   PosIter positrowid;  // Declare object of PosIter class
   java.sql.RowId rowid = null;
   int id = 0;
   String i_fmt = null;  // Declare host expressions
   #sql [ctxt] positrowid =
   {SELECT ITEM_ID, ITEM_FORMAT, ITEM_ROWID FROM ROWIDTAB
   WHERE ITEM_ID=3};
   // Assign the result table of the SELECT
   // to iterator object positrowid
   #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
   // Retrieve the first row
   while (!positrowid.endFetch())
   // Check whether the FETCH returned a row
   {System.out.println("Item ID" + id + " Item format " +
   i_fmt + " Item ROWID ");
   MyUtilities.printBytes(rowid.getBytes());
   // Use the getBytes method to
   // convert the value to bytes for printing.
   // Call a user-defined method called
   // printBytes (not shown) to print
   // the value.
   #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
   // Retrieve the next row
   }
   positrowid.close();  // Close the iterator
}
```

Figure 44. Example of using an iterator to retrieve ROWID values

The following code shows an example of calling a stored procedure that takes three ROWID parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```
java.sql.RowId in_rowid = rowid;
java.sqlRowId out_rowid = null;
java.sql.RowId inout_rowid = rowid;
   // Declare an IN, OUT, and
   // INOUT ROWID parameter
...
#sql [myConnCtx] {CALL SP_ROWID(:IN in_rowid,
   :OUT out_rowid,
   :INOUT inout_rowid)};
   // Call the stored procedure
System.out.println("Parameter values from SP_ROWID call: ");
System.out.println("OUT parameter value ");
MyUtilities.printBytes(out_rowid.getBytes());
   // Use the getBytes method to
   // convert the value to bytes for printing
   // Call a user-defined method called
   // printBytes (not shown) to print
   // the value.
System.out.println("INOUT parameter value ");
MyUtilities.printBytes(inout_rowid.getBytes());
```

Figure 45. Example of calling a stored procedure with a ROWID parameter
Distinct types in SQLJ applications

In an SQLJ program, you can create a distinct type using the CREATE DISTINCT TYPE statement in an executable clause.

You can also use CREATE TABLE in an executable clause to create a table that includes a column of that type. When you retrieve data from a column of that type, or update a column of that type, you use Java host variables or expressions with data types that correspond to the built-in types on which the distinct types are based.

The following example creates a distinct type that is based on an INTEGER type, creates a table with a column of that type, inserts a row into the table, and retrieves the row from the table:

```java
String empNumVar;
int shoeSizeVar;
...
#sql [myConnCtx] {CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS};
  // Create distinct type
#sql [myConnCtx] [COMMIT]; // Commit the create
#sql [myConnCtx] [CREATE TABLE EMP_SHOE
  (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)];
  // Create table using distinct type
#sql [myConnCtx] [COMMIT]; // Commit the create
#sql [myConnCtx] [INSERT INTO EMP_SHOE
  VALUES('000010', 6)]; // Insert a row in the table
#sql [myConnCtx] [COMMIT]; // Commit the INSERT
#sql [myConnCtx] [SELECT EMPNO, EMP_SHOE_SIZE
  INTO :empNumVar, :shoeSizeVar
  FROM EMP_SHOE]; // Retrieve the row
System.out.println("Employee number: " + empNumVar +
  " Shoe size: " + shoeSizeVar);
```

Figure 46. Defining and using a distinct type

Invocation of stored procedures with ARRAY parameters in SQLJ applications

SQLJ applications that run under the IBM Data Server Driver for JDBC and SQLJ and connect to DB2 Database for Linux, UNIX, and Windows data sources can call stored procedures that have ARRAY parameters.

You can use java.sql.Array objects as IN, OUT, or INOUT parameters in a stored procedure.

For IN or INOUT parameters, use the DB2Connection.createArrayOf method (JDBC 3.0 or earlier) or the Connection.createArrayOf method (JDBC 4.0 or later) to create a java.sql.Array object.

There are two ways to retrieve data from an ARRAY output stored procedure parameter:

- Use the java.sql.Array.getArray method to retrieve the contents of output parameter into a Java array.
- Use a java.sql.Array.getResultSet method to retrieve the output parameter data into a ResultSet object. Then use ResultSet methods to retrieve elements of the array. Each row of the ResultSet contains two columns:
  - An index into the array, which starts at 1
  - The array element
You need to retrieve the array elements from the ResultSet using the `getObject` method.

**Example:** Suppose that input and output parameters `IN_PHONE` and `OUT_PHONE` in stored procedure `GET_EMP_DATA` are arrays that are defined like this:

```sql
CREATE TYPE PHONENUMBERS AS VARCHAR(10) ARRAY[5]
```

Call `GET_EMP_DATA` with the two parameters.

```java
Connection con;
String type = "CHAR";
String [] contents = {"1234", "5678", "9101"};
...
com.ibm.db2.jcc.DB2Connection db2con = (com.ibm.db2.jcc.DB2Connection) con;
    // Cast the Connection as a DB2Connection
    // so you can use the
    // DB2Connection.createArrayOf method
java.sql.Array inPhoneData = db2con.createArrayOf(type, contents);
java.sql.Array outPhoneData;
try {
    #sql [db2con] {CALL GET_EMP_DATA(:IN inPhoneData, :OUT outPhoneData )};
} catch( SQLException e ) {
    throw e;
}
ResultSet rs = outPhoneData.getResultSet();
while (rs.next()) {
    String phoneNum = (String)rs.getObject(2); // Get phone number
    System.out.println("Phone number=", phoneNum);
}
```

---

**Savepoints in SQLJ applications**

Under the IBM Data Server Driver for JDBC and SQLJ, you can include any form of the SQL `SAVEPOINT` statement in your SQLJ program.

An SQL savepoint represents the state of data and schemas at a particular point in time within a unit of work. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.

```sql
#sql context Ctx; // Create connection context class Ctx
String empNumVar;
int shoeSizeVar;
...
try {
    // Load the JDBC driver
    Class.forName("com.ibm.db2.jcc.DB2Driver");
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
Connection jdbccon= DriverManager.getConnection("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK", userid,password);
    // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit
Ctx ctxt=new Ctx(jdbccon);
```
XML data in SQLJ applications

In SQLJ applications, you can store data in XML columns and retrieve data from XML columns.

In DB2 tables, the XML built-in data type is used to store XML data in a column as a structured set of nodes in a tree format.

SQLJ applications can send XML data to the data server or retrieve XML data from the data server as textual XML data.

In SQLJ applications, you can:
- Store an entire XML document in an XML column using INSERT, UPDATE, or MERGE statements.
- Retrieve an entire XML document from an XML column using single-row SELECT statements or iterators.
- Retrieve a sequence from a document in an XML column by using the SQL XMLQUERY function to retrieve the sequence in the database, and then using single-row SELECT statements or iterators to retrieve the serialized XML string data into an application variable.
- Retrieve a sequence from a document in an XML column by using an XQuery expression, prepended with the string 'XQUERY', to retrieve the elements of the sequence into a result table in the database, in which each row of the result table represents an item in the sequence. Then use using single-row SELECT statements or iterators to retrieve the data into application variables.
- Retrieve a sequence from a document in an XML column as a user-defined table by using the SQL XMLTABLE function to define the result table and retrieve it. Then use using single-row SELECT statements or iterators to retrieve the data from the result table into application variables.

JDBC 4.0 java.sql.SQLXML objects can be used to retrieve and update data in XML columns. Invocations of metadata methods, such as ResultSetMetaData.getColumnType return the integer value java.sql.Types.SQLXML for an XML column type.
XML column updates in SQLJ applications

When you update or insert data into XML columns of a database table, the input data in your SQLJ applications must be in the textual format.

The host expression data types that you can use to update XML columns are:

- `java.sql.SQLXML` (requires an SDK for Java Version 6 or later, and the IBM Data Server Driver for JDBC and SQLJ version 4.0 or later)
- `com.ibm.db2.jcc.DB2Xml` (deprecated)
- `String`
- `byte`
- `Blob`
- `Clob`
- `sqlj.runtime.AsciiStream`
- `sqlj.runtime.BinaryStream`
- `sqlj.runtime.CharacterStream`

The encoding of XML data can be derived from the data itself, which is known as **internally encoded** data, or from external sources, which is known as **externally encoded** data. XML data that is sent to the database server as binary data is treated as internally encoded data. XML data that is sent to the data source as character data is treated as externally encoded data. The external encoding is the default encoding for the JVM.

External encoding for Java applications is always Unicode encoding.

Externally encoded data can have internal encoding. That is, the data might be sent to the data source as character data, but the data contains encoding information. The data source handles incompatibilities between internal and external encoding as follows:

- If the data source is DB2 Database for Linux, UNIX, and Windows, the data source generates an error if the external and internal encoding are incompatible, unless the external and internal encoding are Unicode. If the external and internal encoding are Unicode, the data source ignores the internal encoding.
- If the data source is DB2 for z/OS, the data source ignores internal encoding.

Data in XML columns is stored in UTF-8 encoding.

**Example:** Suppose that you use the following statement to insert data from String host expression `xmlString` into an XML column in a table. `xmlString` is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

```sql
#sql [ctx] {INSERT INTO CUSTACC VALUES (1, :xmlString)};
```

**Example:** Suppose that you copy the data from `xmlString` into a byte array with CP500 encoding. The data contains an XML declaration with an encoding declaration for CP500. Then you insert the data from the byte[] host expression into an XML column in a table.

```java
byte[] xmlBytes = xmlString.getBytes("CP500");
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :xmlBytes)};
```

A byte string is considered to be internally encoded data. The data is converted from its internal encoding scheme to UTF-8, if necessary, and stored in its hierarchical format on the data source.
Example: Suppose that you copy the data from xmlString into a byte array with US-ASCII encoding. Then you construct an sqlj.runtime.AsciiStream host expression, and insert data from the sqlj.runtime.AsciiStream host expression into an XML column in a table on a data source.

```java
byte[] b = xmlString.getBytes("US-ASCII");
java.io.ByteArrayInputStream xmlAsciiInputStream =
    new java.io.ByteArrayInputStream(b);
sqlj.runtime.AsciiStream sqljXmlAsciiStream =
    new sqlj.runtime.AsciiStream(xmlAsciiInputStream, b.length);
#sql[ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlAsciiStream)};
```

sqljXmlAsciiStream is a stream type, so its internal encoding is used. The data is converted from its internal encoding to UTF-8 encoding and stored in its hierarchical form on the data source.


```java
java.io.StringReader xmlReader =
    new java.io.StringReader(xmlString);
sqlj.runtime.CharacterStream sqljXmlCharacterStream =
    new sqlj.runtime.CharacterStream(xmlReader, xmlString.length());
#sql [ctx] {INSERT INTO CUSTACC VALUES (4, :sqljXmlCharacterStream)};
```

sqljXmlCharacterStream is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

Example: Suppose that you retrieve a document from an XML column into a java.sql.SQLXML host expression, and insert the data into an XML column in a table.

```java
java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTACC");
rs.next();
java.sql.SQLXML xmlObject = (java.sql.SQLXML)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTACC VALUES (6, :xmlObject)};
```

After you retrieve the data it is still in UTF-8 encoding, so when you insert the data into another XML column, no conversion occurs.

Example: Suppose that you retrieve a document from an XML column into a com.ibm.db2.jcc.DB2Xml host expression, and insert the data into an XML column in a table.

```java
java.sql.ResultSet rs = s.executeQuery ("SELECT * FROM CUSTACC");
rs.next();
com.ibm.db2.jcc.DB2Xml xmlObject = (com.ibm.db2.jcc.DB2Xml)rs.getObject(2);
#sql [ctx] {INSERT INTO CUSTACC VALUES (6, :xmlObject)};
```

After you retrieve the data it is still in UTF-8 encoding, so when you insert the data into another XML column, no conversion occurs.

### XML data retrieval in SQLJ applications

When you retrieve data from XML columns of a database table in an SQLJ application, the output data must be explicitly or implicitly serialized.

The host expression or iterator data types that you can use to retrieve data from XML columns are:
- java.sql.SQLXML (requires an SDK for Java Version 6 or later, and the IBM Data Server Driver for JDBC and SQLJ version 4.0 or later)
If the application does not call the XMLSERIALIZE function before data retrieval, the data is converted from UTF-8 to the external application encoding for the character data types, or the internal encoding for the binary data types. No XML declaration is added. If the host expression is an object of the java.sql.SQLXML or com.ibm.db2.jcc.DB2Xml type, you need to call an additional method to retrieve the data from this object. The method that you call determines the encoding of the output data and whether an XML declaration with an encoding specification is added.

The following table lists the methods that you can call to retrieve data from a java.sql.SQLXML or a com.ibm.db2.jcc.DB2Xml object, and the corresponding output data types and type of encoding in the XML declarations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Output data type</th>
<th>Type of XML internal encoding declaration added</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLXML.getBinaryStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getCharacterStream</td>
<td>Reader</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getSource</td>
<td>Source</td>
<td>None</td>
</tr>
<tr>
<td>SQLXML.getString</td>
<td>String</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2AsciiStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2BinaryStream</td>
<td>InputStream</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2Bytes</td>
<td>byte[]</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2CharacterStream</td>
<td>Reader</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2String</td>
<td>String</td>
<td>None</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlAsciiStream</td>
<td>InputStream</td>
<td>US-ASCII</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlBinaryStream</td>
<td>InputStream</td>
<td>Specified by getDB2XmlBinaryStream targetEncoding parameter</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlBytes</td>
<td>byte[]</td>
<td>Specified by DB2XmlgetBytes targetEncoding parameter</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlCharacterStream</td>
<td>Reader</td>
<td>ISO-10646-UCS-2</td>
</tr>
<tr>
<td>DB2Xml.getDB2XmlString</td>
<td>String</td>
<td>ISO-10646-UCS-2</td>
</tr>
</tbody>
</table>

If the application executes the XMLSERIALIZE function on the data that is to be returned, after execution of the function, the data has the data type that is specified in the XMLSERIALIZE function, not the XML data type. Therefore, the driver handles the data as the specified type and ignores any internal encoding declarations.

Example: Suppose that you retrieve data from an XML column into a String host expression.

```sql
#sql iterator XmlStringIter (int, String);
#sql [ctx] siter = {SELECT C1, CADOC from CUSTACC};
#sql {FETCH :siter INTO :row, :outString};
```
The String type is a character type, so the data is converted from UTF-8 to the external encoding, which is the default JVM encoding, and returned without any XML declaration.

Example: Suppose that you retrieve data from an XML column into a byte[] host expression.

```sql
#sql iterator XmlByteArrayIter (int, byte[]);
XmlByteArrayIter biter = null;
#sql [ctx] biter = {SELECT c1, CADOC FROM CUSTACC};
#sql {FETCH :biter INTO :row, :outBytes};
```

The byte[] type is a binary type, so no data conversion from UTF-8 encoding occurs, and the data is returned without any XML declaration.

Example: Suppose that you retrieve a document from an XML column into a java.sql.SQLXML host expression, but you need the data in a binary stream.

```sql
#sql iterator SqlXmlIter (int, java.sql.SQLXML);
SqlXmlIter SQLXMLiter = null;
java.sql.SQLXML outSqlXml = null;
#sql [ctx] SQLXMLiter = {SELECT c1, CADOC FROM CUSTACC};
#sql {FETCH :SqlXmlIter INTO :row, :outSqlXml};
java.io.InputStream XmlStream = outSqlXml.getBinaryStream();
```

The FETCH statement retrieves the data into the SQLXML object in UTF-8 encoding. The SQLXML.getBinaryStream stores the data in a binary stream.

Example: Suppose that you retrieve a document from an XML column into a com.ibm.db2.jcc.DB2Xml host expression, but you need the data in a byte string with an XML declaration that includes an internal encoding specification for UTF-8.

```sql
#sql iterator DB2XmlIter (int, com.ibm.db2.jcc.DB2Xml);
DB2XmlIter db2xmliter = null;
com.ibm.db2.jcc.DB2Xml outDB2Xml = null;
#sql [ctx] db2xmliter = {SELECT c1, CADOC FROM CUSTACC};
#sql {FETCH :db2xmliter INTO :row, :outDB2Xml};
byte[] byteArray = outDB2XML.getDB2XmlBytes("UTF-8");
```

The FETCH statement retrieves the data into the DB2Xml object in UTF-8 encoding. The getDB2XmlBytes method with the UTF-8 argument adds an XML declaration with a UTF-8 encoding specification and stores the data in a byte array.

**XMLCAST in SQLJ applications**

Before you can use XMLCAST to cast a host variable to the XML data type in an SQLJ application, you need to cast the host variable to the corresponding SQL data type.

Example: The following code demonstrates a situation in which it is necessary to cast a String host variable to an SQL character type, such as VARCHAR, before you use XMLCAST to cast the value to the XML data type.

```sql
String xmlresult = null;
String varchar_hv = "San Jose";
...
#sql [con] {SELECT XMLCAST(CAST(:varchar_hv AS VARCHAR(32)) AS XML) INTO :xmlresult FROM SYSIBM.SYSDUMMY1};
```
SQLJ utilization of SDK for Java Version 5 function

Your SQLJ applications can use a number of functions that were introduced with the SDK for Java Version 5.

Static import

The static import construct lets you access static members without qualifying those members with the name of the class to which they belong. For SQLJ applications, this means that you can use static members in host expressions without qualifying them.

Example: Suppose that you want to declare a host expression of this form:

double r = cos(PI * E);

cos, PI, and E are members of the java.lang.Math class. To declare r without explicitly qualifying cos, PI, and E, include the following static import statement in your program:

import static java.lang.Math.*;

Annotations

Java annotations are a means for adding metadata to Java programs that can also affect the way that those programs are treated by tools and libraries. Annotations are declared with annotation type declarations, which are similar to interface declarations. Java annotations can appear in the following types of classes or interfaces:

- Class declaration
- Interface declaration
- Nested class declaration
- Nested interface declaration

You cannot include Java annotations directly in SQLJ programs, but you can include annotations in Java source code, and then include that source code in your SQLJ programs.

Example: Suppose that you declare the following marker annotation in a program called MyAnnot.java:

```java
public @interface MyAnot { }
```

You also declare the following marker annotation in a program called MyAnnot2.java:

```java
public @interface MyAnot2 { }
```

You can then use those annotations in an SQLJ program:

```java
// Class annotations
@MyAnnot2 public @MyAnot class TestAnnotation {
// Field annotation
@MyAnot
private static final int field1 = 0;
// Constructor annotation
@MyAnnot2 public @MyAnot TestAnnotation () {}  
// Method annotation
@MyAnot
public static void main (String a[]) {
```
TestAnnotation TestAnnotation_o = new TestAnnotation();
TestAnnotation_o.runThis();

// Inner class annotation
public static @MyAnot class TestAnotherInnerClass { }
// Inner interface annotation
public static @MyAnot interface TestAnotInnerInterface { }

Enumerated types

An enumerated type is a data type that consists of a set of ordered values. The
SDK for Java version 5 introduces the enum type for enumerated types.

You can include enums in the following places:
- In Java source files (.java files) that you include in an SQLJ program
- In SQLJ class declarations

Example: The TestEnum.sqlj class declaration includes an enum type:

```java
public class TestEnum2
{
    public enum Color {
        RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
    }
    Color color;
    ...
    // Get the value of color
    switch (color) {
        case RED:
            System.out.println("Red is at one end of the spectrum.");
            SQL[ctx] { INSERT INTO MYTABLE VALUES (:color) };
            break;
        case VIOLET:
            System.out.println("Violet is on the other end of the spectrum.");
            break;
        case ORANGE:
        case YELLOW:
        case GREEN:
        case BLUE:
        case INDIGO:
            System.out.println("Everything else is in the middle.");
            break;
    }
}
```

Generics

You can use generics in your Java programs to assign a type to a Java collection.
The SQLJ translator tolerates Java generic syntax. Examples of generics that you
can use in SQLJ programs are:

- A List of List objects:
  ```java
  List<List<String>> strList2 = new ArrayList<List<String>>();
  ```
- A HashMap in which the key/value pair has the String type:
  ```java
  Map<String,String> map = new HashMap<String,String>();
  ```
- A method that takes a List with elements of any type:
  ```java
  public void mthd(List<?> obj) {
      ...
  }
  ```

Although you can use generics in SQLJ host variables, the value of doing so is
limited because the SQLJ translator cannot determine the types of those host
variables.
**Enhanced for loop**

The enhanced for lets you specify that a set of operations is performed on each member of a collection or array. You can use the iterator in the enhanced for loop in host expressions.

**Example:** INSERT each of the items in array names into table TAB.

```java
String[] names = {"ABC","DEF","GHI"};
for (String n : names)
{
    #sql {INSERT INTO TAB (VARCHARCOL) VALUES(:n) };
}
```

**Varargs**

Varargs make it easier to pass an arbitrary number of values to a method. A Vararg in the last argument position of a method declaration indicates that the last arguments are an array or a sequence of arguments. An SQLJ program can use the passed arguments in host expressions.

**Example:** Pass an arbitrary number of parameters of type Object, to a method that inserts each parameter value into table TAB.

```java
public void runThis(Object... objects) throws SQLException
{
    for (Object obj : objects)
    {
        #sql { INSERT INTO TAB (VARCHARCOL) VALUES(:obj) };
    }
}
```

## Transaction control in SQLJ applications

In SQLJ applications, as in other types of SQL applications, transaction control involves explicitly or implicitly committing and rolling back transactions, and setting the isolation level for transactions.

### Setting the isolation level for an SQLJ transaction

To set the isolation level for a unit of work within an SQLJ program, use the SET TRANSACTION ISOLATION LEVEL clause.

The following table shows the values that you can specify in the SET TRANSACTION ISOLATION LEVEL clause and their DB2 equivalents.

<table>
<thead>
<tr>
<th>SET TRANSACTION value</th>
<th>DB2 isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serializable</td>
<td>Repeatable read</td>
</tr>
<tr>
<td>Repeatable Read</td>
<td>Read stability</td>
</tr>
<tr>
<td>Read Committed</td>
<td>Cursor stability</td>
</tr>
<tr>
<td>Read Uncommitted</td>
<td>Uncommitted read</td>
</tr>
</tbody>
</table>

The isolation level affects the underlying JDBC connection as well as the SQLJ connection.
Committing or rolling back SQLJ transactions

If you disable autocommit for an SQLJ connection, you need to perform explicit commit or rollback operations.

You do this using execution clauses that contain the SQL COMMIT or ROLLBACK statements.

To commit a transaction in an SQLJ program, use a statement like this:

```sql
#sql [myConnCtx] {COMMIT};
```

To roll back a transaction in an SQLJ program, use a statement like this:

```sql
#sql [myConnCtx] {ROLLBACK};
```

Handling SQL errors and warnings in SQLJ applications

SQLJ clauses throw SQLExceptions when SQL errors occur, but not when most SQL warnings occur.

SQLJ generates an SQLException under the following circumstances:
- When any SQL statement returns a negative SQL error code
- When a SELECT INTO SQL statement returns a +100 SQL error code

You need to explicitly check for other SQL warnings.
- For SQL error handling, include try/catch blocks around SQLJ statements.
- For SQL warning handling, invoke the getWarnings method after every SQLJ statement.

Handling SQL errors in an SQLJ application

SQLJ clauses use the JDBC class java.sql.SQLException for error handling.

To handle SQL errors in SQLJ applications, following these steps:
1. Import the java.sql.SQLException class.
2. Use the Java error handling try/catch blocks to modify program flow when an SQL error occurs.
3. Obtain error information from the SQLException.
   - You can use the getErrorCode method to retrieve SQL error codes and the getSQLState method to retrieve SQLSTATEs.
   - If you are using the IBM Data Server Driver for JDBC and SQLJ, obtain additional information from the SQLException by casting it to a DB2Diagnosable object, in the same way that you obtain this information in a JDBC application.
   - For the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver), use the standard SQLException to retrieve SQL error information.

The following code prints out the SQL error that occurred if a SELECT statement fails.

```java
try {
    #sql [ctxt] {SELECT LASTNAME INTO :empname
      FROM EMPLOYEE WHERE EMPNO='000010'};
```
Handling SQL warnings in an SQLJ application

Other than a +100 SQL error code on a SELECT INTO statement, warnings from the data server do not throw SQLExceptions. To handle warnings from the data server, you need to give the program access to the java.sql.SQLWarning class.

If you want to retrieve data-server-specific information about a warning, you also need to give the program access to the com.ibm.db2.jdbc.DB2Diagnosable interface and the com.ibm.db2.jdbc.DB2Sqlca class. Then follow these steps:

1. Set up an execution context for that SQL clause. See “Control the execution of SQL statements in SQLJ” for information on how to set up an execution context.
2. To check for a warning from the data server, invoke the getWarnings method after you execute an SQLJ clause.
   
   getWarnings returns the first SQLWarning object that an SQL statement generates. Subsequent SQLWarning objects are chained to the first one.
3. To retrieve data-server-specific information from the SQLWarning object with the IBM Data Server Driver for JDBC and SQLJ, follow the instructions in “Handle an SQLException under the IBM Data Server Driver for JDBC and SQLJ”.

The following example demonstrates how to retrieve an SQLWarning object for an SQL clause with execution context execCtx. The numbers to the right of selected statements correspond to the previously-described steps.

```java
ExecutionContext execCtx=myConnCtx.getExecutionContext();  // Get default execution context from connection context

SQLWarning sqlWarn;
...
#sql [myConnCtx,execCtx] {SELECT LASTNAME INTO :empname
FROM EMPLOYEE WHERE EMPNO='000010'};
if ((sqlWarn = execCtx.getWarnings()) != null)  // Warning
System.out.println("SQLWarning " + sqlWarn);
```

Closing the connection to a data source in an SQLJ application

When you have finished with a connection to a data source, you need to close the connection to the data source. Doing so releases the connection context object’s DB2 and SQLJ resources immediately.

To close the connection to the data source, use one of the ConnectionContext.close methods.

- If you execute `ConnectionContext.close()` or `ConnectionContext.close(ConnectionContext.CLOSE_CONNECTION)`, the connection context, as well as the connection to the data source, are closed.
- If you execute `ConnectionContext.close(ConnectionContext.KEEP_CONNECTION)` the connection context is closed, but the connection to the data source is not.

The following code closes the connection context, but does not close the connection to the data source.
...  
ctx = new EzSqljctx(con0); // Create a connection context object  
    // from JDBC connection con0  
...  
EzSqljctx.close(ConnectionContext.KEEP_CONNECTION);  
    // Close the connection context but keep  
    // the connection to the data source open
Chapter 5. Security under the IBM Data Server Driver for JDBC and SQLJ

When you use the IBM Data Server Driver for JDBC and SQLJ, you choose a security mechanism by specifying a value for the securityMechanism property.

You can set this property in one of the following ways:

- If you use the DriverManager interface, set securityMechanism in a java.util.Properties object before you invoke the form of the getConnection method that includes the java.util.Properties parameter.
- If you use the DataSource interface, and you are creating and deploying your own DataSource objects, invoke the DataSource.setSecurityMechanism method after you create a DataSource object.

You can determine the security mechanism that is in effect for a connection by calling the DB2Connection.getDB2SecurityMechanism method.

The following table lists the security mechanisms that the IBM Data Server Driver for JDBC and SQLJ supports, and the data sources that support those security mechanisms.

<table>
<thead>
<tr>
<th>Security mechanism</th>
<th>Supported by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DB2 Database for Linux, UNIX, and Windows</td>
</tr>
<tr>
<td>User ID and password</td>
<td>Yes</td>
</tr>
<tr>
<td>User ID only</td>
<td>Yes</td>
</tr>
<tr>
<td>User ID and encrypted password</td>
<td>Yes</td>
</tr>
<tr>
<td>Encrypted user ID</td>
<td>Yes</td>
</tr>
<tr>
<td>Encrypted user ID and encrypted password</td>
<td>Yes</td>
</tr>
<tr>
<td>Encrypted user ID and encrypted security-sensitive data</td>
<td>No</td>
</tr>
<tr>
<td>Encrypted user ID, encrypted password, and encrypted security-sensitive data</td>
<td>Yes</td>
</tr>
<tr>
<td>Kerberos1</td>
<td>Yes</td>
</tr>
<tr>
<td>Plugin1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Note:

1. Available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.
2. The version of the data source must be DB2 for i V6R1 or later.
The following table lists the security mechanisms that the IBM Data Server Driver for JDBC and SQLJ supports, and the value that you need to specify for the securityMechanism property to specify each security mechanism.

The default security mechanism is CLEAR_TEXT_PASSWORD_SECURITY. If the server does not support CLEAR_TEXT_PASSWORD_SECURITY but supports ENCRYPTED_USER_AND_PASSWORD_SECURITY, the IBM Data Server Driver for JDBC and SQLJ driver updates the security mechanism to ENCRYPTED_USER_AND_PASSWORD_SECURITY and attempts to connect to the server. Any other mismatch in security mechanism support between the requester and the server results in an error.

Table 30. Security mechanisms supported by the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>Security mechanism</th>
<th>securityMechanism property value</th>
</tr>
</thead>
<tbody>
<tr>
<td>User ID and password</td>
<td>DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>User ID only</td>
<td>DB2BaseDataSource.USER_ONLY_SECURITY</td>
</tr>
<tr>
<td>User ID and encrypted password</td>
<td>DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>Encrypted user ID</td>
<td>DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>Encrypted user ID and encrypted password</td>
<td>DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>Encrypted user ID and encrypted</td>
<td>DB2BaseDataSource.ENCRYPTED_USER_AND_DATA_SECURITY</td>
</tr>
<tr>
<td>Encrypted user ID, encrypted</td>
<td>DB2BaseDataSource.ENCRYPTED_USER_AND_DATA_SECURITY</td>
</tr>
<tr>
<td>Kerberos</td>
<td>DB2BaseDataSource.KERBEROS_SECURITY</td>
</tr>
<tr>
<td>Plugin</td>
<td>DB2BaseDataSource.PLUGIN_SECURITY</td>
</tr>
</tbody>
</table>

The following table shows possible DB2 Database for Linux, UNIX, and Windows server authentication types and the compatible IBM Data Server Driver for JDBC and SQLJ securityMechanism property values.

Table 31. Compatible DB2 Database for Linux, UNIX, and Windows server authentication types and IBM Data Server Driver for JDBC and SQLJ securityMechanism values

<table>
<thead>
<tr>
<th>DB2 Database for Linux, UNIX, and Windows server authentication type</th>
<th>securityMechanism setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT</td>
<td>USER_ONLY_SECURITY</td>
</tr>
<tr>
<td>SERVER</td>
<td>CLEAR_TEXT_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>SERVER_ENCRYPT</td>
<td>CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, or ENCRYPTED_USER_AND_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>DATA_ENCRYPT</td>
<td>ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY</td>
</tr>
<tr>
<td>KERBEROS</td>
<td>KERBEROS_SECURITY or PLUGIN_SECURITY</td>
</tr>
<tr>
<td>KRB_SERVER_ENCRYPT</td>
<td>KERBEROS_SECURITY , PLUGIN_SECURITY, ENCRYPTED_PASSWORD_SECURITY, or ENCRYPTED_USER_AND_PASSWORD_SECURITY</td>
</tr>
<tr>
<td>GSSPLUGIN</td>
<td>PLUGIN_SECURITY, or KERBEROS_SECURITY</td>
</tr>
</tbody>
</table>
Table 31. Compatible DB2 Database for Linux, UNIX, and Windows server authentication types and IBM Data Server Driver for JDBC and SQLJ securityMechanism values (continued)

<table>
<thead>
<tr>
<th>DB2 Database for Linux, UNIX, and Windows server authentication type</th>
<th>securityMechanism setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSS_SERVER_ENCRYPT  ☑</td>
<td>CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, ENCRYPTED_USER_AND_PASSWORD_SECURITY, PLUGIN_SECURITY, or KERBEROS_SECURITY</td>
</tr>
</tbody>
</table>

Notes:
1. For PLUGIN_SECURITY, the plugin must be a Kerberos plugin.
2. For PLUGIN_SECURITY, one of the plugins at the server identifies itself as supporting Kerberos.
3. GSS_SERVER_ENCRYPT is a combination of GSPLUGIN and SERVER_ENCRYPT.

User ID and password security under the IBM Data Server Driver for JDBC and SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, one of the available security methods is user ID and password security.

To specify user ID and password security for a JDBC connection, use one of the following techniques.

For the DriverManager interface: You can specify the user ID and password directly in the DriverManager.getConnection invocation. For example:

```java
import java.sql.*; // JDBC base
...
String id = "dbadm"; // Set user ID
String pw = "dbadm"; // Set password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose"; // Set URL for the data source
Connection con = DriverManager.getConnection(url, id, pw); // Create connection
```

Another method is to set the user ID and password directly in the URL string. For example:

```java
import java.sql.*; // JDBC base
...
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose:user=dbadm;password=dbadm;"; // Set URL for the data source
Connection con = DriverManager.getConnection(url); // Create connection
```

Alternatively, you can set the user ID and password by setting the user and password properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. Optionally, you can set the securityMechanism property to indicate that you are using user ID and password security. For example:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC and SQLJ implementation of JDBC
...
Properties properties = new java.util.Properties();
// Create Properties object
properties.put("user", "dbadm"); // Set user ID for the connection
properties.put("password", "dbadm"); // Set password for the connection
properties.put("securityMechanism",
    new String("" + com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY + ""));
    // Set security mechanism to
    // user ID and password
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
    // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
    // Create connection

For the DataSource interface: you can specify the user ID and password directly in
the DataSource.getConnection Invocation. For example:
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC
    // and SQLJ implementation of JDBC
...
Context ctx=new InitialContext(); // Create context for JNDI
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
    // Get DataSource object
String id = "dbadm";
    // Set user ID
String pw = "dbadm";
    // Set password
Connection con = ds.getConnection(id, pw);
    // Create connection

Alternatively, if you create and deploy the DataSource object, you can set the user
ID and password by invoking the DataSource.setUser and DataSource.setPassword
methods after you create the DataSource object. Optionally, you can invoke the
DataSource.setSecurityMechanism method property to indicate that you are using
user ID and password security. For example:
...
com.ibm.db2.jcc.DB2SimpleDataSource ds =
    // Create DB2SimpleDataSource object
    new com.ibm.db2.jcc.DB2SimpleDataSource();
ds.setDriverType(4); // Set driver type
ds.setDatabaseName("san_jose"); // Set location
ds.setServerName("mvs1.sj.ibm.com"); // Set server name
ds.setPortNumber(5021); // Set port number
ds.setUser("dbadm"); // Set user ID
ds.setPassword("dbadm"); // Set password
ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY);
    // Set security mechanism to
    // user ID and password

User ID-only security under the IBM Data Server Driver for JDBC and
SQLJ

With the IBM Data Server Driver for JDBC and SQLJ, one of the available security
methods is user-ID only security.

To specify user ID security for a JDBC connection, use one of the following
techniques.

For the DriverManager interface: Set the user ID and security mechanism by
setting the user and securityMechanism properties in a Properties object, and then
invoking the form of the getConnection method that includes the Properties object
as a parameter. For example:
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver
    // for JDBC and SQLJ
// implementation of JDBC

Properties properties = new Properties();
// Create a Properties object
properties.put("user", "db2adm"); // Set user ID for the connection
properties.put("securityMechanism",
        new String("" + com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY + "");
// Set security mechanism to
// user ID only
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
// Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
// Create the connection

For the DataSource interface: If you create and deploy the DataSource object, you
can set the user ID and security mechanism by invoking the DataSource.setUser
and DataSource.setSecurityMechanism methods after you create the DataSource
object. For example:

import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver
// for JDBC and SQLJ
// implementation of JDBC

com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
    new com.ibm.db2.jcc.DB2SimpleDataSource();
// Create DB2SimpleDataSource object
db2ds.setDriverType(4); // Set the driver type
db2ds.setDatabaseName("san_jose"); // Set the location
db2ds.setServerName("mvs1.sj.ibm.com");
// Set the server name
db2ds.setPortNumber(5021); // Set the port number
db2ds.setUser("db2adm"); // Set the user ID
db2ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.USER_ONLY_SECURITY);
// Set security mechanism to
// user ID only

Encrypted password, user ID, or user ID and password security under
the IBM Data Server Driver for JDBC and SQLJ

IBM Data Server Driver for JDBC and SQLJ supports encrypted password security,
encrypted user ID security, or encrypted user ID and encrypted password security
for accessing data sources.

The IBM Data Server Driver for JDBC and SQLJ supports 56-bit DES (weak)
encryption or 256-bit AES (strong) encryption. AES encryption is available with
IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only. You set the
encryptionAlgorithm driver property to choose between 56-bit DES encryption
(encryptionAlgorithm value of 1) and 256-bit AES encryption (encryptionAlgorithm
value of 2). 256-bit AES encryption is used for a connection only if the database
server supports it and is configured to use it.

If you use encrypted password security, encrypted user ID security, or encrypted
user ID and encrypted password security, the IBM Java Cryptography Extension
(JCE) needs to be enabled on your client. The IBM JCE is part of the IBM SDK for
Java, Version 1.4.2 or later.

The IBM JCE needs to use 56-bit DES or 256-bit AES encrypted client/server
communication from the IBM Data Server Driver for JDBC and SQLJ driver to DB2
Database for Linux, UNIX, and Windows servers.
For AES encryption, you need to get the unrestricted policy file for JCE. It is available at the following URL: https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=jecesdk

Connections to DB2 for i V6R1 or later servers can use encrypted password security or encrypted user ID and encrypted password security. For encrypted password security or encrypted user ID and encrypted password security, the IBM Java Cryptography Extension (ibmjceprovidere.jar) must be installed on your client. The IBM JCE is part of the IBM SDK for Java, Version 1.4.2 or later.

You can also use encrypted security-sensitive data in addition to encrypted user ID security or encrypted user ID and encrypted password security. You specify encryption of security-sensitive data through the ENCRYPTED_USER_AND_DATA_SECURITY or ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY securityMechanism value. ENCRYPTED_USER_AND_DATA_SECURITY is valid for connections to DB2 for z/OS servers only, and only for DES encryption (encryptionAlgorithm value of 1).

DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows database servers encrypt the following data when you specify encryption of security-sensitive data:
- SQL statements that are being prepared, executed, or bound into a package
- Input and output parameter information
- Result sets
- LOB data
- XML data
- Results of describe operations

Before you can use encrypted security-sensitive data, the z/OS Integrated Cryptographic Services Facility needs to be installed and enabled on the z/OS operating system.

To specify encrypted user ID or encrypted password security for a JDBC connection, use one of the following techniques.

**For the DriverManager interface:** Set the user ID, password, and security mechanism by setting the user, password, and securityMechanism properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the user ID and encrypted password security mechanism, with AES encryption:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC
// and SQLJ implementation of JDBC

Properties properties = new Properties(); // Create a Properties object
properties.put("user", "dbadm"); // Set user ID for the connection
properties.put("password", "dbadm"); // Set password for the connection
properties.put("securityMechanism", "2");
properties.put("encryptionAlgorithm", "2"); // Request AES security
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose"; // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties); // Create the connection
```
For the `DataSource` interface: If you create and deploy the `DataSource` object, you can set the user ID, password, and security mechanism by invoking the `DataSource.setUser`, `DataSource.setPassword`, and `DataSource.setSecurityMechanism` methods after you create the `DataSource` object. For example, use code like this to set the encrypted user ID and encrypted password security mechanism, with AES encryption:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC
// and SQLJ implementation of JDBC
...
com.ibm.db2.jcc.DB2SimpleDataSource ds =
   new com.ibm.db2.jcc.DB2SimpleDataSource();
// Create the DataSource object
ds.setDriverType(4); // Set the driver type
ds.setDatabaseName("san_jose"); // Set the location
ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
ds.setPortNumber(5021); // Set the port number
ds.setUser("db2adm"); // Set the user ID
ds.setPassword("db2adm"); // Set the password
ds.setSecurityMechanism(
   com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY);
   // Set security mechanism to
   // User ID and encrypted password
ds.setEncryptionAlgorithm(2); // Request AES encryption
```

---

**Kerberos security under the IBM Data Server Driver for JDBC and SQLJ**

JDBC support for Kerberos security is available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

To enable JDBC support for Kerberos security, you also need to enable the following components of your software development kit (SDK) for Java:

- Java Cryptography Extension
- Java Generic Security Service (JGSS)
- Java Authentication and Authorization Service (JAAS)

See the documentation for your SDK for Java for information on how to enable these components.

There are three ways to specify Kerberos security for a connection:
- With a user ID and password
- Without a user ID or password
- With a delegated credential

**Kerberos security with a user ID and password**

For this case, Kerberos uses the specified user ID and password to obtain a ticket-granting ticket (TGT) that lets you authenticate to the database server.

You need to set the `user`, `password`, `kerberosServerPrincipal`, and `securityMechanism` properties. Set the `securityMechanism` property to `com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY` (11). The `kerberosServerPrincipal` property specifies the principal name that the database server registers with a Kerberos Key Distribution Center (KDC).
**For the DriverManager interface**: Set the user ID, password, Kerberos server, and security mechanism by setting the user, password, kerberosServerPrincipal, and securityMechanism properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the Kerberos security mechanism with a user ID and password:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC

Properties properties = new Properties(); // Create a Properties object
properties.put("user", "db2adm"); // Set user ID for the connection
properties.put("password", "db2adm"); // Set password for the connection
properties.put("kerberosServerPrincipal", "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
properties.put("securityMechanism", new String("" + com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + ""));
// Set security mechanism to Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose"; // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties); // Create the connection
```

**For the DataSource interface**: If you create and deploy the DataSource object, set the Kerberos server and security mechanism by invoking the DataSource.setKerberosServerPrincipal and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC

com.ibm.db2.jcc.DB2SimpleDataSource db2ds =
    new com.ibm.db2.jcc.DB2SimpleDataSource();
// Create the DataSource object
db2ds.setDriverType(4); // Set the driver type
db2ds.setDatabaseName("san_jose"); // Set the location
db2ds.setUser("db2adm"); // Set the user
db2ds.setPassword("db2adm"); // Set the password
db2ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
db2ds.setPortNumber(5021); // Set the port number
db2ds.setKerberosServerPrincipal("sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
db2ds.setSecurityMechanism(com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY); // Set security mechanism to Kerberos
```

**Kerberos security with no user ID or password**

For this case, the Kerberos default credentials cache must contain a ticket-granting ticket (TGT) that lets you authenticate to the database server.

You need to set the kerberosServerPrincipal and securityMechanism properties. Set the securityMechanism property to com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY (11).
For the DriverManager interface: Set the Kerberos server and security mechanism by setting the `kerberosServerPrincipal` and `securityMechanism` properties in a Properties object, and then invoking the form of the `getConnection` method that includes the Properties object as a parameter. For example, use code like this to set the Kerberos security mechanism without a user ID and password:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC

Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM");
// Set the Kerberos server
properties.put("securityMechanism", new String("" + com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "));
// Set security mechanism to Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose";
// Set URL for the data source
Connection con = DriverManager.getConnection(url, properties);
// Create the connection
```

For the DataSource interface: If you create and deploy the DataSource object, set the Kerberos server and security mechanism by invoking the `DataSource.setKerberosServerPrincipal` and `DataSource.setSecurityMechanism` methods after you create the DataSource object. For example:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC

DB2SimpleDataSource db2ds =
    new com.ibm.db2.jcc.DB2SimpleDataSource();
// Create the DataSource object
db2ds.setDriverType(4); // Set the driver type
db2ds.setDatabaseName("san_jose"); // Set the location
db2ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
db2ds.setPortNumber(5021); // Set the port number
db2ds.setKerberosServerPrincipal("sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
db2ds.setSecurityMechanism(
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY);
// Set security mechanism to Kerberos
```

Kerberos security with a delegated credential from another principal

For this case, you authenticate to the database server using a delegated credential that another principal passes to you.

You need to set the `kerberosServerPrincipal`, `gssCredential`, and `securityMechanism` properties. Set the `securityMechanism` property to `com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY (11)

For the DriverManager interface: Set the Kerberos server, delegated credential, and security mechanism by setting the `kerberosServerPrincipal`, `gssCredential`, and `securityMechanism` properties in a Properties object. Then invoke the form of the
getConnection method that includes the Properties object as a parameter. For example, use code like this to set the Kerberos security mechanism without a user ID and password:

```java
import java.sql.*; // JDBC base
import com.ibm.db2.jcc.*; // IBM Data Server Driver for JDBC
...
Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
properties.put("gssCredential",delegatedCredential); // Set the delegated credential
properties.put("securityMechanism", new String("" +
    com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY + "))); // Set security mechanism to
    // Kerberos
String url = "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose"; // Set URL for the data source
Connection con = DriverManager.getConnection(url, properties); // Create the connection
```

For the DataSource interface: If you create and deploy the DataSource object, set the Kerberos server, delegated credential, and security mechanism by invoking the DataSource.setKerberosServerPrincipal, DataSource.setGssCredential, and DataSource.setSecurityMechanism methods after you create the DataSource object. For example:

```java
DB2SimpleDataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource(); // Create the DataSource object
db2ds.setDriverType(4); // Set the driver type
db2ds.setDatabaseName("san_jose"); // Set the location
db2ds.setServerName("mvs1.sj.ibm.com"); // Set the server name
db2ds.setPortNumber(5021); // Set the port number
db2ds.setKerberosServerPrincipal("sample/srvlsj.ibm.com@SRVLSJ.SJ.IBM.COM"); // Set the Kerberos server
db2ds.setGssCredential(delegatedCredential); // Set the delegated credential
db2ds.setSecurityMechanism(com.ibm.db2.jcc.DB2BaseDataSource.KERBEROS_SECURITY); // Set security mechanism to
    // Kerberos
```

---

**IBM Data Server Driver for JDBC and SQLJ security plugin support**

You can create your own authentication mechanisms in the form of loadable libraries, or plugins, that DB2 Database for Linux, UNIX, and Windows loads to perform user authentication. To support development of security plugins in Java, the IBM Data Server Driver for JDBC and SQLJ provides security plugin support.

IBM Data Server Driver for JDBC and SQLJ security plugin support is available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows servers only.

To use plugin security, you need a security plugin on the client and another plugin on the server.

The security plugins need to include the following things:
- A class that extends the com.ibm.db2.jcc.DB2JCCPlugin abstract class
The com.ibm.db2.jcc.DB2JCCPlugin abstract class is provided with the IBM Data Server Driver for JDBC and SQLJ.

- Within the com.ibm.db2.jcc.DB2JCCPlugin class, a com.ibm.db2.jcc.DB2JCCPlugin.getTicket method
  This method retrieves a Kerberos ticket for a user and returns security context information in a byte array. The information in the byte array is used by the IBM Data Server Driver for JDBC and SQLJ to access the DB2 database server.

- Implementations of several methods that are defined in the org.ietf.jgss.GSSContext and org.ietf.jgss.GSSCredential interfaces
  These method implementations need to follow the Generic Security Service Application Program Interface, Version 2 (IETF RFC2743) and Generic Security Service API Version 2: Java-Bindings (IETF RFC2853) specifications. The plugin must implement and call the following methods:

  GSSContext.dispose
  Releases any system resources and cryptographic information that are stored in a context object, and invalidates the context.

  GSSContext.getCredDelegState
  Determines whether credential delegation is enabled on a context.

  GSSContext.getMutualAuthState
  Determines whether mutual authentication is enabled on the context.

  GSSContext.initSecContext
  Starts the context creation phase, and processes any tokens that are generated by the peer's acceptSecContext method.

  GSSContext.requestCredDeleg
  Requests that the credentials of the initiator are delegated to the acceptor when a context is established.

  GSSContext.requestMutualAuth
  Requests mutual authentication when a context is established.

  GSSCredential.dispose
  Releases any sensitive information that the GSSCredential object contains.

Two Java plugin samples are provided in sqllib/samples/java/jdbc to help you write Java security plugins:

  **JCCSimpleGSSPlugin.java**
  An implementation of a GSS-API plugin for the server, which performs user ID and password checking. This sample is a Java version of the C language sample program gssapi_simple.c.

  **JCKerberosPlugin.java**
  A Kerberos security plugin for the client. This sample is a Java version of the C language sample program IBMkrb5.c.

When an application program obtains a connection using JDBC plugin security, it needs to set the following Connection or DataSource properties:

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.user</td>
<td>The user ID under which the Connection is to be obtained</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.password</td>
<td>The password for the user ID</td>
</tr>
</tbody>
</table>
Table 32. Connection or DataSource property settings for Java security plugin use (continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism</td>
<td>com.ibm.db2.jcc.DB2BaseDataSource.PLUGIN_SECURITY</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.pluginName</td>
<td>The name of the plugin module for a server-side security plugin</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.plugin</td>
<td>The plugin object for a client-side security plugin</td>
</tr>
</tbody>
</table>

Example: The following code sets the properties for a connection that uses GSS-API plugin security. The connection uses the JCCSimpleGSSPlugin sample plugin on the client side, and the gssapi_simple sample plugin on the server side.

```java
java.util.Properties properties = new java.util.Properties();
properties.put("user", "db2admin");
properties.put("password", "admindb2");
properties.put("pluginName", "gssapi_simple");
properties.put("securityMechanism", new String(+com.ibm.db2.jcc.DB2BaseDataSource.PLUGIN_SECURITY+""));
com.ibm.db2.jcc.DB2JCCPlugin plugin = new com.ibm.db2.jcc.samples.plugins.JCCSimpleGSSPlugin();
properties.put("plugin", plugin);
Connection con = java.sql.DriverManager.getConnection(url, properties);
```

Use of alternative security mechanisms with the IBM Data Server Driver for JDBC and SQLJ

If you are using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and you set the `retryWithAlternativeSecurityMechanism` to `com.ibm.db2.jcc.DB2BaseDataSource.YES` (1), and the original security mechanism for a connection fails, the driver retries the connection with the most secure alternative security mechanism.

The following table lists the IBM Data Server Driver for JDBC and SQLJ security mechanisms, and the alternative security mechanisms that are used when the original connection has an authorization failure.

Table 33. Original and alternative IBM Data Server Driver for JDBC and SQLJ security mechanisms

<table>
<thead>
<tr>
<th>Server authentication type</th>
<th>IBM Data Server Driver for JDBC and SQLJ authentication type for the original connection</th>
<th>IBM Data Server Driver for JDBC and SQLJ authentication type for retrying the connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT</td>
<td>CLEAR_TEXT_PASSWORD_SECURITY&lt;br&gt;ENCRYPTED_PASSWORD_SECURITY&lt;br&gt;ENCRYPTED_USER_AND_PASSWORD_SECURITY&lt;br&gt;KERBEROS_SECURITY&lt;br&gt;ENCRYPTED_USER_AND_DATA_SECURITY&lt;br&gt;ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY&lt;br&gt;PLUGIN_SECURITY&lt;br&gt;ENCRYPTED_USER_ONLY_SECURITY&lt;br&gt;USER_ONLY_SECURITY</td>
<td>USER_ONLY_SECURITY&lt;br&gt;None. USER_ONLY_SECURITY does not fail on the original connection.</td>
</tr>
</tbody>
</table>
Table 33. Original and alternative IBM Data Server Driver for JDBC and SQLJ security mechanisms (continued)

<table>
<thead>
<tr>
<th>Server authentication type</th>
<th>IBM Data Server Driver for JDBC and SQLJ authentication type for the original connection</th>
<th>IBM Data Server Driver for JDBC and SQLJ authentication type for retrying the connection</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SERVER</strong></td>
<td>• USER_ONLY_SECURITY&lt;br&gt;• ENCRYPTED_PASSWORD_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_PASSWORD_SECURITY&lt;br&gt;• KERBEROS_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_DATA_SECURITY&lt;br&gt;• ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY&lt;br&gt;• PLUGIN_SECURITY&lt;br&gt;• ENCRYPTED_USER_ONLY_SECURITY</td>
<td>CLEAR_TEXT_PASSWORD_SECURITY&lt;br&gt;None. CLEAR_TEXT_PASSWORD_SECURITY does not fail on the original connection.</td>
</tr>
<tr>
<td><strong>SERVER_ENCRYPT for DB2 Database for Linux, UNIX, and Windows Version 8 Fix Pack 9 or earlier</strong></td>
<td>• CLEAR_TEXT_PASSWORD_SECURITY&lt;br&gt;• USER_ONLY_SECURITY&lt;br&gt;• KERBEROS_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_DATA_SECURITY&lt;br&gt;• ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY&lt;br&gt;• PLUGIN_SECURITY&lt;br&gt;• ENCRYPTED_USER_ONLY_SECURITY</td>
<td>ENCRYPTED_USER_AND_PASSWORD_SECURITY&lt;br&gt;None: ENCRYPTED_PASSWORD_SECURITY and ENCRYPTED_USER_AND_PASSWORD_SECURITY do not fail on the original connection.</td>
</tr>
<tr>
<td><strong>SERVER_ENCRYPT for DB2 Database for Linux, UNIX, and Windows Version 8 Fix Pack 10 or later</strong></td>
<td>• USER_ONLY_SECURITY&lt;br&gt;• KERBEROS_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_DATA_SECURITY&lt;br&gt;• ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY&lt;br&gt;• PLUGIN_SECURITY&lt;br&gt;• ENCRYPTED_USER_ONLY_SECURITY&lt;br&gt;• CLEAR_TEXT_PASSWORD_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_PASSWORD_SECURITY&lt;br&gt;</td>
<td>ENCRYPTED_USER_AND_PASSWORD_SECURITY&lt;br&gt;None: CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, and ENCRYPTED_USER_AND_PASSWORD_SECURITY do not fail on the original connection.</td>
</tr>
<tr>
<td><strong>DATA_ENCRYPT</strong></td>
<td>• CLEAR_TEXT_PASSWORD_SECURITY&lt;br&gt;• USER_ONLY_SECURITY&lt;br&gt;• ENCRYPTED_PASSWORD_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_PASSWORD_SECURITY&lt;br&gt;• KERBEROS_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_DATA_SECURITY&lt;br&gt;• PLUGIN_SECURITY&lt;br&gt;• ENCRYPTED_USER_ONLY_SECURITY&lt;br&gt;• ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY&lt;br&gt;</td>
<td>ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY&lt;br&gt;None: ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY does not fail on the original connection.</td>
</tr>
<tr>
<td><strong>KERBEROS</strong></td>
<td>• CLEAR_TEXT_PASSWORD_SECURITY&lt;br&gt;• USER_ONLY_SECURITY&lt;br&gt;• ENCRYPTED_PASSWORD_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_PASSWORD_SECURITY&lt;br&gt;• KERBEROS_SECURITY&lt;br&gt;• ENCRYPTED_USER_AND_DATA_SECURITY&lt;br&gt;• ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY&lt;br&gt;• PLUGIN_SECURITY&lt;br&gt;• ENCRYPTED_USER_ONLY_SECURITY&lt;br&gt;</td>
<td>KERBEROS_SECURITY&lt;br&gt;None: KERBEROS_SECURITY does not fail on the original connection.</td>
</tr>
</tbody>
</table>
Table 33. Original and alternative IBM Data Server Driver for JDBC and SQLJ security mechanisms (continued)

<table>
<thead>
<tr>
<th>Server authentication type</th>
<th>IBM Data Server Driver for JDBC and SQLJ authentication type for the original connection</th>
<th>IBM Data Server Driver for JDBC and SQLJ authentication type for retrying the connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSSPLUGIN</td>
<td>• CLEAR_TEXT_PASSWORD_SECURITY, USER_ONLY_SECURITY, ENCRYPTED_PASSWORD_SECURITY, ENCRYPTED_USER_AND_PASSWORD_SECURITY, KERBEROS_SECURITY, ENCRYPTED_USER_AND_DATA_SECURITY, ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY, ENCRYPTED_USER_ONLY_SECURITY</td>
<td>PLUGIN_SECURITY</td>
</tr>
<tr>
<td></td>
<td>PLUGIN_SECURITY None. PLUGIN_SECURITY does not fail on the original connection.</td>
<td></td>
</tr>
<tr>
<td>KRB_SERVER_ENCRYPT</td>
<td>• USER_ONLY_SECURITY, ENCRYPTED_USER_AND_DATA_SECURITY, ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY, ENCRYPTED_USER_ONLY_SECURITY</td>
<td>KERBEROS_SECURITY</td>
</tr>
<tr>
<td></td>
<td>• CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, ENCRYPTED_USER_AND_PASSWORD_SECURITY, KERBEROS_SECURITY, PLUGIN_SECURITY</td>
<td>None. CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, ENCRYPTED_USER_AND_PASSWORD_SECURITY, KERBEROS_SECURITY, and PLUGIN_SECURITY do not fail on the original connection.</td>
</tr>
<tr>
<td>GSS_SERVER_ENCRYPT</td>
<td>• USER_ONLY_SECURITY, ENCRYPTED_USER_AND_DATA_SECURITY, ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY, ENCRYPTED_USER_ONLY_SECURITY</td>
<td>KERBEROS_SECURITY</td>
</tr>
<tr>
<td></td>
<td>• CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, ENCRYPTED_USER_AND_PASSWORD_SECURITY, KERBEROS_SECURITY, PLUGIN_SECURITY</td>
<td>None. CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, ENCRYPTED_USER_AND_PASSWORD_SECURITY, KERBEROS_SECURITY, and PLUGIN_SECURITY do not fail on the original connection.</td>
</tr>
</tbody>
</table>

IBM Data Server Driver for JDBC and SQLJ trusted context support

The IBM Data Server Driver for JDBC and SQLJ provides methods that allow you to establish and use trusted connections in Java programs.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

A three-tiered application model consists of a database server, a middleware server such as WebSphere Application Server, and end users. With this model, the middleware server is responsible for accessing the database server on behalf of end users. Trusted context support ensures that an end user's database identity and database privileges are used when the middleware server performs any database requests on behalf of that end user.

A trusted context is an object that the database administrator defines that contains a system authorization ID and a set of trust attributes. Currently, for DB2 database servers, a database connection is the only type of context that is supported. The trust attributes identify a set of characteristics of a connection that are required for...
the connection to be considered a trusted connection. The relationship between a
database connection and a trusted context is established when the connection to
the database server is first created, and that relationship remains for the life of the
database connection.

After a trusted context is defined, and an initial trusted connection to the data
server is made, the middleware server can use that database connection under a
different user without reauthenticating the new user at the database server.

To avoid vulnerability to security breaches, an application server that uses these
trusted methods should not use untrusted connection methods.

The DB2ConnectionPoolDataSource class provides several versions of the
getDB2TrustedPooledConnection method, and the DB2XADatasource class
provides several versions of the getDB2TrustedXAConnection method, which allow
an application server to establish the initial trusted connection. You choose a
method based on the types of connection properties that you pass and whether
you use Kerberos security. When an application server calls one of these methods,
the IBM Data Server Driver for JDBC and SQLJ returns an Object[] array with two
elements:

• The first element contains a connection instance for the initial connection.
• The second element contains a unique cookie for the connection instance. The
  cookie is generated by the JDBC driver and is used for authentication during
  subsequent connection reuse.

The DB2PooledConnection class provides several versions of the getDB2Connection
method, and the DB2Connection class provides several versions of the
reuseDB2Connection method, which allow an application server to reuse an
existing trusted connection on behalf of a new user. The application server uses the
method to pass the following items to the new user:

• The cookie from the initial connection
• New connection properties for the reused connection

The JDBC driver checks that the supplied cookie matches the cookie of the
underlying trusted physical connection, to ensure that the connection request
originates from the application server that established the trusted physical
connection. If the cookies match, the connection becomes available for immediate
use by this new user, with the new properties.

Example: Obtain the initial trusted connection:

```
// Create a DB2ConnectionPoolDataSource instance
com.ibm.db2.jcc.DB2ConnectionPoolDataSource dataSource =
    new com.ibm.db2.jcc.DB2ConnectionPoolDataSource();
// Set properties for this instance
dataSource.setDatabaseName("STLEC1");
dataSource.setServerName("v7ec167.svl.ibm.com");
dataSource.setDriverType(4);
dataSource.setPortNumber(446);
java.util.Properties properties = new java.util.Properties();
// Set other properties using
// properties.put("property", "value");
// Supply the user ID and password for the connection
String user = "user";
String password = "password";
// Call getDB2TrustedPooledConnection to get the trusted connection
// instance and the cookie for the connection
Object[] objects = dataSource.getDB2TrustedPooledConnection(
    user, password, properties);
```
**Example:** Reuse an existing trusted connection:

```java
// The first item that was obtained from the previous getDB2TrustedPooledConnection
// call is a connection object. Cast it to a PooledConnection object.
javax.sql.PooledConnection pooledCon =
    (javax.sql.PooledConnection)objects[0];
properties = new java.util.Properties();
// Set new properties for the reused object using
// properties.put("property", "value");
// The second item that was obtained from the previous getDB2TrustedPooledConnection
// call is the cookie for the connection. Cast it as a byte array.
byte[] cookie = ((byte[])(objects[1]);
// Supply the user ID for the new connection.
String newuser = "newuser";
// Supply the name of a mapping service that maps a workstation user
// ID to a z/OS RACF ID
String userRegistry = "registry";
// Do not supply any security token data to be traced.
byte[] userSecTkn = null;
// Do not supply a previous user ID.
String originalUser = null;
// Call getDB2Connection to get the connection object for the new
// user.
java.sql.Connection con =
    ((com.ibm.db2.jcc.DB2PooledConnection)pooledCon).getDB2Connection(
        cookie,newuser,password,userRegistry,userSecTkn,originalUser,properties);
```

---

**IBM Data Server Driver for JDBC and SQLJ support for SSL**

The IBM Data Server Driver for JDBC and SQLJ provides support for the Secure Sockets Layer (SSL) through the Java Secure Socket Extension (JSSE).

You can use SSL support in your Java applications if you use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS Version 9 or later, to DB2 Database for Linux, UNIX, and Windows Version 9.1, Fix Pack 2 or later, or to IBM Informix (IDS) Version 11.50 or later.

If you use SSL support for a connection to a DB2 for z/OS data source, and the z/OS version is V1.8, V1.9, or V1.10, the appropriate PTF for APAR PK72201 must be applied to Communication Server for z/OS IP Services.

To use SSL connections, you need to:

- Configure connections to the data source to use SSL.
- Configure your Java Runtime Environment to use SSL.

**Configuring connections under the IBM Data Server Driver for JDBC and SQLJ to use SSL**

To configure database connections under the IBM Data Server Driver for JDBC and SQLJ to use SSL, you need to set the DB2BaseDataSource.sslConnection property to true.

**Prerequisite:** Before a connection to a data source can use SSL, the port to which the application connects must be configured in the database server as the SSL listener port.

1. Set DB2BaseDataSource.sslConnection on a Connection or DataSource instance.
2. Optional: Set DB2BaseDataSource.ssslTrustStoreLocation on a Connection or DataSource instance to identify the location of the truststore. Setting the ssslTrustStoreLocation property is an alternative to setting the Java
Configuring the Java Runtime Environment to use SSL

Before you can use Secure Sockets Layer (SSL) connections in your JDBC and SQLJ applications, you need to configure the Java Runtime Environment to use SSL.

Before you can configure your Java Runtime Environment for SSL, you need to satisfy the following prerequisites:

- The Java Runtime Environment must include a Java security provider. The IBM JSSE provider or the Sun JSSE provider must be installed. The IBM JSSE provider is automatically installed with the IBM SDK for Java.

  **Restriction:** You can only use the Sun JSSE provider only with a Sun Java Runtime Environment. The Sun JSSE provider does not work with an IBM Java Runtime Environment.

- SSL support must be configured on the database server.

To configure your Java Runtime Environment to use SSL, follow these steps.

1. Import a certificate from the database server to a Java truststore on the client.
   Use the Java keytool utility to import the certificate into the truststore.
   For example, suppose that the server certificate is stored in a file named jcc.cacert. Issue the following keytool utility statement to read the certificate from file jcc.cacert, and store it in a truststore named cacerts.
   ```shell
   keytool -import -file jcc.cacert -keystore cacerts
   ```

2. Configure the Java Runtime Environment for the Java security providers by adding entries to the java.security file.
   The format of a security provider entry is:
   ```plaintext
   security.provider.n=provider-package-name
   ```
   A provider with a lower value of n takes precedence over a provider with a higher value of n.

   The Java security provider entries that you add depend on whether you use the IBM JSSE provider or the Sun JSSE provider.
   - If you use the Sun JSSE provider, add entries for the Sun security providers to your java.security file.
   - If you use the IBM JSSE provider, use one of the following methods:
     - **Use the IBMJSSE2 provider** (supported for the IBM SDK for Java 1.4.2 and later):
**Recommendation:** Use the IBMJSSE2 provider, and use it in FIPS mode.

- If you do not need to operate in FIPS-compliant mode:
  - For the IBM SDK for Java 1.4.2, add an entry for the IBMJSSE2Provider to the java.security file. Ensure that an entry for the IBMJCE provider is in the java.security file. The java.security file that is shipped with the IBM SDK for Java contains an entry for entries for IBMJCE.
  - For later versions of the IBM SDK for Java, ensure that entries for the IBMJSSE2Provider and the IBMJCE provider are in the java.security file. The java.security file that is shipped with the IBM SDK for Java contains entries for those providers.

- If you need to operate in FIPS-compliant mode:
  - Add an entry for the IBMJCEFIPS provider to your java.security file before the entry for the IBMJCE provider. Do not remove the entry for the IBMJCE provider.
  - Enable FIPS mode in the IBMJSSE2 provider. See step 3 on page 185.

  **Use the IBMJSSE provider (supported for the IBM SDK for Java 1.4.2 only):**
  - If you do not need to operate in FIPS-compliant mode, ensure that entries for the IBMJSSEProvider and the IBMJCE provider are in the java.security file. The java.security file that is shipped with the IBM SDK for Java contains entries for those providers.
  - If you need to operate in FIPS-compliant mode, add entries for the FIPS-approved provider IBMJSSEFIPSProvider and the IBMJCEFIPS provider to your java.security file, before the entry for the IBMJCE provider.

**Restriction:** If you use the IBMJSSE provider on the Solaris operating system, you need to include an entry for the SunJSSE provider before entries for the IBMJCE, IBMJCEFIPS, IBMJSSE, or IBMJSSE2 providers.

**Example:** Use a java.security file similar to this one if you need to run in FIPS-compliant mode, and you enable FIPS mode in the IBMJSSE2 provider:

```java
# Set the Java security providers
security.provider.1=com.ibm.jsse2.IBMJSSEProvider2
security.provider.2=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.3=com.ibm.crypto.provider.IBMJCE
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

**Example:** Use a java.security file similar to this one if you need to run in FIPS-compliant mode, and you use the IBMJSSE provider:

```java
# Set the Java security providers
security.provider.1=com.ibm.jsse.IBMJSSEFIPSProvider
security.provider.2=com.ibm.crypto.fips.provider.IBMJCEFIPS
security.provider.3=com.ibm.crypto.provider.IBMJCE
security.provider.4=com.ibm.security.jgss.IBMJGSSProvider
security.provider.5=com.ibm.security.cert.IBMCertPath
security.provider.6=com.ibm.security.sasl.IBMSASL
```

**Example:** Use a java.security file similar to this one if you use the Sun JSSE provider:

```java
# Set the Java security providers
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.rsajca.Provider
security.provider.3=com.sun.crypto.provider.SunJCE
security.provider.4=com.sun.net.ssl.internal.ssl.Provider
```
3. If you plan to use the IBM Data Server Driver for JDBC and SQLJ in FIPS-compliant mode, you need to set the `com.ibm.jsse2.JSSEFIPS` Java system property:

```
com.ibm.jsse2.JSSEFIPS=true
```

**Restriction:** Non-FIPS-mode JSSE applications cannot run in a JVM that is in FIPS mode.

**Restriction:** When the IBMJSSE2 provider runs in FIPS mode, it cannot use hardware cryptography.

4. Configure the Java Runtime Environment for the SSL socket factory providers by adding entries to the `java.security` file.

The format of SSL socket factory provider entries are:

```
ssl.SocketFactory.provider=provider-package-name
ssl.ServerSocketFactory.provider=provider-package-name
```

Specify the SSL socket factory provider for the Java security provider that you are using.

**Example:** Include SSL socket factory provider entries like these in the `java.security` file when you enable FIPS mode in the IBMJSSE2 provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.jsse2.SSLSocketFactoryImpl
ssl.ServerSocketFactory.provider=com.ibm.jsse2.SSLServerSocketFactoryImpl
```

**Example:** Include SSL socket factory provider entries like these in the `java.security` file when you enable FIPS mode in the IBMJSSE provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.ibm.fips.jsse.JSSESocketFactory
ssl.ServerSocketFactory.provider=com.ibm.fips.jsse.JSSEServerSocketFactory
```

**Example:** Include SSL socket factory provider entries like these when you use the Sun JSSE provider:

```
# Set the SSL socket factory provider
ssl.SocketFactory.provider=com.sun.net.ssl.internal.ssl.SSLSocketFactory
ssl.ServerSocketFactory.provider=com.sun.net.ssl.internal.ssl.SSLServerSocketFactory
```

5. Configure Java system properties to use the truststore.

To do that, set the following Java system properties:

**javax.net.ssl.trustStore**

Specifies the name of the truststore that you specified with the
-keystore parameter in the keytool utility in step 1 [on page 183](#).

If the IBM Data Server Driver for JDBC and SQLJ property
DB2BaseDataSource.sslTrustStoreLocation is set, its value overrides the
javax.net.ssl.trustStore property value.

**javax.net.ssl.trustStorePassword (optional)**

Specifies the password for the truststore. You do not need to set a
truststore password. However, if you do not set the password, you
cannot protect the integrity of the truststore.

If the IBM Data Server Driver for JDBC and SQLJ property
DB2BaseDataSource.sslTrustStorePassword is set, its value overrides the
javax.net.ssl.trustStorePassword property value.

**Example:** One way that you can set Java system properties is to specify them as the arguments of the -D option when you run a Java application. Suppose that you want to run a Java application named MySQL.java, which accesses a data source using an SSL connection. You have defined a truststore named cacerts.

The following command sets the truststore name when you run the application.
Security for preparing SQLJ applications with the IBM Data Server Driver for JDBC and SQLJ

Two ways to provide security during SQLJ application preparation are to allow users to customize applications only, and to limit access to a specific set of tables during customization.

Allowing users to customize only

You can use one of the following techniques to allow a set of users to customize SQLJ applications, but not to bind or run those applications:

• **Create a database system for customization only (recommended solution):**
  Follow these steps:
  1. Create a new database manager instance. This is the customization-only system.
  2. On the customization-only system, define all the tables and views that are accessed by the SQLJ applications. The table or view definitions must be the same as the definitions on the database manager instance where the application will be bound and will run (the bind-and-run system). Executing the DESCRIBE statement on the tables or views must give the same results on the customization-only system and the bind-and-run system.
  3. On the customization-only system, grant the necessary table or view privileges to users who will customize SQLJ applications.
  4. On the customization-only system, users run the sqlj command with the -compile=true option to create Java byte codes and serialized profiles for their programs. Then they run the db2sqljcustomize command with the -automaticbind NO option to create customized serialized profiles.
  5. Copy the java byte code files and customized serialized profiles to the bind-and-run system.
  6. A user with authority to bind packages on the bind-and-run system runs the db2sqljbind command on the customized serialized profiles that were copied from the customization-only system.

• **Use a stored procedure to do customization:** Write a Java stored procedure that customizes serialized profiles and binds packages for SQLJ applications on behalf of the end user. This Java stored procedure needs to use a JDBC driver package that was bound with one of the DYNAMICRULES options that causes dynamic SQL to be performed under a different user ID from the end user’s authorization ID. For example, you might use the DYNAMICRULES option DEFINEBIND or DEFINERUN to execute dynamic SQL under the authorization ID of the creator of the Java stored procedure. You need to grant EXECUTE authority on the stored procedure to users who need to do SQLJ customization.

  The stored does the following things:
  1. Receives the compiled SQLJ program and serialized profiles in BLOB input parameters
  2. Copies the input parameters to its file system
  3. Runs db2sqljcustomize to customize the serialized profiles and bind the packages for the SQLJ program
  4. Returns the customized serialized profiles in output parameters

• **Use a stand-alone program to do customization:** This technique involves writing a program that performs the same steps as a Java stored procedure that
customizes serialized profiles and binds packages for SQLJ applications on behalf of the end user. However, instead of running the program as a stored procedure, you run the program as a stand-alone program under a library server.

**Restricting table access during customization**

When you customize serialized profiles, you should do online checking, to give the application program information about the data types and lengths of table columns that the program accesses. By default, customization includes online checking.

Online checking requires that the user who customizes a serialized profile has authorization to execute PREPARE and DESCRIBE statements against SQL statements in the SQLJ program. That authorization includes the SELECT privilege on tables and views that are accessed by the SQL statements. If SQL statements contain unqualified table names, the qualifier that is used during online checking is the value of the `db2sqljcustomize -qualifier` parameter. Therefore, for online checking of tables and views with unqualified names in an SQLJ application, you can grant the SELECT privilege only on tables and views with a qualifier that matches the value of the `qualifier` parameter.
Chapter 6. Security under the DB2 JDBC Type 2 Driver

The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) supports user ID and password security.

You must set the user ID and the password, or set neither. If you do not set a user ID and password, the driver uses the user ID and password of the user who is currently logged on to the operating system.

To specify user ID and password security for a JDBC connection, use one of the following techniques.

**For the DriverManager interface:** you can specify the user ID and password directly in the DriverManager.getConnection invocation. For example:

```java
import java.sql.*; // JDBC base
... String id = "db2adm"; // Set user ID String pw = "db2adm"; // Set password String url = "jdbc:db2:toronto"; // Set URL for the data source Connection con = DriverManager.getConnection(url, id, pw); // Create connection
```

Alternatively, you can set the user ID and password by setting the user and password properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example:

```java
import java.sql.*; // JDBC base import COM.ibm.db2.jdbc.*; // DB2 JDBC Type 2 driver ... Properties properties = new java.util.Properties(); // Create Properties object properties.put("user", "db2adm"); // Set user ID for the connection properties.put("password", "db2adm"); // Set password for the connection String url = "jdbc:db2:toronto"; // Set URL for the data source Connection con = DriverManager.getConnection(url, properties); // Create connection
```

**For the DataSource interface:** you can specify the user ID and password directly in the DataSource.getConnection invocation. For example:

```java
import java.sql.*; // JDBC base import COM.ibm.db2.jdbc.*; // DB2 JDBC Type 2 driver ... Context ctx = new InitialContext(); // Create context for JNDI DataSource ds = (DataSource) ctx.lookup("jdbc/sampledb"); // Get DataSource object String id = "db2adm"; // Set user ID String pw = "db2adm"; // Set password Connection con = ds.getConnection(id, pw); // Create connection
```

Alternatively, if you create and deploy the DataSource object, you can set the user ID and password by invoking the DataSource.setUser and DataSource.setPassword methods after you create the DataSource object. For example:
import java.sql.*; // JDBC base
import COM.ibm.db2.jdbc.*; // DB2 JDBC Type 2 driver
// implementation of JDBC
...

DB2DataSource db2ds = new DB2DataSource(); // Create DataSource object
db2ds.setDatabaseName("toronto"); // Set location
db2ds.setUser("db2adm"); // Set user ID
db2ds.setPassword("db2adm"); // Set password
Chapter 7. Building Java database applications

You can build JDBC and SQLJ database applications manually. Alternatively, you can use a Java makefile to build JDBC applications, and use the bldsqlj build file that is shipped with DB2 Database for Linux, UNIX, and Windows to build SQLJ applications.

Building JDBC applets

You can use a Java makefile or manually execute the javac command to build JDBC applications.

The following steps demonstrate how to build and run the Applt.java sample JDBC applet.

1. Compile Applt.java to produce the file Applt.class with this command:
   javac Applt.java

2. Ensure that your working directory is accessible by your web browser, or by your Java applet viewer, if you are using it. If your directory is not accessible, copy the following files into a directory that is accessible:
   • Applt.html
   • Applt.class

3. Copy sqllib\java\db2jcc.jar on Windows or sqllib/java/db2jcc.jar on UNIX, into the same directory as Applt.class and Applt.html.
   If you are using any JDBC 4.0 functions, copy db2jcc4.jar instead of db2jcc.jar.

4. If you are using the IBM Data Server Driver for JDBC and SQLJ, connect with that driver by modifying the Applt.html file according to the instructions in the file. For the TCP/IP port number, you should use the database port number 50000.

5. To run this applet, either ensure that a web server is installed and running on your DB2 machine (server or client), or you can use the applet viewer that comes with the SDK for Java by entering the following command in the working directory of your client machine:
   appletviewer Applt.html

Building JDBC applications

You can use a Java makefile or manually execute the javac command to build JDBC applications.

The following steps demonstrate how to build and run the DbInfo sample JDBC application.

1. Compile DbInfo.java to produce the file DbInfo.class with this command:
   javac DbInfo.java

2. If you are running a Java application on UNIX in a 64-bit DB2 instance but the software development kit for Java is 32-bit, you need to change the DB2 library path before running the application. For example, on AIX:

   • If using bash or Korn shell:
     export LIBPATH=$HOME/sqllib/lib32
   • If using C shell:
     setenv LIBPATH $HOME/sqllib/lib32
3. Run the Java interpreter on the application with this command:

```
java DbInfo
```

## Building JDBC routines

You can use a Java `makefile` or the `javac` command to build JDBC routines. After you build those routines, you need to catalog them.

The following steps demonstrate how to build and run these routines:

- The `SpServer` sample JDBC stored procedure
- The `UDFsrv` sample user-defined function, which has no SQL statements
- The `UDFsqslsv` sample user-defined function, which has SQL statements
- To build and run the `SpServer` stored procedure:

1. Compile `SpServer.java` to produce the file `SpServer.class` with this command:

   ```
   javac SpServer.java
   ```

2. Copy `SpServer.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on UNIX.

3. Catalog the routines by running the `spcat` script on the server. The `spcat` script connects to the sample database, uncatalogs the routines if they were previously cataloged by calling `SpDrop.db2`, then catalogs them by calling `SpCreate.db2`, and finally disconnects from the database. You can also run the `SpDrop.db2` and `SpCreate.db2` scripts individually.

4. Stop and restart the database to allow the new class file to be recognized. If necessary, set the file mode for the class file to "read" so it is readable by the fenced user.

5. Compile and run the `SpClient` client application to access the stored procedure class.

- To build and run the `UDFsrv.java` user-defined function program (user-defined function with no SQL statements) on the server, from the command line:

1. Compile `UDFsrv.java` to produce the file `UDFsrv.class` with this command:

   ```
   javac UDFsrv.java
   ```

2. Copy `UDFsrv.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on UNIX.

3. Compile and run a client program that calls `UDFsrv`.

   To access the `UDFsrv` library, you can use the `UDFcli.java` JDBC application, or the `UDFcli.sqlj` SQLJ client application. Both versions of the client program contain the `CREATE FUNCTION` SQL statement that you use to register the user-defined functions with the database, and also contain SQL statements that use the user-defined functions.

- To build and run the `UDFsqslsv.java` user-defined function program (user-defined function with SQL statements) on the server, from the command line:

1. Compile `UDFsqslsv.java` to produce the file `UDFsqslsv.class` with this command:

   ```
   javac UDFsqslsv.java
   ```

2. Copy `UDFsqslsv.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on UNIX.

3. Compile and run a client program that calls `UDFsqslsv`.

   To access the `UDFsqslsv` library, you can use the `UDFsqlcl.java` JDBC application. The client program contains the `CREATE FUNCTION` SQL
statement that you use to register the user-defined functions with the
database, and also contains SQL statements that use the user-defined
functions.

Building SQLJ applets

You can use a Java makefile or the bldsqlj build file to build SQLJ applets.

The following steps demonstrate how to build and run the Applt sample SQLJ
applet. These steps use the build file, bldsqlj (UNIX), or bldsqlj.bat (Windows),
which contains commands to build either an SQLJ applet or application.

The build file takes up to six parameters: $1, $2, $3, $4, $5, and $6 on UNIX, and
%1, %2, %3, %4, %5, and %6 on Windows. The first parameter specifies the name
of your program. The second parameter specifies the user ID for the database
instance, the third parameter specifies the password. The fourth parameter
specifies the server name. The fifth parameter specifies the port number. And the
sixth parameter specifies the database name. For all but the first parameter,
program name, default values can be used. See the build file for details about
using default parameter values.

1. Build the applet with this command:
   
   bldsqlj Applt <userid> <password> <server_name> <port_number> <db_name>

2. Ensure that your working directory is accessible by your web browser, or by
   your Java applet viewer, if you are using it. If your directory is not accessible,
copy the following files into a directory that is accessible:
   • Applt.html
   • Applt.class
   • Applt_Cursor1.class
   • Applt_Cursor2.class
   • Applt_SJProfileKeys.class
   • Applt_SJProfile0.ser

3. Copy sqllib\java\db2jcc.jar on Windows or sqllib/java/db2jcc.jar on
   UNIX, into the same directory as Applt.class and Applt.html.
   If you are using any JDBC 4.0 functions, copy db2jcc4.jar instead of db2jcc.jar.

4. If you are using the IBM Data Server Driver for JDBC and SQLJ, connect with
   that driver by modifying the Applt.html file according to the instructions in the
   file. For the TCP/IP port number, you should use the database port number
   50000.

5. To run this applet, either ensure that a web server is installed and running on
   your DB2 machine (server or client), or you can use the applet viewer that
   comes with the SDK for Java by entering the following command in the
   working directory of your client machine:
      appletviewer Applt.html

Building SQLJ applications

You can use a Java makefile or the bldsqlj build file to build SQLJ applications.

The following steps demonstrate how to build and run the TbMod sample SQLJ
application. These steps use the build file, bldsqlj (UNIX), or bldsqlj.bat
(Windows), which contains commands to build either an SQLJ applet or
application.
The build file takes up to six parameters: $1, $2, $3, $4, $5, and $6 on UNIX, and %1, %2, %3, %4, %5, and %6 on Windows. The first parameter specifies the name of your program. The second parameter specifies the user ID for the database instance, the third parameter specifies the password. The fourth parameter specifies the server name. The fifth parameter specifies the port number. And the sixth parameter specifies the database name. For all but the first parameter, program name, default values can be used. See the build file for details about using default parameter values.

1. Build the application with this command:
   ```
   bldsqlj TbMod <userid> <password> <server_name> <port_number> <db_name>
   ```

2. If you are running a Java application on UNIX in a 64-bit DB2 instance but the software development kit for Java is 32-bit, you need to change the DB2 library path before running the application. For example, on AIX:
   - If using bash or Korn shell:
     ```
     export LIBPATH=$HOME/sqllib/lib32
     ```
   - If using C shell:
     ```
     setenv LIBPATH $HOME/sqllib/lib32
     ```

3. Run the Java interpreter on the application with this command:
   ```
   java TbMod
   ```

**Java applet considerations**

DB2 databases can be accessed by using Java applets.

Keep the following points in mind when using them:
- For a larger JDBC or SQLJ applet that consists of several Java classes, you might choose to package all its classes in a single JAR file. For an SQLJ applet, you would also have to package its serialized profiles along with its classes. If you choose to do this, add your JAR file into the archive parameter in the "applet" tag. For details, see the documentation for your software development kit for Java.

For SQLJ applets, some browsers do not yet have support for loading a serialized object from a resource file associated with the applet. For example, you will get the following error message when trying to load the supplied sample applet Applt in those browsers:

```
java.lang.ClassNotFoundException: Applt_SJProfile0
```

As a workaround, there is a utility which converts a serialized profile into a profile stored in Java class format. The utility is a Java class called `sqlj.runtime.profile.util.SerProfileToClass`. It takes a serialized profile resource file as input and produces a Java class containing the profile as output. Your profile can be converted using one of the following commands:

```
profconv Applt_SJProfile0.ser
```

or

```
java sqlj.runtime.profile.util.SerProfileToClass Applt_SJProfile0.ser
```

The class `Applt_SJProfile0.class` is created as a result. Replace all profiles in .ser format used by the applet with profiles in .class format, and the problem should go away.
- You can place the file `db2jcc.jar` into a directory that is shared by several applets that might be loaded from your Web site. `db2jcc.jar` is for applets using the IBM Data Server Driver for JDBC and SQLJ or for any SQLJ applet. This file
is in the sql1lib\java directory on Windows operating systems, and in the
sql1lib/java directory on UNIX. You might need to add a codebase parameter
into the "applet" tag in the HTML file to identify the directory. For details, see
the documentation for your software development kit for Java.
If you are using any JDBC 4.0 functions, copy db2jcc4.jar instead of db2jcc.jar.

- The JDBC applet server (listener), db2jd, contains signal handling to make it
  more robust. As a result, you cannot use the CTRL-C key sequence to terminate
db2jd. Therefore, the only way to terminate the listener is to kill the process by
using kill -9 (for UNIX) or the Task Manager (for Windows).

---

**SQLJ application and applet options for UNIX**

The bldsqlj build script builds SQLJ applications and applets on UNIX operating
systems. bldsqlj specifies a set of SQLJ translator and customizer options.

**Recommendation:** Use the same SQLJ translator and customizer options that
bldsqlj uses when you build your SQLJ applications and applets on UNIX
platforms.

The options that bldsqlj includes are:

- `sqlj`  The SQLJ translator (also compiles the program).
- `${profilename}.sqlj`  The SQLJ source file. The `profilename=${1%.sqlj}` command removes the
extension if it was included in the input file name, so when the extension
is added back again, it is not duplicated.
- `db2sqljcustomize`  The SQLJ profile customizer.
- `-url`  Specifies a JDBC URL for establishing a database connection, such as
  `jdbc:db2://servername:50000/sample`.
- `-user`  Specifies a user ID.
- `-password`  Specifies a password.
- `${profilename}_SJProfile0`  Specifies a serialized profile for the program.

---

**SQLJ application and applet options for Windows**

The bldsqlj.bat batch file builds SQLJ applications and applets on Windows
operating systems. bldsqlj.bat specifies a set of SQLJ translator and customizer
options.

**Recommendation:** Use the same SQLJ translator and customizer options that
bldsqlj.bat uses when you build your SQLJ applications and applets on Windows
operating systems.

The options that bldsqlj.bat includes are:

- `sqlj`  The SQLJ translator (also compiles the program).
- `%1.sqlj`  The SQLJ source file.
db2sqljcustomize
The SQLJ profile customizer.
-url
Specifies a JDBC URL for establishing a database connection, such as
jdbc:db2://servername:50000/sample.
-user
Specifies a user ID.
-password
Specifies a password.
%1_SJProfile0
Specifies a serialized profile for the program.

Building SQL routines
You can use a Java makefile or the bldsqljs build file to build SQLJ routines.
After you build those routines, you need to catalog them.

The following steps demonstrate how to build and run the SpServer sample SQLJ
stored procedure. These steps use the build file, bldsqljs (UNIX), or bldsqljs.bat
(Windows), which contains commands to build either an SQLJ applet or
application.

The build file takes up to six parameters: $1, $2, $3, $4, $5, and $6 on UNIX, and
%1, %2, %3, %4, %5, and %6 on Windows. The first parameter specifies the name
of your program. The second parameter specifies the user ID for the database
instance, the third parameter specifies the password. The fourth parameter
specifies the server name. The fifth parameter specifies the port number. And the
sixth parameter specifies the database name. For all but the first parameter,
program name, default values can be used. See the build file for details about
using default parameter values.

1. Build the stored procedure application with this command:
   bldsqljs SpServer <userid> <password> <server_name> <port_number> <db_name>
2. Catalog the stored procedure with this command:
   spcat
   This script connects to the sample database, uncatalogs the routines if they
   were previously cataloged by calling SpDrop.db2, then catalogs them by calling
   SpCreate.db2, and finally disconnects from the database. You can also run the
   SpDrop.db2 and SpCreate.db2 scripts individually.
3. Stop and restart the database to allow the new class file to be recognized. If
   necessary, set the file mode for the class file to read, so it is readable by the
   fenced user.
4. Compile and run the SpClient client application to access the stored procedure
   class. You can build SpClient with the application build file, bldsqlj (UNIX) or
   bldsqlj.bat (Windows).

SQLJ routine options for UNIX
The bldsqljs build script builds SQLJ routines on UNIX operating systems.
bldsqljs specifies a set of SQLJ translator and customizer options.

Recommendation: Use the same SQLJ translator and customizer options that
bldsqljs uses when you build your SQLJ routines on UNIX platforms.
The options that bldsqljs includes are:

**sqlj**  The SQLJ translator (also compiles the program).

`${proname}.sqlj`

The SQLJ source file. The `proname=${1%.sqlj}` command removes the extension if it was included in the input file name, so when the extension is added back again, it is not duplicated.

**db2sqljcustomize**

The SQLJ profile customizer.

- **-url**  Specifies a JDBC URL for establishing a database connection, such as `jdbc:db2://servername:50000/sample`.

- **-user**  Specifies a user ID.

- **-password**  Specifies a password.

`${proname}_SJProfile0`

Specifies a serialized profile for the program.

---

**SQLJ routine options for Windows**

The `bldsqljs.bat` batch file builds SQLJ routines on Windows operating systems.

*bldsqljs.bat* specifies a set of SQLJ translator and customizer options.

**Recommendation**: Use the same SQLJ translator and customizer options that `bldsqljs.bat` uses when you build your SQLJ routines on Windows operating systems.

The following SQLJ translator and customizer options are used in the `bldsqljs.bat` batch file on Windows operating systems. These are the options DB2 recommends that you use to build SQLJ routines (stored procedures and user-defined functions).

**sqlj**  The SQLJ translator (also compiles the program).

%1.sqlj

The SQLJ source file.

**db2sqljcustomize**

The DB2 for Java profile customizer.

- **-url**  Specifies a JDBC URL for establishing a database connection, such as `jdbc:db2://servername:50000/sample`.

- **-user**  Specifies a user ID.

- **-password**  Specifies a password.

%1_SJProfile0

Specifies a serialized profile for the program.
Chapter 8. Problem diagnosis with the IBM Data Server Driver for JDBC and SQLJ

To obtain data for diagnosing SQLJ or JDBC problems with the IBM Data Server Driver for JDBC and SQLJ, collect trace data and run utilities that format the trace data.

You should run the trace and diagnostic utilities only under the direction of IBM software support.

Collecting JDBC trace data

Use one of the following procedures to start the trace:

Procedure 1: For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity for DB2 for Linux, UNIX and Windows, the recommended method is to start the trace by setting the db2.jcc.override.traceFile property or the db2.jcc.override.traceDirectory property in the IBM Data Server Driver for JDBC and SQLJ configuration properties file. You can set the db2.jcc.tracePolling and db2.jcc.tracePollingInterval properties before you start the driver to allow you to change global configuration trace properties while the driver is running.

Procedure 2: If you use the DataSource interface to connect to a data source, follow this method to start the trace:
1. Invoke the DB2BaseDataSource.setTraceLevel method to set the type of tracing that you need. The default trace level is TRACE_ALL. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information on how to specify more than one type of tracing.
2. Invoke the DB2BaseDataSource.setJccLogWriter method to specify the trace destination and turn the trace on.

Procedure 3:

If you use the DataSource interface to connect to a data source, invoke the javax.sql.DataSource.setLogWriter method to turn the trace on. With this method, TRACE_ALL is the only available trace level.

If you use the DriverManager interface to connect to a data source, follow this procedure to start the trace.
1. Invoke the DriverManager.getConnection method with the traceLevel property set in the info parameter or url parameter for the type of tracing that you need. The default trace level is TRACE_ALL. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for information on how to specify more than one type of tracing.
2. Invoke the DriverManager.setLogWriter method to specify the trace destination and turn the trace on.

After a connection is established, you can turn the trace off or back on, change the trace destination, or change the trace level with the DB2Connection.setJccLogWriter method. To turn the trace off, set the logWriter value to null.
The logWriter property is an object of type java.io.PrintWriter. If your application cannot handle java.io.PrintWriter objects, you can use the traceFile property to specify the destination of the trace output. To use the traceFile property, set the logWriter property to null, and set the traceFile property to the name of the file to which the driver writes the trace data. This file and the directory in which it resides must be writable. If the file already exists, the driver overwrites it.

Procedure 4: If you are using the DriverManager interface, specify the traceFile and traceLevel properties as part of the URL when you load the driver. For example:

```java
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose" + 
   "traceFile=/u/db2p/jcctrace;" + 
   "traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS + ";";
```

Procedure 5: Use DB2TraceManager methods. The DB2TraceManager class provides the ability to suspend and resume tracing of any type of log writer.

Example of starting a trace using configuration properties: For a complete example of using configuration parameters to collect trace data, see "Example of using configuration properties to start a JDBC trace".

Trace example program: For a complete example of a program for tracing under the IBM Data Server Driver for JDBC and SQLJ, see "Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ".

**Collecting SQLJ trace data during customization or bind**

To collect trace data to diagnose problems during the SQLJ customization or bind process, specify the -tracelevel and -tracefile options when you run the db2sqljcustomize or db2sqljbind bind utility.

**Formatting information about an SQLJ serialized profile**

The profp utility formats information about each SQLJ clause in a serialized profile. The format of the profp utility is:

```
<---profp--serialized-profile-name---
```

Run the profp utility on the serialized profile for the connection in which the error occurs. If an exception is thrown, a Java stack trace is generated. You can determine which serialized profile was in use when the exception was thrown from the stack trace.

**Formatting information about an SQLJ customized serialized profile**

The db2sqljprint utility formats information about each SQLJ clause in a serialized profile that is customized for the IBM Data Server Driver for JDBC and SQLJ.

Run the db2sqljprint utility on the customized serialized profile for the connection in which the error occurs.
Example of using configuration properties to start a JDBC trace

You can control tracing of JDBC applications without modifying those applications.

Suppose that you want to collect trace data for a program named Test.java, which uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. Test.java does no tracing, and you do not want to modify the program, so you enable tracing using configuration properties. You want your trace output to have the following characteristics:

- Trace information for each connection on the same DataSource is written to a separate trace file. Output goes into a directory named /Trace.
- Each trace file name begins with jccTrace1.
- If trace files with the same names already exist, the trace data is appended to them.

Although Test1.java does not contain any code to do tracing, you want to set the configuration properties so that if the application is modified in the future to do tracing, the settings within the program will take precedence over the settings in the configuration properties. To do that, use the set of configuration properties that begin with db2.jcc, not db2.jcc.override.

The configuration property settings look like this:

- `db2.jcc.override.traceDirectory=/Trace`
- `db2.jcc.traceFile=jccTrace1`
- `db2.jcc.traceFileAppend=true`

You want the trace settings to apply only to your stand-alone program Test1.java, so you create a file with these settings, and then refer to the file when you invoke the Java program by specifying the -Ddb2.jcc.propertiesFile option. Suppose that the file that contains the settings is /Test/jcc.properties. To enable tracing when you run Test1.java, you issue a command like this:

```
java -Ddb2.jcc.propertiesFile=/Test/jcc.properties Test1
```

Suppose that Test1.java creates two connections for one DataSource. The program does not define a logWriter object, so the driver creates a global logWriter object for the trace output. When the program completes, the following files contain the trace data:

- `/Trace/jccTrace1_global_0`
- `/Trace/jccTrace1_global_1`

Example of a trace program under the IBM Data Server Driver for JDBC and SQLJ

You might want to write a single class that includes methods for tracing under the DriverManager interface, as well as the DataSource interface.

The following example shows such a class. The example uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

```
public class TraceExample
{

    public static void main(String[] args)
```

Figure 48. Example of tracing under the IBM Data Server Driver for JDBC and SQLJ
private static void sampleConnectUsingSimpleDataSource()
{
    java.sql.Connection c = null;
    java.io.PrintWriter printWriter =
        new java.io.PrintWriter(System.out, true);
        // Prints to console, true means
        // auto-flush so you don't lose trace
    try {
        javax.sql.DataSource ds =
            new com.ibm.db2.jcc.DB2SimpleDataSource();
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setPortNumber(5021);
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDatabaseName("san_jose");
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).setDriverType(4);
            ds.setLogWriter(printWriter); // This turns on tracing
            // Refine the level of tracing detail
            ((com.ibm.db2.jcc.DB2BaseDataSource) ds).
                setTraceLevel(com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_CONNECTS | com.ibm.db2.jcc.DB2SimpleDataSource.TRACE_DRDA_FLOWS);
            // This connection request is traced using trace level
            // TRACE_CONNECTS | TRACE_DRDA_FLOWS
            c = ds.getConnection("myname", "mypass");
            // Change the trace level to TRACE_ALL
            // for all subsequent requests on the connection
            ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter,
                com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);
            // The following INSERT is traced using trace level TRACE_ALL
            java.sql.Statement s1 = c.createStatement();
            s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
            s1.close();
            // This code disables all tracing on the connection
            ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);
            // The following INSERT statement is not traced
            java.sql.Statement s2 = c.createStatement();
            s2.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
            s2.close();
            c.close();
    } catch(java.sql.SQLException e) {
        com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(e,
            printWriter, "[TraceExample]");
    }
    finally {
        cleanup(c, printWriter);
        printWriter.flush();
    }
}

// If the code ran successfully, the connection should
// already be closed. Check whether the connection is closed.
// If so, just return.
// If a failure occurred, try to roll back and close the connection.
private static void cleanup(java.sql.Connection c,
    java.io.PrintWriter printWriter)
if (c == null) return;

try {
    if (c.isClosed()) {
        PrintWriter.println("[TraceExample] " +
                       "The connection was successfully closed");
        return;
    }
}

// If we get to here, something has gone wrong.
// Roll back and close the connection.
printWriter.println("[TraceExample] Rolling back the connection");
try {
    c.rollback();
}

catch(java.sql.SQLException e) {
    PrintWriter.println("[TraceExample] " +
                       "Trapped the following java.sql.SQLException while trying to roll back:");
    com.ibm.db2.jdbc.DB2ExceptionFormatter.printStackTrace(e, PrintWriter,
               "[TraceExample]");
    PrintWriter.println("[TraceExample] " +
                       "Unable to roll back the connection");
}

catch(java.lang.Throwable e) {
    PrintWriter.println("[TraceExample] Trapped the " +
                       "following java.lang.Throwable while trying to roll back:");
    com.ibm.db2.jdbc.DB2ExceptionFormatter.printStackTrace(e, PrintWriter,
               "[TraceExample]");
    PrintWriter.println("[TraceExample] Unable to " +
                       "roll back the connection");
}

// Close the connection
printWriter.println("[TraceExample] Closing the connection");
try {
    c.close();
}

catch(java.sql.SQLException e) {
    PrintWriter.println("[TraceExample] Exception while " +
                       "trying to close the connection");
    PrintWriter.println("[TraceExample] Deadlocks could " +
                       "occur if the connection is not closed.");
    com.ibm.db2.jdbc.DB2ExceptionFormatter.printStackTrace(e, PrintWriter,
               "[TraceExample]");
}

catch(java.lang.Throwable e) {
    PrintWriter.println("[TraceExample] Throwable caught " +
                       "while trying to close the connection");
    PrintWriter.println("[TraceExample] Deadlocks could " +
                       "occur if the connection is not closed.");
    com.ibm.db2.jdbc.DB2ExceptionFormatter.printStackTrace(e, PrintWriter,
               "[TraceExample]");
}

catch(java.lang.Throwable e) {
    PrintWriter.println("[TraceExample] Unable to " +
                       "force the connection to close");
    PrintWriter.println("[TraceExample] Deadlocks " +
                       "could occur if the connection is not closed.");
    com.ibm.db2.jdbc.DB2ExceptionFormatter.printStackTrace(e, PrintWriter,
               "[TraceExample]");
}

private static void sampleConnectWithURLUsingDriverManager() {
    java.sql.Connection c = null;
This time, send the PrintWriter to a file.
java.io.PrintWriter printWriter = null;
try {
    printWriter =
        new java.io.PrintWriter(
            new java.io.BufferedOutputStream( 
                new java.io.FileOutputStream("/temp/driverLog.txt"), 4096), true);
} catch(java.io.FileNotFoundException e) {
    java.lang.System.err.println("Unable to establish a print writer for trace");
    java.lang.System.err.flush();
    return;
}
try {
    Class.forName("com.ibm.db2.jcc.DB2Driver");
} catch(ClassNotFoundException e) {
    printWriter.println("[TraceExample] " +
        "IBM Data Server Driver for JDBC and SQLJ type 4 connectivity " +
        "is not in the application classpath. Unable to load driver.");
    printWriter.flush();
    return;
}
// This URL describes the target data source for Type 4 connectivity.
// The traceLevel property is established through the URL syntax,
// and driver tracing is directed to file "/temp/driverLog.txt"
// The traceLevel property has type int. The constants
// com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS and
// com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS represent
// int values. Those constants cannot be used directly in the
// first getConnection parameter. Resolve the constants to their
// int values by assigning them to a variable. Then use the
// variable as the first parameter of the getConnection method.
String databaseURL =
    "jdbc:db2://sysmvs1.stl.ibm.com:5021" +
    "/sample:traceFile="/temp/driverLog.txt";traceLevel=" +
    (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS +
    com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS) + ";";
// Set other properties
java.util.Properties properties = new java.util.Properties();
properties.setProperty("user", "myname");
properties.setProperty("password", "mypass");
try {
    // This connection request is traced using trace level
    // TRACE_CONNECTS | TRACE_DRDA_FLOWS
    c = java.sql.DriverManager.getConnection(databaseURL, properties);
    // Change the trace level for all subsequent requests
    // on the connection to TRACE_ALL
    ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(printWriter, 
    com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL);
    // The following INSERT is traced using trace level TRACE_ALL
    java.sql.Statement s1 = c.createStatement();
    s1.executeUpdate("INSERT INTO sampleTable(sampleColumn) VALUES(1)");
    s1.close();
    // Disable all tracing on the connection
    ((com.ibm.db2.jcc.DB2Connection) c).setJccLogWriter(null);
    // The following SQL insert code is not traced
    java.sql.Statement s2 = c.createStatement();
Techniques for monitoring IBM Data Server Driver for JDBC and SQLJ
Sysplex support

To monitor IBM Data Server Driver for JDBC and SQLJ Sysplex support, you need to monitor the global transport objects pool.

You can monitor the global transport objects pool in either of the following ways:

- Using traces that you start by setting IBM Data Server Driver for JDBC and SQLJ configuration properties
- Using an application programming interface

Configuration properties for monitoring the global transport objects pool


For example, the following set of configuration property settings cause error messages and dump pool error messages to be written every 60 seconds to a file named `/home/WAS/logs/srv1/poolstats`:

- `db2.jcc.dumpPool=DUMP_SYSPLEX_MSG|DUMP_POOL_ERROR`
- `db2.jcc.dumpPoolStatisticsOnSchedule=60`
- `db2.jcc.dumpPoolStatisticsOnScheduleFile=/home/WAS/logs/srv1/poolstats`

An entry in the pool statistics file looks like this:

```
```

The meanings of the fields are:

- **npr**: The total number of requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created.
- **nsr**: The number of successful requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created. A successful request means that the pool returned an object.
- **lwroc**: The number of objects that were reused but were not in the pool. This can happen if a Connection object releases a transport object at a transaction...
boundary. If the Connection object needs a transport object later, and the original transport object has not been used by any other Connection object, the Connection object can use that transport object.

**hwroc**  
The number of objects that were reused from the pool.

**coc**  
The number of objects that the IBM Data Server Driver for JDBC and SQLJ created since the pool was created.

**aoooc**  
The number of objects that exceeded the idle time that was specified by db2.jcc.maxTransportObjectIdleTime and were deleted from the pool.

**rmoc**  
The number of objects that have been deleted from the pool since the pool was created.

**nbr**  
The number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached its maximum capacity. A blocked request might be successful if an object is returned to the pool before the db2.jcc.maxTransportObjectWaitTime is exceeded and an exception is thrown.

**tbt**  
The total time in milliseconds for requests that were blocked by the pool. This time can be much larger than the elapsed execution time of the application if the application uses multiple threads.

**sbt**  
The shortest time in milliseconds that a thread waited to get a transport object from the pool. If the time is under one millisecond, the value in this field is zero.

**lbt**  
The longest time in milliseconds that a thread waited to get a transport object from the pool.

**abt**  
The average amount of time in milliseconds that threads waited to get a transport object from the pool. This value is tbt/nbr.

**tpo**  
The number of objects that are currently in the pool.

**Application programming interfaces for monitoring the global transport objects pool**

You can write applications to gather statistics on the global transport objects pool. Those applications create objects in the DB2PoolMonitor class and invoke methods to retrieve information about the pool.

For example, the following code creates an object for monitoring the global transport objects pool:

```java
import com.ibm.db2.jcc.DB2PoolMonitor;
DB2PoolMonitor transportObjectPoolMonitor =
    DB2PoolMonitor.getPoolMonitor (DB2PoolMonitor.TRANSPORT_OBJECT);
```
After you create the DB2PoolMonitor object, you can use methods in the DB2PoolMonitor class to monitor the pool.
Chapter 9. System monitoring for the IBM Data Server Driver for JDBC and SQLJ

To assist you in monitoring the performance of your applications with the IBM Data Server Driver for JDBC and SQLJ, the driver provides two methods to collect information for a connection.

That information is:

Core driver time
The sum of elapsed monitored API times that were collected while system monitoring was enabled, in microseconds. In general, only APIs that might result in network I/O or database server interaction are monitored.

Network I/O time
The sum of elapsed network I/O times that were collected while system monitoring was enabled, in microseconds.

Server time
The sum of all reported database server elapsed times that were collected while system monitoring was enabled, in microseconds.

Application time
The sum of the application, JDBC driver, network I/O, and database server elapsed times, in milliseconds.

The two methods are:
• The DB2SystemMonitor interface
• The TRACE_SYSTEM_MONITOR trace level

To collect system monitoring data using the DB2SystemMonitor interface: Perform these basic steps:
1. Invoke the DB2Connection.getDB2SystemMonitor method to create a DB2SystemMonitor object.
2. Invoke the DB2SystemMonitor.enable method to enable the DB2SystemMonitor object for the connection.
3. Invoke the DB2SystemMonitor.start method to start system monitoring.
4. When the activity that is to be monitored is complete, invoke DB2SystemMonitor.stop to stop system monitoring.
5. Invoke the DB2SystemMonitor.getCoreDriverTimeMicros, DB2SystemMonitor.getNetworkIOTimeMicros, DB2SystemMonitor.getServerTimeMicros, or DB2SystemMonitor.getApplicationTimeMillis methods to retrieve the elapsed time data.

For example, the following code demonstrates how to collect each type of elapsed time data. The numbers to the right of selected statements correspond to the previously described steps.
To collect system monitoring information using the trace method:

Start a JDBC trace, using configuration properties or Connection or DataSource properties. Include TRACE_SYSTEM_MONITOR when you set the traceLevel property. For example:

```java
import java.sql.*;
import com.ibm.db2.jcc.*;
public class TestSystemMonitor
{
    public static void main(String[] args)
    {
        String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose"
        String user="db2adm"
        String password="db2adm"
        try
        {
            // Load the IBM Data Server Driver for JDBC and SQLJ
            Class.forName("com.ibm.db2.jcc.DB2Driver");
            System.out.println("**** Loaded the JDBC driver");

            // Create the connection using the IBM Data Server Driver for JDBC and SQLJ
            Connection conn = DriverManager.getConnection(url, user, password);
            System.out.println("**** Created a JDBC connection to the data source");

            DB2SystemMonitor systemMonitor = ((DB2Connection)conn).getDB2SystemMonitor();
            systemMonitor.enable(true);
            systemMonitor.start(DB2SystemMonitor.RESET_TIMES);

            Statement stmt = conn.createStatement();
            int numUpd = stmt.executeUpdate("UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
            systemMonitor.stop();

            System.out.println("Server elapsed time (microseconds)=" + systemMonitor.getServerTimeMicros());
            System.out.println("Network I/O elapsed time (microseconds)=" + systemMonitor.getNetworkIOTimeMicros());
            System.out.println("Core driver elapsed time (microseconds)=" + systemMonitor.getCoreDriverTimeMicros());
            System.out.println("Application elapsed time (milliseconds)=" + systemMonitor.getApplicationTimeMillis());

            conn.rollback();
            stmt.close();
            conn.close();
        }
        catch(ClassNotFoundException e)
        {
            System.err.println("Unable to load the driver," + e);
        }
        catch(SQLException e)
        {
            System.out.println("SQLException:" + e + e.printStackTrace();
        }
    }
}
```

**Figure 49. Example of using DB2SystemMonitor methods to collect system monitoring data**

To collect system monitoring information using the trace method: Start a JDBC trace, using configuration properties or Connection or DataSource properties. Include TRACE_SYSTEM_MONITOR when you set the traceLevel property. For example:

```java
```

The trace records with system monitor information look similar to this:
IBM Data Server Driver for JDBC and SQLJ remote trace controller

The IBM Data Server Driver for JDBC and SQLJ provides a facility for controlling IBM Data Server Driver for JDBC and SQLJ traces dynamically.

This remote trace controller lets you perform operations like these for multiple driver instances:
- Start, stop, or resume a trace
- Change the output trace file or directory location
- Change the trace level

The remote trace controller uses the Java Management Extensions (JMX) architecture, which is part of the Java Standard Edition, Version 6, or later. The JMX consists of:
- A set of built-in management utilities, which let you do monitoring from a management console such as the Java Monitoring and Management Console (JConsole).
- A set of APIs that let you write applications to perform the same functions.

Enabling the remote trace controller

Enabling the remote trace controller involves enabling Java Management Extensions (JMX) in the IBM Data Server Driver for JDBC and SQLJ, and making the JMX agent available to clients.

The remote trace controller requires Java Standard Edition, Version 6 or later.

The steps for enabling the remote trace controller are:
1. Enable JMX to the IBM Data Server Driver for JDBC and SQLJ by setting the db2.jcc.jmxEnabled global configuration property to true or yes.
   For example, include this string in DB2JccConfiguration.properties:
   
   `db2.jcc.jmxEnabled=true`

2. Make the JMX agent (the platform MBean server) available to local or remote clients.
   - For local clients:
     Monitoring and management capabilities are automatically made available when the JVM is started. After your application is started, you can use a JMX client such as JConsole to connect locally to your Java process.
   - For remote clients, use one of the following methods:
     - Use the out-of-the-box JMX agent.
       Out-of-the-box management uses JMX built-in management utilities. To enable out-of-the-box management, you need to set a number of Java system properties. You must at least set the following property:
       `com.sun.management.jmxremote.port=portNum`
       In addition, you should ensure that authentication and SSL are properly configured.
       Full information on enabling out-of-the-box management is at the following URL:
Write a JMX agent. This technique is also discussed at:


In the following example, an RMI connector server is created for the PlatformMBeanServer using the MyCustomJMXAuthenticator object. The MyCustomJMXAuthenticator class defines how remote credentials are converted into a JAAS Subject by implementing the JMXAuthenticator interface:

```java
HashMap<String> env = new HashMap<String>();
env.put(JMXConnectorServer.AUTHENTICATOR, new MyCustomJMXAuthenticator());
env.put("jmx.remote.x.access.file", "my.access.file");

MBeanServer mbs =
    java.lang.management.ManagementFactory.getPlatformMBeanServer();
JMXServiceURL url =
    new JMXServiceURL("service:jmx:rmi:///jndi/rmi:///9999/jmxrmi");

JMXConnectorServer cs =
    JMXConnectorServerFactory.newJMXConnectorServer(url, env, mbs);
    cs.start();

public class MyCustomJMXAuthenticator implements JMXAuthenticator {
    public Subject authenticate(Object credentials) {
        // the hash contains username, password, etc...
        Hashtable<String> credentialsHash
            = (Hashtable<String>) credentials;

        ...
        // Authenticate using the provided credentials
        ...
        if (authentication-successful) {
            return new Subject(true,
                Collections.singleton
                    (new JMXPrincipal(credentialsHash.get("username"))),
                Collections.EMPTY_SET,
                Collections.EMPTY_SET);
        }
        throw new SecurityException("Invalid credentials");
    }
}
```

Accessing the remote trace controller

You can access the remote trace controller through out-of-the-box management tools, or through an application.

You use out-of-the-box management through a JMX-compliant management client, such as JConsole, which is part of Java Standard Edition, Version 6. Information on using JConsole for out-of-the-box management is at the following URL:


In an application that accesses the remote trace controller, the remote trace controller is a managed bean (MBean). JMX manages resources through JMX agents. A JMX agent is an MBean server. Each MBean represents a resource. Every MBean has a name, which you define through an object of class javax.management.ObjectName. You use the ObjectName object to register and retrieve MBeans in the MBeanServer.
The MBean name has two parts: the domain and the key properties. For the ObjectName for the IBM Data Server Driver for JDBC and SQLJ remote trace controller, the domain is com.ibm.db2.jcc, and the key properties are name=DB2TraceManager.

An application that accesses the remote trace controller must include these steps:
1. Establish a Remote Method Invocation (RMI) connection to an MBean server.
2. Perform a lookup on the remote trace controller in the MBean server.
3. Invoke trace operations on the MBean.

You can operate on the MBean in the following ways:
- Using an MBean proxy
- Without a proxy, through an MBeanServerConnection.

Example: accessing the remote trace controller without proxies: This example demonstrates accessing MBeans directly from an MBeanServerConnection. This method is the most generic because it does not require matching interface definitions on the JMX client application.

```java
Hashtable<String> env = new Hashtable<String>();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");
try {
    System.out.println(""DH...
    System.out.println("------------------------------D"D...);            
    System.out.println("Establish an RMI connection to an MBeanServer"D...);
    System.out.println("------------------------------D...);            
    JMXServiceURL url = new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi");
    JMXConnector jmxc = JMXConnectorFactory.connect(url, env);
    MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();

    System.out.println("D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("Process...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    String objectNameString = "com.ibm.db2.jcc:name=DB2TraceManager"D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    ObjectName name = new ObjectName(objectNameString);
    System.out.println("ObjectName"D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");

    System.out.println("D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("Print all attributes of the MBean"D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("------------------------------D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println(""D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("TraceDirectory = "D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("TraceFile = "D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("TraceFileAppend = "D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("TraceLevel = "D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");

    System.out.println("D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("Invoke some operations on the MBean"D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    System.out.println("------------------------------D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
    mbsc.invoke(name, "suspendTrace", null, null);
    System.out.println("success");

    System.out.println("Invoking resumeTrace..."D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...D...");
}
```

Chapter 9. System monitoring for the IBM Data Server Driver for JDBC and SQLJ 213
mbsc.invoke(name, "resumeTrace", null, null);
System.out.println("success");
}
catch (Exception e) {
    System.out.println("failure");
e.printStackTrace();
}

Example: accessing the remote trace controller with proxies: This example demonstrates the creation of a proxy to an MBean. The proxy implements the com.ibm.db2.jcc.mx.DB2TraceManagerMXBean interface. The application makes calls directly on the proxy, and the underlying proxy implementation invokes the MBean operation on the remote MBean server.

```
Hashtable<String> env = new Hashtable<String>();
env.put(Context.INITIAL_CONTEXT_FACTORY, 
        "com.sun.jndi.fscontext.RefFSContextFactory");
try {
    System.out.println("---------");
    System.out.println("Establish an RMI connection to an MBeanServer");
    System.out.println("---------");
    JMXServiceURL url = 
        new JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:9999/jmxrmi");
    JMXConnector jmxc = JMXConnectorFactory.connect(url, env);
    MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
    System.out.println("---------");
    System.out.println("Processing MBean");
    System.out.println("---------");
    String objectNameString = "com.ibm.db2.jcc:name=DB2TraceManager";
    ObjectName name = new ObjectName(objectNameString);
    System.out.println("ObjectName=\"+objectNameString\"");
    System.out.println("---------");
    System.out.println("Print all attributes of the MBean");
    System.out.println("---------");
    com.ibm.db2.jcc.mx.DB2TraceManagerMXBean mbeanProxy = 
        JM.newMBeanProxy(mbsc, name, 
            com.ibm.db2.jcc.mx.DB2TraceManagerMXBean.class, true);
    System.out.println("TraceDirectory = \"+mbeanProxy.getTraceDirectory\"");
    System.out.println("TraceFile = \"+mbeanProxy.getTraceFile\"");
    System.out.println("TraceFileAppend = \"+mbeanProxy.getTraceFileAppend\"");
    System.out.println("TraceLevel = \"+mbeanProxy.getTraceLevel\"");
    System.out.println("---------");
    System.out.println("Invoke some operations on the MBean");
    System.out.println("---------");
    mbeanProxy.suspendTrace();
    System.out.println("success");
    System.out.println("---------");
    mbeanProxy.resumeTrace();
    System.out.println("success");
}
```
Chapter 10. Java client support for high availability on IBM data servers

Client applications that connect to DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix (IDS) can easily take advantage of the high availability features of those data servers.

Client applications can use the following high availability features:

- **Automatic client reroute**
  Automatic client reroute capability is available on all IBM data servers. Automatic client reroute uses information that is provided by the data servers to redirect client applications from a server that experiences an outage to an alternate server. Automatic client reroute enables applications to continue their work with minimal interruption. Redirection of work to an alternate server is called failover.
  
  For connections to DB2 for z/OS data servers, automatic client reroute is part of the workload balancing feature. In general, for DB2 for z/OS, automatic client reroute should not be enabled without workload balancing.

- **Client affinities**
  Client affinities is a failover solution that is controlled completely by the client. It is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you use client affinities to enforce a specific order for failover to alternate servers.
  Client affinities is not applicable to a DB2 for z/OS data sharing environment, because all members of a data sharing group can access data concurrently. Data sharing is the recommended solution for high availability for DB2 for z/OS.

- **Workload balancing**
  Workload balancing is available on DB2 for z/OS and IBM Informix. Workload balancing ensures that work is distributed efficiently among servers in an IDS high-availability cluster or a DB2 for z/OS data sharing group.

The following table provides links to server-side information about these features.

<table>
<thead>
<tr>
<th>Data server</th>
<th>Related topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 Database for Linux, UNIX, and Windows</td>
<td>Automatic client reroute road map</td>
</tr>
<tr>
<td>IDS</td>
<td>Manage Cluster Connections with the Connection Manager</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Communicating with data sharing groups</td>
</tr>
</tbody>
</table>

**Important:** For connections to DB2 for z/OS, this information discusses direct connections to DB2 for z/OS. For information about high availability for connections through DB2 Connect Server, see the DB2 Connect documentation.
Java client support for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers

DB2 Database for Linux, UNIX, and Windows servers provide high availability for client applications, through automatic client reroute. This support is available for applications that use Java clients (JDBC, SQLJ, or pureQuery), as well as non-Java clients (ODBC, CLI, .NET, OLE DB, PHP, Ruby, or embedded SQL).

For Java clients, you need to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to take advantage of DB2 Database for Linux, UNIX, and Windows high-availability support. You need IBM Data Server Driver for JDBC and SQLJ version 3.58 or 4.8, or later.

High availability support for connections to DB2 Database for Linux, UNIX, and Windows servers includes:

**Automatic client reroute**

This support enables a client to recover from a failure by attempting to reconnect to the database through an alternate server. Reconnection to another server is called failover. For Java clients, automatic client reroute support is always enabled.

Automatic client reroute capability is available for a database that is defined on a single server. The configuration for that database includes specification of an alternate server. Failover involves reconnection to the alternate server.

For Java, client applications, failover for automatic client reroute can be seamless or non-seamless. With non-seamless failover, when the client application reconnects to another server, an error is always returned to the application, to indicate that failover (connection to the alternate server) occurred. With seamless failover, the driver does not return an error if a connection failure and successful reconnection to an alternate server occur during execution of the first SQL statement in a transaction.

In a DB2 pureScale™ instance, automatic client reroute support can be used without workload balancing or with workload balancing.

**Client affinities**

Client affinities is an automatic client reroute solution that is controlled completely by the client. It is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you use client affinities to enforce a specific order for failover to alternate servers.

Configuration of DB2 Database for Linux, UNIX, and Windows automatic client reroute support for Java clients

For connections to DB2 Database for Linux, UNIX, and Windows data servers that are set up for automatic client reroute, automatic client reroute capability is enabled at a Java client by default. However, you can set properties to fine-tune automatic client reroute support.

Automatic client reroute support for Java client applications that connect to DB2 Database for Linux, UNIX, and Windows works for connections that are obtained using the javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, javax.sql.XADataSource, or java.sql.DriverManager interface.
To configure automatic client reroute on a IBM Data Server Driver for JDBC and SQLJ client:

1. Set the appropriate properties to specify the primary and alternate server addresses to use if the first connection fails.
   - If your application is using the DriverManager interface for connections:
     a. Specify the server name and port number of the primary server that you want to use in the connection URL.
     b. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server name and port number of the alternate server that you want to use.

Restriction: Automatic client reroute support for connections that are made with the DriverManager interface has the following restrictions:
   - Alternate server information is shared between DriverManager connections only if you create the connections with the same URL and properties.
   - You cannot set the clientRerouteServerListJNDIName property or the clientRerouteServerListJNDIContext properties for a DriverManager connection.
   - Automatic client reroute is not enabled for default connections (jdbc:default:connection).

   - If your application is using the DataSource interface for connections, use one or both of the following techniques:
     - Set the server names and port numbers in DataSource properties:
       a. Set the serverName and portNumber properties to the server name and port number of the primary server that you want to use.
       b. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server name and port number of the alternate server that you want to use.
     - Configure JNDI for automatic client reroute by using a DB2ClientRerouteServerList instance to identify the primary server and alternate server.
       a. Create an instance of DB2ClientRerouteServerList.
       DB2ClientRerouteServerList is a serializable Java bean with the following properties:

       | Property name                                           | Data type          |
       |--------------------------------------------------------|--------------------|
       | com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName | String[]           |
       | com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber | int[]              |
       | com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName  | String[]           |
       | com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber  | int[]              |

       getXXX and setXXX methods are defined for each property.
       b. Set the
       com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber properties to the server name and port number of the primary server that you want to use.
       c. Set the
       com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName and
com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber properties to the server names and port numbers of the alternate server that you want to use.

d. To make the DB2ClientRerouteServerList persistent:
   1) Bind the DB2ClientRerouteServerList instance to the JNDI registry.
   2) Assign the JNDI name of the DB2ClientRerouteServerList object to the IBM Data Server Driver for JDBC and SQLJ clientRerouteServerListJNDIName property.
   3) Assign the name of the JNDI context that is used for binding and lookup of the DB2ClientRerouteServerList instance to the clientRerouteServerListJNDIContext property.

When a DataSource is configured to use JNDI for storing automatic client reroute alternate information, the standard server and port properties of the DataSource are not used for a getConnection request. Instead, the primary server address is obtained from the transient clientRerouteServerList information. If the JNDI store is not available due to a JNDI bind or lookup failure, the IBM Data Server Driver for JDBC and SQLJ attempts to make a connection using the standard server and port properties of the DataSource. Warnings are accumulated to indicate that a JNDI bind or lookup failure occurred.

After a failover:
- The IBM Data Server Driver for JDBC and SQLJ attempts to propagate the updated server information to the JNDI store.
- primaryServerName and primaryPortNumber values that are specified in DB2ClientRerouteServerList are used for the connection. If primaryServerName is not specified, the serverName and portNumber values for the DataSource instance are used.

If you configure DataSource properties as well as configuring JNDI for automatic client reroute, the DataSource properties have precedence over the JNDI configuration.

2. Set properties to control the number of retries, time between retries, and the frequency with which the server list is refreshed.

The following properties control retry behavior for automatic client reroute.

**maxRetriesForClientReroute**

The maximum number of connection retries for automatic client reroute.

When client affinities support is not configured, if maxRetriesForClientReroute or retryIntervalForClientReroute is not set, the default behavior is that the connection is retried for 10 minutes, with a wait time between retries that increases as the length of time from the first retry increases.

When client affinities is configured, the default for maxRetriesForClientReroute is 3.

**retryIntervalForClientReroute**

The number of seconds between consecutive connection retries.

When client affinities support is not configured, if retryIntervalForClientReroute or maxRetriesForClientReroute is not set, the default behavior is that the connection is retried for 10 minutes, with a wait time between retries that increases as the length of time from the first retry increases.
When client affinities is configured, the default for retryIntervalForClientReroute is 0 (no wait).

**Example of enabling DB2 Database for Linux, UNIX, and Windows automatic client reroute support in Java applications**

Java client setup for DB2 Database for Linux, UNIX, and Windows automatic client reroute support includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following example demonstrates setting up Java client applications for DB2 Database for Linux, UNIX, and Windows automatic client reroute support.

Suppose that your installation has a primary server and an alternate server with the following server names and port numbers:

<table>
<thead>
<tr>
<th>Server name</th>
<th>Port number</th>
</tr>
</thead>
<tbody>
<tr>
<td>srv1.sj.ibm.com</td>
<td>50000</td>
</tr>
<tr>
<td>srv3.sj.ibm.com</td>
<td>50002</td>
</tr>
</tbody>
</table>

The following code sets up DataSource properties in an application so that the application connects to srv1.sj.ibm.com as the primary server, and srv3.sj.ibm.com as the alternative server. That is, if srv1.sj.ibm.com is down during the initial connection, the driver should connect to srv3.sj.ibm.com.

```java
ds.setDriverType(4);
ds.setServerName("srv1.sj.ibm.com");
ds.setPortNumber("50000");
ds.setClientRerouteAlternateServerName("srv3.sj.ibm.com");
ds.setClientRerouteAlternatePortNumber("50002");
```

The following code configures JNDI for automatic client reroute. It creates an instance of DB2ClientRerouteServerList, binds that instance to the JNDI registry, and assigns the JNDI name of the DB2ClientRerouteServerList object to the clientRerouteServerListJNDIName property.

```java
// Create a starting context for naming operations
InitialContext registry = new InitialContext();

// Create a DB2ClientRerouteServerList object
DB2ClientRerouteServerList address = new DB2ClientRerouteServerList();

// Set the port number and server name for the primary server
address.setPrimaryPortNumber(50000);
address.setPrimaryServerName("srv1.sj.ibm.com");

// Set the port number and server name for the alternate server
int[] port = {50002};
String[] server = {"srv3.sj.ibm.com"};
address.setAlternatePortNumber(port);
address.setAlternateServerName(server);

registry.rebind("serverList", address);

// Assign the JNDI name of the DB2ClientRerouteServerList object to the clientRerouteServerListJNDIName property
```

```java
datasource.setClientRerouteServerListJNDIName("serverList");
```
Operation of automatic client reroute for connections to DB2 Database for Linux, UNIX, and Windows from Java clients

When IBM Data Server Driver for JDBC and SQLJ client reroute support is enabled, a Java application that is connected to a DB2 Database for Linux, UNIX, and Windows database can continue to run when the primary server has a failure.

Automatic client reroute for a Java application that is connected to a DB2 Database for Linux, UNIX, and Windows database operates in the following way when support for client affinities is disabled:

1. During each connection to the data source, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server information.
   - For the first connection to a DB2 Database for Linux, UNIX, and Windows database:
     a. If the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are set, the IBM Data Server Driver for JDBC and SQLJ loads those values into memory as the alternate server values, along with the primary server values serverName and portNumber.
     b. If the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties are not set, and a JNDI store is configured by setting the property clientRerouteServerListJNDIName on the DB2BaseDataSource, the IBM Data Server Driver for JDBC and SQLJ loads the primary and alternate server information from the JNDI store into memory.
     c. If no DataSource properties are set for the alternate servers, and JNDI is not configured, the IBM Data Server Driver for JDBC and SQLJ checks DNS tables for primary and alternate server information. If DNS information exists, the IBM Data Server Driver for JDBC and SQLJ loads those values into memory.
     d. If no primary or alternate server information is available, a connection cannot be established, and the IBM Data Server Driver for JDBC and SQLJ throws an exception.
   - For subsequent connections, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server values from driver memory.

2. The IBM Data Server Driver for JDBC and SQLJ attempts to connect to the data source using the primary server name and port number.
   If the connection is through the DriverManager interface, the IBM Data Server Driver for JDBC and SQLJ creates an internal DataSource object for automatic client reroute processing.

3. If the connection to the primary server fails:
   a. If this is the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to reconnect to a server using information that is provided by driver properties such as clientRerouteAlternateServerName and clientRerouteAlternatePortNumber.
   b. If this is not the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to make a connection using the information from the latest server list that is returned from the server.

   Connection to an alternate server is called failover.
   The IBM Data Server Driver for JDBC and SQLJ uses the maxRetriesForClientReroute and retryIntervalForClientReroute properties to
determine how many times to retry the connection and how long to wait between retries. An attempt to connect to the primary server and alternate servers counts as one retry.

4. If the connection is not established, maxRetriesForClientReroute and retryIntervalForClientReroute are not set, and the original serverName and portNumber values that are defined on the DataSource are different from the serverName and portNumber values that were used for the current connection, the connection is retried with the serverName and portNumber values that are defined on the DataSource.

5. If failover is successful during the initial connection, the driver generates an SQLWarning. If a successful failover occurs after the initial connection:
   • If seamless failover is enabled, and the following conditions are satisfied, the driver retries the transaction on the new server, without notifying the application.
     – The enableSeamlessFailover property is set to DB2BaseDataSource.YES (1).
     – The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
     – There are no global temporary tables in use on the server.
     – There are no open, held cursors.
   • If seamless failover is not in effect, the driver throws an SQLException to the application with error code -4498, to indicate to the application that the connection was automatically reestablished and the transaction was implicitly rolled back. The application can then retry its transaction without doing an explicit rollback first.

A reason code that is returned with error code -4498 indicates whether any database server special registers that were modified during the original connection are reestablished in the failover connection.

You can determine whether alternate server information was used in establishing the initial connection by calling the DB2Connection.alternateWasUsedOnConnect method.

6. After failover, driver memory is updated with new primary and alternate server information that is returned from the new primary server.

Examples

*Example: Automatic client reroute to a DB2 Database for Linux, UNIX, and Windows server when maxRetriesForClientReroute and retryIntervalForClientReroute are not set:*

Suppose that the following properties are set for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.NO (2)</td>
</tr>
<tr>
<td>serverName</td>
<td>host1</td>
</tr>
<tr>
<td>portNumber</td>
<td>port1</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host2</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port2</td>
</tr>
</tbody>
</table>

The following steps demonstrate an automatic client reroute scenario for a connection to a DB2 Database for Linux, UNIX, and Windows server:

1. The IBM Data Server Driver for JDBC and SQLJ loads host1:port1 into its memory as the primary server address, and host2:port2 into its memory as the alternate server address.
2. On the initial connection, the driver tries to connect to host1:port1.
3. The connection to host1:port1 fails, so the driver tries another connection to host1:port1.
4. The reconnection to host1:port1 fails, so the driver tries to connect to host2:port2.
5. The connection to host2:port2 succeeds.
6. The driver retrieves alternate server information that was received from server host2:port2, and updates its memory with that information.
   Assume that the driver receives a server list that contains host2:port2, host2a:port2a. host2:port2 is stored as the new primary server, and host2a:port2a is stored as the new alternate server. If another communication failure is detected on this same connection, or on another connection that is created from the same DataSource, the driver tries to connect to host2:port2 as the new primary server. If that connection fails, the driver tries to connect to the new alternate server host2a:port2a.
7. A communication failure occurs during the connection to host2:port2.
8. The driver tries to connect to host2a:port2a.
9. The connection to host2a:port2a is successful.
10. The driver retrieves alternate server information that was received from server host2a:port2a, and updates its memory with that information.

Example: Automatic client reroute to a DB2 Database for Linux, UNIX, and Windows server when maxRetriesForClientReroute and retryIntervalForClientReroute are set for multiple retries: Suppose that the following properties are set for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.NO (2)</td>
</tr>
<tr>
<td>serverName</td>
<td>host1</td>
</tr>
<tr>
<td>portNumber</td>
<td>port1</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host2</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port2</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
</tbody>
</table>

The following steps demonstrate an automatic client reroute scenario for a connection to a DB2 Database for Linux, UNIX, and Windows server:

1. The IBM Data Server Driver for JDBC and SQLJ loads host1:port1 into its memory as the primary server address, and host2:port2 into its memory as the alternate server address.
2. On the initial connection, the driver tries to connect to host1:port1.
3. The connection to host1:port1 fails, so the driver tries another connection to host1:port1.
4. The connection to host1:port1 fails again, so the driver tries to connect to host2:port2.
5. The connection to host2:port2 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1 and fails.
8. The driver tries to connect to host2:port2 and fails.
9. The driver waits two seconds.
10. The driver tries to connect to host1:port1 and fails.
11. The driver tries to connect to host2:port2 and fails.
12. The driver waits two seconds.
13. The driver throws an SQLException with error code -4499.

**Application programming requirements for high availability for connections to DB2 Database for Linux, UNIX, and Windows servers**

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to DB2 Database for Linux, UNIX, and Windows is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is non-seamless, and a connection is reestablished with the server, SQLCODE -4498 (for Java clients) or SQL30108N (for non-Java clients) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:

- Check the reason code that is returned with the error. Determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
- Execute all SQL operations that occurred during the previous transaction.

The following conditions must be satisfied for failover for connections to DB2 Database for Linux, UNIX, and Windows to be seamless:

- The application programming language is Java, CLI, or .NET.
- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- All global session data is closed or dropped.
- There are no open, held cursors.
- If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
- The application is not a stored procedure.
- Autocommit is not enabled. Seamless failover can occur when autocommit is enabled. However, the following situation can cause problems: Suppose that SQL work is successfully executed and committed at the data server, but the connection or server goes down before acknowledgment of the commit operation is sent back to the client. When the client re-establishes the connection, it replays the previously committed SQL statement. The result is that the SQL statement is executed twice. To avoid this situation, turn autocommit off when you enable seamless failover.

**Client affinities for DB2 Database for Linux, UNIX, and Windows**

Client affinities is a client-only method for providing automatic client reroute capability.
Client affinities is available for applications that use CLI, .NET, or Java (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity). All rerouting is controlled by the driver.

Client affinities is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you need to enforce a specific order for failover to alternate servers. You should use client affinities for automatic client reroute only if automatic client reroute that uses server failover capabilities does not work in your environment.

As part of configuration of client affinities, you specify a list of alternate servers, and the order in which connections to the alternate servers are tried. When client affinities is in use, connections are established based on the list of alternate servers instead of the host name and port number that are specified by the application. For example, if an application specifies that a connection is made to server1, but the configuration process specifies that servers should be tried in the order (server2, server3, server1), the initial connection is made to server2 instead of server1.

Failover with client affinities is seamless, if the following conditions are true:

- The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
- There are no global temporary tables in use on the server.
- There are no open, held cursors.

When you use client affinities, you can specify that if the primary server returns to operation after an outage, connections return from an alternate server to the primary server on a transaction boundary. This activity is known as failback.

**Configuration of client affinities for Java clients for DB2 Database for Linux, UNIX, and Windows connections**

To enable support for client affinities in Java applications, you set properties to indicate that you want to use client affinities, and to specify the primary and alternate servers.

The following table describes the property settings for enabling client affinities for Java applications.

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>A comma-separated list of the primary server and alternate servers</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>A comma-separated list of the port numbers for the primary server and alternate servers</td>
</tr>
<tr>
<td>enableSeamlessFailover</td>
<td>DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>The number of times to retry the connection to each server, including the primary server, after a connection to the primary server fails. The default is 3.</td>
</tr>
</tbody>
</table>
Table 35. Property settings to enable client affinities for Java applications (continued)

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>retryIntervalForClientReroute</td>
<td>The number of seconds to wait between retries. The default is no wait.</td>
</tr>
<tr>
<td>affinityFailbackInterval</td>
<td>The number of seconds to wait after the first transaction boundary to fail back to the primary server. Set this value if you want to fail back to the primary server.</td>
</tr>
</tbody>
</table>

Example of enabling client affinities in Java clients for DB2 Database for Linux, UNIX, and Windows connections

Before you can use client affinities for automatic client reroute in Java applications, you need to set properties to indicate that you want to use client affinities, and to identify the primary alternate servers.

The following example shows how to enable client affinities for failover without failback.

Suppose that you set the following properties for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServername</td>
<td>host1,host2,host3</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1,port2,port3</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
</tbody>
</table>

Suppose that a communication failure occurs during a connection to the server that is identified by host1:port1. The following steps demonstrate automatic client reroute with client affinities.

1. The driver tries to connect to host1:port1.
2. The connection to host1:port1 fails.
3. The driver waits two seconds.
4. The driver tries to connect to host1:port1.
5. The connection to host1:port1 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1.
8. The connection to host1:port1 fails.
9. The driver waits two seconds.
10. The driver tries to connect to host2:port2.
11. The connection to host2:port2 fails.
12. The driver waits two seconds.
13. The driver tries to connect to host2:port2.
14. The connection to host2:port2 fails.
15. The driver waits two seconds.
16. The driver tries to connect to host2:port2.
17. The connection to host2:port2 fails.
18. The driver waits two seconds.
19. The driver tries to connect to host3:port3.
20. The connection to host3:port3 fails.
21. The driver waits two seconds.
22. The driver tries to connect to host3:port3.
23. The connection to host3:port3 fails.
24. The driver waits two seconds.
25. The driver tries to connect to host3:port3.
26. The connection to host3:port3 fails.
27. The driver waits two seconds.
28. The driver throws an SQLException with error code -4499.

The following example shows how to enable client affinities for failover with failback.

Suppose that you set the following properties for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServername</td>
<td>host1,host2,host3</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1,port2,port3</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
<tr>
<td>affinityFailbackInterval</td>
<td>300</td>
</tr>
</tbody>
</table>

Suppose that the database administrator takes the server that is identified by host1:port1 down for maintenance after a connection is made to host1:port1. The following steps demonstrate failover to an alternate server and failback to the primary server after maintenance is complete.

1. The driver successfully connects to host1:port1 on behalf of an application.
2. The database administrator brings down host1:port1.
3. The application tries to do work on the connection.
4. The driver successfully fails over to host2:port2.
5. After a total of 200 seconds have elapsed, the work is committed.
6. After a total of 300 seconds have elapsed, the failback interval has elapsed. The driver checks whether the primary server is up. It is not up, so no failback occurs.
7. After a total of 350 seconds have elapsed, host1:port1 is brought back online.
8. The application continues to do work on host2:port2, because the latest failback interval has not elapsed.
9. After a total of 600 seconds have elapsed, the failback interval has elapsed again. The driver checks whether the primary server is up. It is now up.
10. After a total of 650 seconds have elapsed, the work is committed.
11. After a total of 651 seconds have elapsed, the application tries to start a new transaction on host2:port2. Failback to host1:port1 occurs, so the new transaction starts on host1:port1.
Java client support for high availability for connections to IDS servers

High-availability cluster support on IBM Informix (IDS) servers provides high availability for client applications, through workload balancing and automatic client reroute. This support is available for applications that use Java clients (JDBC, SQLJ, or pureQuery), or non-Java clients (ODBC, CLI, .NET, OLE DB, PHP, Ruby, or embedded SQL).

For Java clients, you need to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to take advantage of IDS high-availability cluster support.

For non-Java clients, you need to use one of the following clients or client packages to take advantage of high-availability cluster support:

- IBM Data Server Client
- IBM Data Server Runtime Client
- IBM Data Server Driver Package
- IBM Data Server Driver for ODBC and CLI

Cluster support for high availability for connections to IDS servers includes:

**Automatic client reroute**

This support enables a client to recover from a failure by attempting to reconnect to the database through any available server in a high-availability cluster. Reconnection to another server is called failover. You enable automatic client reroute on the client by enabling workload balancing on the client.

In an IDS environment, primary and standby servers correspond to members of a high-availability cluster that is controlled by a Connection Manager. If multiple Connection Managers exist, the client can use them to determine primary and alternate server information. The client uses alternate Connection Managers only for the initial connection.

Failover for automatic client reroute can be seamless or non-seamless. With non-seamless failover, when the client application reconnects to an alternate server, the server always returns an error to the application, to indicate that failover (connection to the alternate server) occurred.

For Java, CLI, or .NET client applications, failover for automatic client reroute can be seamless or non-seamless. Seamless failover means that when the application successfully reconnects to an alternate server, the server does not return an error to the application.

**Workload balancing**

Workload balancing can improve availability of an IDS high-availability cluster. When workload balancing is enabled, the client gets frequent status information about the members of a high-availability cluster. The client uses this information to determine the server to which the next transaction should be routed. With workload balancing, IDS Connection Managers ensure that work is distributed efficiently among servers and that work is transferred to another server if a server has a failure.

**Connection concentrator**

This support is available for Java applications that connect to IDS. The connection concentrator reduces the resources that are required on IDS database servers to support large numbers of workstation and web users. With the connection concentrator, only a few concurrent, active physical connections are needed to support many applications that concurrently access the database.
server. When you enable workload balancing on a Java client, you automatically enable the connection concentrator.

**Client affinities**

Client affinities is an automatic client reroute solution that is controlled completely by the client. It is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you use client affinities to enforce a specific order for failover to alternate servers.

**Configuration of IDS high-availability support for Java clients**

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects to an IDS high-availability cluster, you need to connect to an address that represents a Connection Manager, and set the properties that enable workload balancing and the maximum number of connections.

High availability support for Java clients that connect to IBM Informix works for connections that are obtained using the javax.sql.DataSource, javax.sql.ConnectionPoolDataDataSource, javax.sql.XADataSource, or java.sql.DriverManager interface.

**Restriction:** High availability support for connections that are made with the DriverManager interface has the following restrictions:

- Alternate server information is shared between DriverManager connections only if you create the connections with the same URL and properties.
- You cannot set the clientRerouteServerListJNDIName property or the clientRerouteServerListJNDIContext properties for a DriverManager connection.
- High availability support is not enabled for default connections (jdbc:default:connection).

Before you can enable IBM Data Server Driver for JDBC and SQLJ for high availability for connections to IBM Informix, your installation must have one or more Connection Managers, a primary server, and one or more alternate servers.

The following table describes the basic property settings for enabling workload balancing for Java applications.

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB property</td>
<td>true</td>
</tr>
<tr>
<td>maxTransportObjects property</td>
<td>The maximum number of connections that the requester can make to the high-availability cluster</td>
</tr>
</tbody>
</table>

**Connection address:**

| server | The IP address of a Connection Manager. See "Setting server and port properties for connecting to a Connection Manager” on page 229. |
| port   | The SQL port number for the Connection Manager. See "Setting server and port properties for connecting to a Connection Manager” on page 229. |
| database | The database name |
If you want to enable the connection concentrator, but you do not want to enable workload balancing, you can use these properties.

**Table 37. Settings to enable the IDS connection concentrator without workload balancing in Java applications**

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB property</td>
<td>false</td>
</tr>
<tr>
<td>enableConnectionConcentrator property</td>
<td>true</td>
</tr>
</tbody>
</table>

If you want to fine-tune IDS high-availability support, additional properties are available. The properties for the IBM Data Server Driver for JDBC and SQLJ are listed in the following table. Those properties are configuration properties, and not Connection or DataSource properties.

**Table 38. Properties for fine-tuning IDS high-availability support for connections from the IBM Data Server Driver for JDBC and SQLJ**

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ configuration property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2.jcc.maxTransportObjectIdleTime</td>
<td>Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60. The minimum supported value is 0.</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectWaitTime</td>
<td>Specifies the number of seconds that the client will wait for a transport to become available. The default is -1 (unlimited). The minimum supported value is 0.</td>
</tr>
<tr>
<td>db2.jcc.minTransportObjects</td>
<td>Specifies the lower limit for the number of transport objects in a global transport object pool. The default value is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.</td>
</tr>
</tbody>
</table>

**Setting server and port properties for connecting to a Connection Manager**

To set the server and port number for connecting to a Connection Manager, follow this process:

- If your high-availability cluster is using a single Connection Manager, and your application is using the DataSource interface for connections, set the serverName and portNumber properties to the server name and port number of the Connection Manager.
- If your high-availability cluster is using a single Connection Manager, and your application is using the DriverManager interface for connections, specify the server name and port number of the Connection manager in the connection URL.
- If your high-availability cluster is using more than one Connection manager, and your application is using the DriverManager interface for connections:
  1. Specify the server name and port number of the main Connection Manager that you want to use in the connection URL.
  2. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.
If your high-availability cluster is using more than one Connection Manager, and your application is using the DataSource interface for connections, use one of the following techniques:

- Set the server names and port numbers in DataSource properties:
  1. Set the serverName and portNumber properties to the server name and port number of the main Connection Manager that you want to use.
  2. Set the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.

- Configure JNDI for high availability by using a DB2ClientRerouteServerList instance to identify the main Connection Manager and alternative Connection Managers.
  1. Create an instance of DB2ClientRerouteServerList.
     DB2ClientRerouteServerList is a serializable Java bean with the following properties:

     | Property name                                               | Data type |
     |-------------------------------------------------------------|-----------|
     | com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName | String[]  |
     | com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber | int[]     |
     | com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName  | String[]  |
     | com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber  | int[]     |

     getXXX and setXXX methods are defined for each property.
  2. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.primaryPortNumber properties to the server name and port number of the main Connection Manager that you want to use.
  3. Set the com.ibm.db2.jcc.DB2ClientRerouteServerList.alternateServerName and com.ibm.db2.jcc.DB2ClientRerouteServerList.alternatePortNumber properties to the server names and port numbers of the alternative Connection Managers that you want to use.
  4. To make the DB2ClientRerouteServerList persistent:
     a. Bind the DB2ClientRerouteServerList instance to the JNDI registry.
     b. Assign the JNDI name of the DB2ClientRerouteServerList object to the IBM Data Server Driver for JDBC and SQLJ clientRerouteServerListJNDIName property.
     c. Assign the name of the JNDI context that is used for binding and lookup of the DB2ClientRerouteServerList instance to the clientRerouteServerListJNDIContext property.

When a DataSource is configured to use JNDI for storing automatic client reroute alternate information, the standard server and port properties of the DataSource are not used for a getConnection request. Instead, the primary server address is obtained from the transient clientRerouteServerList information. If the JNDI store is not available due to a JNDI bind or lookup failure, the IBM Data Server Driver for JDBC and SQLJ attempts to make a connection using the standard server and port properties of the DataSource. Warnings are accumulated to indicate that a JNDI bind or lookup failure occurred.

After a failover:
- The IBM Data Server Driver for JDBC and SQLJ attempts to propagate the updated server information to the JNDI store.
- primaryServerName and primaryPortNumber values that are specified in DB2ClientRerouteServerList are used for the connection. If primaryServerName is not specified, the serverName value for the DataSource instance is used.

Example of enabling IDS high availability support in Java applications

Java client setup for IDS high availability support includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following example demonstrates setting up Java client applications for IDS high availability support.

Before you can set up the client, you need to configure one or more high availability clusters that are controlled by Connection Managers.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support workload balancing by following these steps:
   a. Issue the following command in a command line window:
      
      ```
      java com.ibm.db2.jcc.DB2Jcc -version
      ```
   b. Find a line in the output like this, and check that `nnn` is 3.52 or later.
   c. `[jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture nnn xxx`

2. Set IBM Data Server Driver for JDBC and SQLJ Connection or DataSource properties to enable workload balancing:
   - `enableSysplexWLB`
   - `maxTransportObjects`
   - `maxRefreshInterval`
   
   Start with settings similar to these:

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB</td>
<td>true</td>
</tr>
<tr>
<td>maxTransportObjects</td>
<td>80</td>
</tr>
<tr>
<td>maxRefreshInterval</td>
<td>30</td>
</tr>
</tbody>
</table>

   Enabling workload balancing enables the connection concentrator by default.
   The values that are specified are not intended to be recommended values. You need to determine values based on factors such as the number of transport objects that are available. The number of transport objects must be equal to or greater than the number of connection objects.

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune the workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a DB2JccConfiguration.properties file by following these steps:
   a. Create a DB2JccConfiguration.properties file or edit the existing DB2JccConfiguration.properties file.
   b. Set the following configuration property:
• db2.jcc.maxTransportObjects
  Start with a setting similar to this one:
  db2.jcc.maxTransportObjects=500
  
c. Include the directory that contains DB2JccConfiguration.properties in the
  CLASSPATH concatenation.

**Operation of automatic client reroute for connections to IDS from Java clients**

When IBM Data Server Driver for JDBC and SQLJ client reroute support is
enabled, a Java application that is connected to an IBM Informix (IDS)
high-availability cluster can continue to run when the primary server has a failure.

Automatic client reroute for a Java application that is connected to an IDS server
operates in the following way when automatic client reroute is enabled:

1. During each connection to the data source, the IBM Data Server Driver for
   JDBC and SQLJ obtains primary and alternate server information.
   - For the first connection to IDS:
     a. The application specifies a server and port for the initial connection.
        Those values identify a Connection Manager.
     b. The IBM Data Server Driver for JDBC and SQLJ uses the information
        from the Connection Manager to obtain information about the primary
        and alternate servers. IBM Data Server Driver for JDBC and SQLJ loads
        those values into memory.
     c. If the initial connection to the Connection Manager fails:
        – If the clientRerouteAlternateServerName and
          clientRerouteAlternatePortNumber properties are set, the IBM Data
          Server Driver for JDBC and SQLJ connects to the Connection Manager
          that is identified by clientRerouteAlternateServerName and
          clientRerouteAlternatePortNumber, and obtains information about
          primary and alternate servers from that Connection Manager. The IBM
          Data Server Driver for JDBC and SQLJ loads those values into memory
          as the primary and alternate server values.
        – If the clientRerouteAlternateServerName and
          clientRerouteAlternatePortNumber properties are not set, and a JNDI
          store is configured by setting the property
          clientRerouteServerListJNDIName on the DB2BaseDataSource, the IBM
          Data Server Driver for JDBC and SQLJ connects to the Connection
          Manager that is identified by
          DB2ClientRerouteServerList.alternateServerName and
          DB2ClientRerouteServerList.alternatePortNumber, and obtains
          information about primary and alternate servers from that Connection
          Manager. IBM Data Server Driver for JDBC and SQLJ loads the
          primary and alternate server information from the Connection
          Manager into memory.
     d. If clientRerouteAlternateServerName and
        clientRerouteAlternatePortNumber are not set, and JNDI is not
        configured, the IBM Data Server Driver for JDBC and SQLJ checks DNS
        tables for Connection Manager server and port information. If DNS
        information exists, the IBM Data Server Driver for JDBC and SQLJ
        connects to the Connection Manager, obtains information about primary
        and alternate servers, and loads those values into memory.
e. If no primary or alternate server information is available, a connection cannot be established, and the IBM Data Server Driver for JDBC and SQLJ throws an exception.

- For subsequent connections, the IBM Data Server Driver for JDBC and SQLJ obtains primary and alternate server values from driver memory.

2. The IBM Data Server Driver for JDBC and SQLJ attempts to connect to the data source using the primary server name and port number.
   If the connection is through the DriverManager interface, the IBM Data Server Driver for JDBC and SQLJ creates an internal DataSource object for automatic client reroute processing.

3. If the connection to the primary server fails:
   a. If this is the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to reconnect to the original primary server.
   b. If this is not the first connection, the IBM Data Server Driver for JDBC and SQLJ attempts to reconnect to the new primary server, whose server name and port number were provided by the server.
   c. If reconnection to the primary server fails, the IBM Data Server Driver for JDBC and SQLJ attempts to connect to the alternate servers.
      If this is not the first connection, the latest alternate server list is used to find the next alternate server.

Connection to an alternate server is called failover.

The IBM Data Server Driver for JDBC and SQLJ uses the maxRetriesForClientReroute and retryIntervalForClientReroute properties to determine how many times to retry the connection and how long to wait between retries. An attempt to connect to the primary server and alternate servers counts as one retry.

4. If the connection is not established, maxRetriesForClientReroute and retryIntervalForClientReroute are not set, and the original serverName and portNumber values that are defined on the DataSource are different from the serverName and portNumber values that were used for the original connection, retry the connection with the serverName and portNumber values that are defined on the DataSource.

5. If failover is successful during the initial connection, the driver generates an SQLWarning. If a successful failover occurs after the initial connection:
   - If seamless failover is enabled, the driver retries the transaction on the new server, without notifying the application.
     The following conditions must be satisfied for seamless failover to occur:
     - The enableSeamlessFailover property is set to DB2BaseDataSource.YES (1).
       If Sysplex workload balancing is in effect (the value of the enableSysplexWLB is true), seamless failover is attempted, regardless of the enableSeamlessFailover setting.
     - The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
     - There are no global temporary tables in use on the server.
     - There are no open, held cursors.
   - If seamless failover is not in effect, the driver throws an SQLException to the application with error code -4498, to indicate to the application that the connection was automatically reestablished and the transaction was implicitly rolled back. The application can then retry its transaction without doing an explicit rollback first.
A reason code that is returned with error code -4498 indicates whether any
database server special registers that were modified during the original
connection are reestablished in the failover connection.

You can determine whether alternate server information was used in
establishing the initial connection by calling the
DB2Connection.alternateWasUsedOnConnect method.

6. After failover, driver memory is updated with new primary and alternate
   server information from the new primary server.

Examples

Example: Automatic client reroute to an IDS server when maxRetriesForClientReroute and
retryIntervalForClientReroute are not set: Suppose that the following properties are set
for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.NO (2)</td>
</tr>
<tr>
<td>serverName</td>
<td>host1</td>
</tr>
<tr>
<td>portNumber</td>
<td>port1</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>host2</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port2</td>
</tr>
</tbody>
</table>

The following steps demonstrate an automatic client reroute scenario for a
connection to IDS:
1. The IBM Data Server Driver for JDBC and SQLJ tries to connect to the
   Connection Manager that is identified by host1:port1.
2. The connection to host1:port1 fails, so the driver tries to connect to the
   Connection Manager that is identified by host2:port2.
3. The connection to host2:port2 succeeds.
4. The driver retrieves alternate server information that was received from server
   host2:port2, and updates its memory with that information.
   Assume that the driver receives a server list that contains host2:port2,
   host2a:port2a. host2:port2 is stored as the new primary server, and
   host2a:port2a is stored as the new alternate server. If another communication
   failure is detected on this same connection, or on another connection that is
   created from the same DataSource, the driver tries to connect to host2:port2 as
   the new primary server. If that connection fails, the driver tries to connect to
   the new alternate server host2a:port2a.
5. The driver connects to host1a:port1a.
6. A failure occurs during the connection to host1a:port1a.
7. The driver tries to connect to host2a:port2a.
8. The connection to host2a:port2a is successful.
9. The driver retrieves alternate server information that was received from server
   host2a:port2a, and updates its memory with that information.

Example: Automatic client reroute to an IDS server when maxRetriesForClientReroute and
retryIntervalForClientReroute are set for multiple retries: Suppose that the following
properties are set for a connection to a database:
The following steps demonstrate an automatic client reroute scenario for a connection to IDS:

1. The IBM Data Server Driver for JDBC and SQLJ tries to connect to the Connection Manager that is identified by host1:port1.
2. The connection to host1:port1 fails, so the driver tries to connect to the Connection Manager that is identified by host2:port2.
3. The connection to host2:port2 succeeds.
4. The driver retrieves alternate server information from the connection manager that is identified by host2:port2, and updates its memory with that information. Assume that the Connection Manager identifies host1a:port1a as the new primary server, and host2a:port2a as the new alternate server.
5. The driver tries to connect to host1a:port1a.
6. The connection to host1a:port1a fails.
7. The driver tries to connect to host2a:port2a.
8. The connection to host2a:port2a fails.
9. The driver waits two seconds.
10. The driver tries to connect to host1a:port1a.
11. The connection to host1a:port1a fails.
12. The driver tries to connect to host2a:port2a.
13. The connection to host2a:port2a fails.
14. The driver waits two seconds.
15. The driver tries to connect to host1a:port1a.
16. The connection to host1a:port1a fails.
17. The driver tries to connect to host2a:port2a.
18. The connection to host2a:port2a fails.
19. The driver waits two seconds.
20. The driver throws an SQLException with error code -4499.

**Operation of workload balancing for connections to IDS from Java clients**

Workload balancing (also called transaction-level workload balancing) for connections to IBM Informix (IDS) contributes to high availability by balancing work among servers in a high-availability cluster at the start of a transaction.

The following overview describes the steps that occur when a client connects to an IDS Connection Manager, and workload balancing is enabled:
1. When the client first establishes a connection using the IP address of the Connection Manager, the Connection Manager returns the server list and the connection details (IP address, port, and weight) for the servers in the cluster. The server list is cached by the client. The default lifespan of the cached server list is 30 seconds.

2. At the start of a new transaction, the client reads the cached server list to identify a server that has untapped capacity, and looks in the transport pool for an idle transport that is tied to the under-utilized server. (An idle transport is a transport that has no associated connection object.)
   - If an idle transport is available, the client associates the connection object with the transport.
   - If, after a user-configurable timeout, no idle transport is available in the transport pool and no new transport can be allocated because the transport pool has reached its limit, an error is returned to the application.

3. When the transaction runs, it accesses the server that is tied to the transport.

4. When the transaction ends, the client verifies with the server that transport reuse is still allowed for the connection object.

5. If transport reuse is allowed, the server returns a list of SET statements for special registers that apply to the execution environment for the connection object.
   The client caches these statements, which it replays in order to reconstruct the execution environment when the connection object is associated with a new transport.

6. The connection object is then dissociated from the transport, if the client determines that it needs to do so.

7. The client copy of the server list is refreshed when a new connection is made, or every 30 seconds, or at the user-configured interval.

8. When workload balancing is required for a new transaction, the client uses the previously described process to associate the connection object with a transport.

**Application programming requirements for high availability for connections from Java clients to IDS servers**

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to IDS is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is non-seamless, and a connection is reestablished with the server, SQLCODE -4498 (for Java clients) or SQL30108N (for non-Java clients) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:
   - Check the reason code that is returned with the error. Determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
   - Execute all SQL operations that occurred during the previous transaction.

The following conditions must be satisfied for seamless failover to occur during connections to IDS databases:
   - The application programming language is Java, CLI, or .NET.
   - The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
• The data server must allow transport reuse at the end of the previous transaction.
• All global session data is closed or dropped.
• There are no open held cursors.
• If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
• The application is not a stored procedure.
• Autocommit is not enabled. Seamless failover can occur when autocommit is enabled. However, the following situation can cause problems: Suppose that SQL work is successfully executed and committed at the data server, but the connection or server goes down before acknowledgment of the commit operation is sent back to the client. When the client re-establishes the connection, it replays the previously committed SQL statement. The result is that the SQL statement is executed twice. To avoid this situation, turn autocommit off when you enable seamless failover.

In addition, seamless automatic client reroute might not be successful if the application has autocommit enabled. With autocommit enabled, a statement might be executed and committed multiple times.

Client affinities for connections to IDS from Java clients

Client affinities is a client-only method for providing automatic client reroute capability.

Client affinities is available for applications that use CLI, .NET, or Java (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity). All rerouting is controlled by the driver.

Client affinities is intended for situations in which you need to connect to a particular primary server. If an outage occurs during the connection to the primary server, you need to enforce a specific order for failover to alternate servers. You should use client affinities for automatic client reroute only if automatic client reroute that uses server failover capabilities does not work in your environment.

As part of configuration of client affinities, you specify a list of alternate servers, and the order in which connections to the alternate servers are tried. When client affinities is in use, connections are established based on the list of alternate servers instead of the host name and port number that are specified by the application. For example, if an application specifies that a connection is made to server1, but the configuration process specifies that servers should be tried in the order (server2, server3, server1), the initial connection is made to server2 instead of server1.

Failover with client affinities is seamless, if the following conditions are true:
• The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
• There are no global temporary tables in use on the server.
• There are no open, held cursors.

When you use client affinities, you can specify that if the primary server returns to operation after an outage, connections return from an alternate server to the primary server on a transaction boundary. This activity is known as failback.
Configuration of client affinities for Java clients for IDS connections

To enable support for client affinities in Java applications, you set properties to indicate that you want to use client affinities, and to specify the primary and alternate servers.

The following table describes the property settings for enabling client affinities for Java applications.

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServerName</td>
<td>A comma-separated list of the primary server and alternate servers</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>A comma-separated list of the port numbers for the primary server and alternate servers</td>
</tr>
<tr>
<td>enableSeamlessFailover</td>
<td>DB2BaseDataSource.YES (1) for seamless failover; DB2BaseDataSource.NO (2) or enableSeamlessFailover not specified for no seamless failover</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>The number of times to retry the connection to each server, including the primary server, after a connection to the primary server fails. The default is 3.</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>The number of seconds to wait between retries. The default is no wait.</td>
</tr>
<tr>
<td>affinityFailbackInterval</td>
<td>The number of seconds to wait after the first transaction boundary to fail back to the primary server. Set this value if you want to fail back to the primary server.</td>
</tr>
</tbody>
</table>

Example of enabling client affinities in Java clients for IDS connections

Before you can use client affinities for automatic client reroute in Java applications, you need to set properties to indicate that you want to use client affinities, and to identify the primary alternate servers.

The following example shows how to enable client affinities for failover without failback.

Suppose that you set the following properties for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServername</td>
<td>host1,host2,host3</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1,port2,port3</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
</tbody>
</table>
Suppose that a communication failure occurs during a connection to the server that is identified by host1:port1. The following steps demonstrate automatic client reroute with client affinities.

1. The driver tries to connect to host1:port1.
2. The connection to host1:port1 fails.
3. The driver waits two seconds.
4. The driver tries to connect to host1:port1.
5. The connection to host1:port1 fails.
6. The driver waits two seconds.
7. The driver tries to connect to host1:port1.
8. The connection to host1:port1 fails.
9. The driver waits two seconds.
10. The driver tries to connect to host2:port2.
11. The connection to host2:port2 fails.
12. The driver waits two seconds.
13. The driver tries to connect to host2:port2.
14. The connection to host2:port2 fails.
15. The driver waits two seconds.
16. The driver tries to connect to host2:port2.
17. The connection to host2:port2 fails.
18. The driver waits two seconds.
19. The driver tries to connect to host3:port3.
20. The connection to host3:port3 fails.
21. The driver waits two seconds.
22. The driver tries to connect to host3:port3.
23. The connection to host3:port3 fails.
24. The driver waits two seconds.
25. The driver tries to connect to host3:port3.
26. The connection to host3:port3 fails.
27. The driver waits two seconds.
28. The driver throws an SQLException with error code -4499.

The following example shows how to enable client affinities for failover with failback.

Suppose that you set the following properties for a connection to a database:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableClientAffinitiesList</td>
<td>DB2BaseDataSource.YES (1)</td>
</tr>
<tr>
<td>clientRerouteAlternateServername</td>
<td>host1, host2, host3</td>
</tr>
<tr>
<td>clientRerouteAlternatePortNumber</td>
<td>port1, port2, port3</td>
</tr>
<tr>
<td>maxRetriesForClientReroute</td>
<td>3</td>
</tr>
<tr>
<td>retryIntervalForClientReroute</td>
<td>2</td>
</tr>
<tr>
<td>affinityFailbackInterval</td>
<td>300</td>
</tr>
</tbody>
</table>

Chapter 10. Java client support for high availability on IBM data servers 239
Suppose that the database administrator takes the server that is identified by host1:port1 down for maintenance after a connection is made to host1:port1. The following steps demonstrate failover to an alternate server and failback to the primary server after maintenance is complete.

1. The driver successfully connects to host1:port1 on behalf of an application.
2. The database administrator brings down host1:port1.
3. The application tries to do work on the connection.
4. The driver successfully fails over to host2:port2.
5. After a total of 200 seconds have elapsed, the work is committed.
6. After a total of 300 seconds have elapsed, the failback interval has elapsed. The driver checks whether the primary server is up. It is not up, so no failback occurs.
7. After a total of 350 seconds have elapsed, host1:port1 is brought back online.
8. The application continues to do work on host2:port2, because the latest failback interval has not elapsed.
9. After a total of 600 seconds have elapsed, the failback interval has elapsed again. The driver checks whether the primary server is up. It is now up.
10. After a total of 650 seconds have elapsed, the work is committed.
11. After a total of 651 seconds have elapsed, the application tries to start a new transaction on host2:port2. Failback to host1:port1 occurs, so the new transaction starts on host1:port1.

Java client support for high availability for connections to DB2 for z/OS servers

Sysplex workload balancing functionality on DB2 for z/OS servers provides high availability for client applications that connect directly to a data sharing group. Sysplex workload balancing functionality provides workload balancing and automatic client reroute capability. This support is available for applications that use Java clients (JDBC, SQLJ, or pureQuery) that use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or non-Java clients (ODBC, CLI, .NET, OLE DB, PHP, Ruby, or embedded SQL). Workload balancing is transparent to applications.

A Sysplex is a set of z/OS systems that communicate and cooperate with each other through certain multisystem hardware components and software services to process customer workloads. DB2 for z/OS subsystems on the z/OS systems in a Sysplex can be configured to form a data sharing group. With data sharing, applications that run on more than one DB2 for z/OS subsystem can read from and write to the same set of data concurrently. One or more coupling facilities provide high-speed caching and lock processing for the data sharing group. The Sysplex, together with the Workload Manager (WLM), dynamic virtual IP address (DVIPA), and the Sysplex Distributor, allow a client to access a DB2 for z/OS database over TCP/IP with network resilience, and distribute transactions for an application in a balanced manner across members within the data sharing group.

Central to these capabilities is a server list that the data sharing group returns on connection boundaries and optionally on transaction boundaries. This list contains the IP address and WLM weight for each data sharing group member. With this information, a client can distribute transactions in a balanced manner, or identify the member to use when there is a communication failure.
The server list is returned on the first successful connection to the DB2 for z/OS data server. After the client has received the server list, the client directly accesses a data sharing group member based on information in the server list.

DB2 for z/OS provides several methods for clients to access a data sharing group. The access method that is set up for communication with the data sharing group determines whether Sysplex workload balancing is possible. The following table lists the access methods and indicates whether Sysplex workload balancing is possible.

Table 41. Data sharing access methods and Sysplex workload balancing

<table>
<thead>
<tr>
<th>Data sharing access method</th>
<th>Description</th>
<th>Sysplex workload balancing possible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group access</td>
<td>A requester uses the group’s dynamic virtual IP address (DVIPA) to make an initial connection to the DB2 for z/OS location. A connection to the data sharing group that uses the group IP address and SQL port is always successful if at least one member is started. The server list that is returned by the data sharing group contains: • A list of members that are currently active and can perform work • The WLM weight for each member The group IP address is configured using the z/OS Sysplex distributor. To clients that are outside the Sysplex, the Sysplex distributor provides a single IP address that represents a DB2 location. In addition to providing fault tolerance, the Sysplex distributor can be configured to provide connection load balancing.</td>
<td>Yes</td>
</tr>
<tr>
<td>Member-specific access</td>
<td>A requester uses a location alias to make an initial connection to one of the members that is represented by the alias. A connection to the data sharing group that uses the group IP address and alias SQL port is always successful if at least one member is started. The server list that is returned by the data sharing group contains: • A list of members that are currently active, can perform work, and have been configured as an alias • The WLM weight for each member The requester uses this information to connect to the member or members with the most capacity that are also associated with the location alias. Member-specific access is used when requesters need to take advantage of Sysplex workload balancing among a subset of members of a data sharing group.</td>
<td>Yes</td>
</tr>
<tr>
<td>Single-member access</td>
<td>Single-member access is used when requesters need to access only one member of a data sharing group. For single-member access, the connection uses the member-specific IP address.</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 41. Data sharing access methods and Sysplex workload balancing (continued)

<table>
<thead>
<tr>
<th>Data sharing access method</th>
<th>Description</th>
<th>Sysplex workload balancing possible?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:


Sysplex workload balancing includes automatic client reroute: Automatic client reroute support enables a client to recover from a failure by attempting to reconnect to the database through any available member of a Sysplex. Reconnection to another member is called failover.

Sysplex workload balancing during migration of a data sharing group to DB2 9.1 for z/OS: When you migrate a data sharing group to DB2 9.1 for z/OS new-function mode, you need to take these steps:

1. Restart all members of the data group.
2. Restart the JVMs under which applications that connect to the data sharing group using IBM Data Server Driver for JDBC and SQLJ type 4 connectivity run.

Stopping and starting all members prevents applications that use Sysplex workload balancing from having unbalanced connections.

For Java, CLI, or .NET client applications, failover for automatic client reroute can be seamless or non-seamless. Seamless failover means that when the application successfully reconnects to an alternate server, the server does not return an error to the application.

Client support for high availability is incompatible with a DB2 Connect environment: Client support for high availability requires a DB2 Connect license. However, this support is incompatible with a DB2 Connect environment. Do not use a DB2 Connect server when you use client support for high availability.

Do not use client affinities: Client affinities should not be used as a high availability solution for direct connections to DB2 for z/OS. Client affinities is not applicable to a DB2 for z/OS data sharing environment, because all members of a data sharing group can access data concurrently. A major disadvantage of client affinities in a data sharing environment is that if failover occurs because a data sharing group member fails, the member that fails might have retained locks that can severely affect transactions on the member to which failover occurs.

Configuration of Sysplex workload balancing at a Java client

To configure a IBM Data Server Driver for JDBC and SQLJ client application that connects directly to DB2 for z/OS to use Sysplex workload balancing, you need to use IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. You also need to connect to an address that represents the data sharing group (for group access) or a subset of the data sharing group (for member-specific access), and set the properties that enable workload balancing and the maximum number of connections.

The following table describes the basic property settings for Java applications.
Table 42. Basic settings to enable Sysplex high availability support in Java applications

<table>
<thead>
<tr>
<th>Data sharing access method</th>
<th>IBM Data Server Driver for JDBC and SQLJ setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group access</td>
<td>enableSysplexWLB property</td>
<td>true</td>
</tr>
<tr>
<td>Connection address:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>The group IP address or domain name of the data sharing group</td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>The SQL port number for the DB2 location</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>The DB2 location name that is defined during installation</td>
<td></td>
</tr>
<tr>
<td>Member-specific access</td>
<td>enableSysplexWLB property</td>
<td>true</td>
</tr>
<tr>
<td>Connection address:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>The group IP address or domain name of the data sharing group</td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>The port number for the DB2 location alias</td>
<td></td>
</tr>
<tr>
<td>database</td>
<td>The name of the DB2 location alias that represents a subset of the members of the data sharing group</td>
<td></td>
</tr>
</tbody>
</table>

If you want to fine-tune Sysplex workload balancing, additional properties are available.

The following IBM Data Server Driver for JDBC and SQLJ Connection or DataSource properties control Sysplex workload balancing.

Table 43. Connection or DataSource properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blockingReadConnectionTimeout</td>
<td>Specifies the amount of time in seconds before a connection socket read times out. This property affects all requests that are sent to the data source after a connection is successfully established. The default is 0, which means that there is no timeout. Set this property to a value greater by a few seconds than the time that is required to execute the longest query in the application.</td>
</tr>
<tr>
<td>loginTimeout</td>
<td>Specifies the maximum time in seconds to wait for a new connection to a data source. After the number of seconds that are specified by loginTimeout have elapsed, the driver closes the connection to the data source. The default is 0, which means that the timeout value is the default system timeout value. The recommended value is five seconds.</td>
</tr>
<tr>
<td>maxRefreshInterval</td>
<td>Specifies the maximum amount of time in seconds between refreshes of the client copy of the server list. The default is 30. The minimum valid value is 1.</td>
</tr>
<tr>
<td>maxTransportObjects</td>
<td>Specifies the maximum number of connections that the requester can make to the data sharing group. The default is -1, which means an unlimited number.</td>
</tr>
</tbody>
</table>
The following IBM Data Server Driver for JDBC and SQLJ configuration properties also control Sysplex workload balancing.

Table 44. Configuration properties for fine-tuning Sysplex workload balancing for direct connections from the IBM Data Server Driver for JDBC and SQLJ to DB2 for z/OS

<table>
<thead>
<tr>
<th>IBM Data Server Driver for JDBC and SQLJ configuration property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>db2.jcc.maxTransportObjectIdleTime</code></td>
<td>Specifies the maximum elapsed time in number of seconds before an idle transport is dropped. The default is 60. The minimum supported value is 0.</td>
</tr>
<tr>
<td><code>db2.jcc.maxTransportObjectWaitTime</code></td>
<td>Specifies the number of seconds that the client will wait for a transport to become available. The default is -1 (unlimited). The minimum supported value is 0.</td>
</tr>
<tr>
<td><code>db2.jcc.minTransportObjects</code></td>
<td>Specifies the lower limit for the number of transport objects in a global transport object pool. The default value is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.</td>
</tr>
</tbody>
</table>

Example of enabling DB2 for z/OS Sysplex workload balancing in Java applications

Java client setup for Sysplex workload balancing includes setting several IBM Data Server Driver for JDBC and SQLJ properties.

The following examples demonstrate setting up Java client applications for Sysplex workload balancing for high availability.

Before you can set up the client, you need to configure the following server software:

- WLM for z/OS
  For workload balancing to work efficiently, DB2 work needs to be classified. Classification applies to the first non-SET SQL statement in each transaction. Among the areas by which you need to classify the work are:
  - Authorization ID
  - Client info properties
  - Stored procedure name
  
  The stored procedure name is used for classification only if the first statement that is issued by the client in the transaction is an SQL CALL statement.

  For a complete list of classification attributes, see the information on classification attributes at the following URL:

- DB2 for z/OS, set up for data sharing

Example of setup with WebSphere Application Server

This example assumes that you are using WebSphere Application Server. The minimum level of WebSphere Application Server is Version 5.1.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support the Sysplex workload balancing by following these steps:
   a. Issue the following command in a command line window
      `java com.ibm.db2.jcc.DB2Jcc -version`
b. Find a line in the output like this, and check that nnn is 3.50 or later.

2. Set the IBM Data Server Driver for JDBC and SQLJ data source property enableSysplexWLB to enable the Sysplex workload balancing.

   In the WebSphere Application Server administrative console, set the following properties for the DataSource that your application uses to connect to the data source. Start with settings similar to these:

   **Table 45. Example of data source property settings for IBM Data Server Driver for JDBC and SQLJ Sysplex workload balancing for DB2 for z/OS**

<table>
<thead>
<tr>
<th>Property</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>enableSysplexWLB</td>
<td>true</td>
</tr>
<tr>
<td>maxRefreshInterval</td>
<td>30</td>
</tr>
<tr>
<td>maxTransportObjects</td>
<td>80</td>
</tr>
</tbody>
</table>

   Use property maxTransportObjects to limit the total number of connections to the DB2 for z/OS data sharing group.

   **Recommendation:** Set maxTransportObjects to a value that is larger than the MaxConnections value for the WebSphere Application Server connection pool. Doing so allows workload balancing to occur between data sharing members without the need to open and close connections to DB2.

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a DB2JccConfiguration.properties file by following these steps:
   a. Create a DB2JccConfiguration.properties file or edit the existing DB2JccConfiguration.properties file.
   b. Set the db2.jcc.maxTransportObjects configuration property only if multiple DataSource objects are defined that point to the same data sharing group, and the number of connections across the different DataSource objects needs to be limited.
      
      Start with a setting similar to this one:
      
      ```
      db2.jcc.maxTransportObjects=500
      ```
   c. Add the directory path for DB2JccConfiguration.properties to the WebSphere Application Server IBM Data Server Driver for JDBC and SQLJ classpath.
   d. Restart WebSphere Application Server.

**Example of setup for DriverManager connections**

This example assumes that you are using the DriverManager interface to establish a connection.

Follow these steps to set up the client:

1. Verify that the IBM Data Server Driver for JDBC and SQLJ is at the correct level to support the Sysplex workload balancing by following these steps:
   a. Issue the following command in a command line window
      ```
      java com.ibm.db2.jcc.DB2Jcc -version
      ```
   b. Find a line in the output like this, and check that nnn is 3.50 or later. A minimum driver level of 3.50 is required for using Sysplex workload balancing for DriverManager connections.
c. [jcc] Driver: IBM Data Server Driver for JDBC and SQLJ Architecture

2. Set the IBM Data Server Driver for JDBC and SQLJ Connection property enableSysplexWLB to enable workload balancing. You can also use property maxTransportObjects to limit the total number of connections to the DB2 for z/OS data sharing group, and maxRefreshInterval to specify the maximum amount of time between refreshes of the client copy of the server list. A minimum driver level of 3.58 is required for using maxRefreshInterval.

```java
java.util.Properties properties = new java.util.Properties();
properties.put("user", "xxxx");
properties.put("password", "yyyy");
properties.put("enableSysplexWLB", "true");
properties.put("maxTransportObjects", "80");
properties.put("maxRefreshInterval", "30");
java.sql.Connection con =
    java.sql.DriverManager.getConnection(url, properties);
```

3. Set IBM Data Server Driver for JDBC and SQLJ configuration properties to fine-tune workload balancing for all DataSource or Connection instances that are created under the driver. Set the configuration properties in a DB2JccConfiguration.properties file by following these steps:
   a. Create a DB2JccConfiguration.properties file or edit the existing DB2JccConfiguration.properties file.
   b. Set the db2.jcc.maxTransportObjects configuration property only if multiple DataSource objects are defined that point to the same data sharing group, and the number of connections across the different DataSource objects needs to be limited.
      Start with a setting similar to this one:
      `db2.jcc.maxTransportObjects=500`
   c. Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation.

**Operation of Sysplex workload balancing for connections from Java clients to DB2 for z/OS servers**

Sysplex workload balancing (also called transaction-level workload balancing) for connections to DB2 for z/OS contributes to high availability by balancing work among members of a data sharing group at the start of a transaction.

The following overview describes the steps that occur when a client connects to a DB2 for z/OS Sysplex, and Sysplex workload balancing is enabled:

1. When the client first establishes a connection using the sysplex-wide IP address called the group IP address, or when a connection is reused by another connection object, the server returns member workload distribution information.
   The default lifespan of the cached server list is 30 seconds.

2. At the start of a new transaction, the client reads the cached server list to identify a member that has untapped capacity, and looks in the transport pool for an idle transport that is tied to the under-utilized member. (An idle transport is a transport that has no associated connection object.)
   • If an idle transport is available, the client associates the connection object with the transport.
   • If, after a user-configurable timeout, no idle transport is available in the transport pool and no new transport can be allocated because the transport pool has reached its limit, an error is returned to the application.
3. When the transaction runs, it accesses the member that is tied to the transport.

4. When the transaction ends, the client verifies with the server that transport reuse is still allowed for the connection object.

5. If transport reuse is allowed, the server returns a list of SET statements for special registers that apply to the execution environment for the connection object.

   The client caches these statements, which it replays in order to reconstruct the execution environment when the connection object is associated with a new transport.

6. The connection object is then disassociated from the transport.

7. The client copy of the server list is refreshed when a new connection is made, or every 30 seconds.

8. When workload balancing is required for a new transaction, the client uses the same process to associate the connection object with a transport.

**Operation of automatic client reroute for connections from Java clients to DB2 for z/OS**

Automatic client reroute support provides failover support when an IBM data server client loses connectivity to a member of a DB2 for z/OS Sysplex. Automatic client reroute enables the client to recover from a failure by attempting to reconnect to the database through any available member of the Sysplex.

Automatic client reroute is enabled by default when Sysplex workload balancing is enabled.

Client support for automatic client reroute is available in IBM data server clients that have a DB2 Connect license. The DB2 Connect server is not required to perform automatic client reroute.

Automatic client reroute for connections to DB2 for z/OS operates in the following way:

1. As part of the response to a COMMIT request from the client, the data server returns:
   - An indicator that specifies whether transports can be reused. Transports can be reused if there are no resources remaining, such as held cursors.
   - SET statements that the client can use to replay the connection state during transport reuse.

2. If the first SQL statement in a transaction fails, and transports can be reused:
   - No error is reported to the application.
   - The failing SQL statement is executed again.
   - The SET statements that are associated with the logical connection are replayed to restore the connection state.

3. If an SQL statement that is not the first SQL statement in a transaction fails, and transports can be reused:
   - The transaction is rolled back.
   - The application is reconnected to the data server.
   - The SET statements that are associated with the logical connection are replayed to restore the connection state.
   - SQL error -30108 (for Java) or SQL30108N (for non-Java clients) is returned to the application to notify it of the rollback and successful reconnection. The application needs to include code to retry the failed transaction.
4. If an SQL statement that is not the first SQL statement in a transaction fails, and transports cannot be reused:
   - The logical connection is returned to its initial, default state.
   - SQL error -30081 (for Java) or SQL30081N (for non-Java clients) is returned to the application to notify it that reconnection was unsuccessful. The application needs to reconnect to the data server, reestablish the connection state, and retry the failed transaction.

5. If connections to all members of the data sharing member list have been tried, and none have succeeded, a connection is tried using the URL that is associated with the data sharing group, to determine whether any members are now available.

**Application programming requirements for high availability for connections from Java clients to DB2 for z/OS servers**

Failover for automatic client reroute can be seamless or non-seamless. If failover for connections to DB2 for z/OS is not seamless, you need to add code to account for the errors that are returned when failover occurs.

If failover is not seamless, and a connection is reestablished with the server, SQLCODE -30108 (SQL30108N) is returned to the application. All work that occurred within the current transaction is rolled back. In the application, you need to:
   - Check the reason code that is returned with the -30108 error to determine whether special register settings on the failing data sharing member are carried over to the new (failover) data sharing member. Reset any special register values that are not current.
   - Execute all SQL operations that occurred since the previous commit operation.

The following conditions must be satisfied for seamless failover to occur for direct connections to DB2 for z/OS:
   - The application language is Java, CLI, or .NET.
   - The connection is not in a transaction. That is, the failure occurs when the first SQL statement in the transaction is executed.
   - The data server allows transport reuse at the end of the previous transaction. An exception to this condition is if transport reuse is not granted because the application was bound with KEEP_DYNAMIC(YES).
   - All global session data is closed or dropped.
   - There are no open, held cursors.
   - If the application uses CLI, the application cannot perform actions that require the driver to maintain a history of previously called APIs in order to replay the SQL statement. Examples of such actions are specifying data at execution time, performing compound SQL, or using array input.
   - The application is not a stored procedure.
   - The application is not running in a Federated environment.
   - Two-phase commit is used, if transactions are dependent on the success of previous transactions. When a failure occurs during a commit operation, the client has no information about whether work was committed or rolled back at the server. If each transaction is dependent on the success of the previous transaction, use two-phase commit. Two-phase commit requires the use of XA support.
Chapter 11. Java 2 Platform, Enterprise Edition

The Java 2 Platform, Enterprise Edition (J2EE), reduces the cost and complexity of developing multi-tier services.

In today's global business environment, organizations need to extend their reach, lower their costs, and lower their response times by providing services that are easily accessible to their customers, employees, suppliers, and other business partners. These services need to have the following characteristics:

- Highly available, to meet the requirements of global business environment
- Secure, to protect the privacy of the users and the integrity of the enterprise
- Reliable and scalable, so that business transactions are accurately and promptly processed

In most cases, these services are provided with the help of multi-tier applications with each tier serving a specific purpose.

J2EE achieves these benefits by defining a standard architecture that is delivered as the following elements:

- J2EE Application Model, a standard application model for developing multi-tier, thin-client services
- J2EE Platform, a standard platform for hosting J2EE applications
- J2EE Compatibility Test Suite for verifying that a J2EE platform product complies with the J2EE platform standard
- J2EE Reference Implementation for demonstrating the capabilities of J2EE, and for providing an operational definition of the J2EE platform

Application components of Java 2 Platform, Enterprise Edition support

The Java 2 Platform, Enterprise Edition (J2EE) provides the runtime environment for hosting J2EE applications.

The runtime environment defines four application component types that a J2EE product must support:

- Application clients are Java programming language programs that are typically GUI programs that execute on a desktop computer. Application clients have access to all of the facilities of the J2EE middle tier.
- Applets are GUI components that typically execute in a web browser, but can execute in a variety of other applications or devices that support the applet programming model.
- Servlets, JavaServer Pages (JSPs), filters, and web event listeners typically execute in a web server and might respond to HTTP requests from web clients. Servlets, JSPs, and filters can be used to generate HTML pages that are an application's user interface. They can also be used to generate XML or other format data that is consumed by other application components. Servlets, pages created with the JSP technology, web filters, and web event listeners are referred to collectively in this specification as web components. Web applications are composed of web components and other data such as HTML pages.
Enterprise JavaBeans (EJB) components execute in a managed environment that supports transactions. Enterprise beans typically contain the business logic for a J2EE application.

The application components listed above can divided into three categories, based on how they can be deployed and managed:

- Components that are deployed, managed, and executed on a J2EE server.
- Components that are deployed, managed on a J2EE server, but are loaded to and executed on a client machine.
- Components whose deployment and management are not completely defined by this specification. Application clients can be under this category.

The runtime support for these components is provided by containers.

**Java 2 Platform, Enterprise Edition containers**

A container provides a federated view of the underlying Java 2 Platform, Enterprise Edition (J2EE) APIs to the application components.

A typical J2EE product will provide a container for each application component type; application client container, applet container, web container, and enterprise bean container. The container tools also understand the file formats for packaging the application components for deployment.

The specification requires that these containers provide a Java-compatible runtime environment. This specification defines a set of standard services that each J2EE product must support. These standard services are:

- HTTP service
- HTTPS service
- Java transaction API
- Remote invocation method
- Java IDL
- JDBC API
- Java message service
- Java naming and directory interface
- JavaMail
- JavaBeans activation framework
- Java API for XML parsing
- Connector architecture
- Java authentication and authorization service

**Java 2 Platform, Enterprise Edition Server**

One part of a Java 2 Platform, Enterprise Edition (J2EE) container is a server.

A J2EE Product Provider typically implements the J2EE server-side functionality. The J2EE client functionality is typically built on J2SE technology.

The IBM WebSphere Application Server is a J2EE-compliant server.

**Java 2 Platform, Enterprise Edition database requirements**

Java 2 Platform, Enterprise Edition requires a data server to store business data. The data server must be accessible through the JDBC API.
The database is accessible from web components, enterprise beans, and application client components. The database need not be accessible from applets.

Java Naming and Directory Interface (JNDI)

JNDI enables Java platform-based applications to access multiple naming and directory services.

It is a part of the Java Enterprise application programming interface (API) set. JNDI makes it possible for developers to create portable applications that are enabled for a number of different naming and directory services, including: file systems; directory services such as Lightweight Directory Access Protocol (LDAP) and Novell Directory Services, and distributed object systems such as the Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), and Enterprise JavaBeans (EJB).

The JNDI API has two parts: an application-level interface used by the application components to access naming and directory services and a service provider interface to attach a provider of a naming and directory service.

Java transaction management

Java 2 Platform, Enterprise Edition (J2EE) simplifies application programming for distributed transaction management.

J2EE includes support for distributed transactions through two specifications, Java Transaction API (JTA) and Java Transaction Service (JTS). JTA is a high-level, implementation-independent, protocol-independent API that allows applications and application servers to access transactions. In addition, the JTA is always enabled.

The IBM Data Server Driver for JDBC and SQLJ and the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows implement the JTA and JTS specifications.

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity distributed transactions are supported to DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, and DB2 for i servers.

JTA specifies standard Java interfaces between a transaction manager and the parties involved in a distributed transaction system: the resource manager, the application server, and the transactional applications.

JTS specifies the implementation of a Transaction Manager which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. JTS propagates transactions using IIOP.

JTA and JTS allow application J2EE servers to take the burden of transaction management off of the component developer. Developers can define the transactional properties of EJB technology based components during design or deployment using declarative statements in the deployment descriptor. The application server takes over the transaction management responsibilities.

In the DB2 and WebSphere Application Server environment, WebSphere Application Server assumes the role of transaction manager, and DB2 acts as a resource manager. WebSphere Application Server implements JTS and part of JTA.
and the JDBC drivers also implement part of JTA so that WebSphere Application Server and DB2 can provide coordinated distributed transactions.

It is not necessary to configure DB2 to be JTA-enabled in the WebSphere Application Server environment because the JDBC drivers automatically detect this environment.

The DB2 JDBC Type 2 Driver provides these two DataSource classes:

- COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource
- COM.ibm.db2.jdbc.DB2XADataSource

The IBM Data Server Driver for JDBC and SQLJ provides these two DataSource classes:

- com.ibm.db2.jcc.DB2ConnectionPoolDataSource
- com.ibm.db2.jcc.DB2XADataSource

WebSphere Application Server provides pooled connections to databases. If the application will be involved in a distributed transaction, the com.ibm.db2.jdbc.DB2XADataSource class should be used when defining DB2 data sources within the WebSphere Application Server.

For the detail information about how to configure the WebSphere Application Server with DB2, refer to WebSphere Application Server InfoCenter at:

**Example of a distributed transaction that uses JTA methods**

Distributed transactions typically involve multiple connections to the same data source or different data sources, which can include data sources from different manufacturers.

The best way to demonstrate distributed transactions is to contrast them with local transactions. With local transactions, a JDBC application makes changes to a database permanent and indicates the end of a unit of work in one of the following ways:

- By calling the Connection.commit or Connection.rollback methods after executing one or more SQL statements
- By calling the Connection.setAutoCommit(true) method at the beginning of the application to commit changes after every SQL statement

![Figure 50](image)

oulines code that executes local transactions.

```java
con1.setAutoCommit(false); // Set autocommit off
// execute some SQL
...  
con1.commit(); // Commit the transaction
// execute some more SQL
...  
con1.rollback(); // Roll back the transaction
con1.setAutoCommit(true); // Enable commit after every SQL statement
...  
// Execute some more SQL, which is automatically committed after
// every SQL statement.
```

*Figure 50. Example of a local transaction*

In contrast, applications that participate in distributed transactions cannot call the Connection.commit, Connection.rollback, or Connection.setAutoCommit(true) methods within the distributed transaction. With distributed transactions, the
Connection.commit or Connection.rollback methods do not indicate transaction boundaries. Instead, your applications let the application server manage transaction boundaries.

**Figure 51** demonstrates an application that uses distributed transactions. While the code in the example is running, the application server is also executing other EJBs that are part of this same distributed transaction. When all EJBs have called utx.commit(), the entire distributed transaction is committed by the application server. If any of the EJBs are unsuccessful, the application server rolls back all the work done by all EJBs that are associated with the distributed transaction.

```java
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a distributed transaction.
utx.begin();
...
// Execute some SQL with one Connection object.
// Do not call Connection methods commit or rollback.
...
// Use the commit method on the UserTransaction object to
// drive all transaction branches to commit and indicate
// the end of the distributed transaction.
utx.commit();
...
```

*Figure 51. Example of a distributed transaction under an application server*

**Figure 52** illustrates a program that uses JTA methods to execute a distributed transaction. This program acts as the transaction manager and a transactional application. Two connections to two different data sources do SQL work under a single distributed transaction.

```java
class XASample
{
    javax.sql.XADataSource xaDS1;
    javax.sql.XADataSource xaDS2;
    javax.sql.XAConnection xacconn1;
    javax.sql.XAConnection xacconn2;
    javax.transaction.xa.XAResource xares1;
    javax.transaction.xa.XAResource xares2;
    java.sql.Connection conn1;
    java.sql.Connection conn2;

    public static void main(String args[]) throws java.sql.SQLException
    {
        XASample xat = new XASample();
xat.runThis(args);
    }

    // As the transaction manager, this program supplies the global
    // transaction ID and the branch qualifier. The global
    // transaction ID and the branch qualifier must not be
    // equal to each other, and the combination must be unique for
    // this transaction manager.
    public void runThis(String[] args)
    {
        byte[] gtrid = new byte[] { 0x44, 0x11, 0x55, 0x66 };
        byte[] bqual = new byte[] { 0x00, 0x22, 0x00 };
        int rc1 = 0;
        int rc2 = 0;

        javax.transaction.UserTransaction utx;
        // Use the begin method on a UserTransaction object to indicate
        // the beginning of a distributed transaction.
        utx.begin();
        ...
        // Execute some SQL with one Connection object.
        // Do not call Connection methods commit or rollback.
        ...
        // Use the commit method on the UserTransaction object to
        // drive all transaction branches to commit and indicate
        // the end of the distributed transaction.
        utx.commit();
        ...
    }

    // Figure 52. Example of a distributed transaction that uses the JTA
```
try {
  javax.naming.InitialContext context = new javax.naming.InitialContext();
  /* Note that javax.sql.XADataSource is used instead of a specific
   * driver implementation such as com.ibm.db2.jcc.DB2XDataSource.
   */
  xadS1 = (javax.sql.XADataSource)context.lookup("checkingAccounts");
  xadS2 = (javax.sql.XADataSource)context.lookup("savingsAccounts");
  // The XADatasource contains the user ID and password.
  // Get the XAConnection object from each XADataSource
  xaconn1 = xadS1.getXAConnection();
  xaconn2 = xadS2.getXAConnection();
  // Get the java.sql.Connection object from each XAConnection
  conn1 = xaconn1.getConnection();
  conn2 = xaconn2.getConnection();
  // Get the XAResource object from each XAConnection
  xares1 = xaconn1.getXAResource();
  xares2 = xaconn2.getXAResource();
  // Create the Xid object for this distributed transaction.
  // This example uses the com.ibm.db2.jcc.DB2Xid implementation
  // of the Xid interface. This Xid can be used with any JDBC driver
  // that supports JTA.
  javax.transaction.xa.Xid xid1 =
      new com.ibm.db2.jcc.DB2Xid(100, gtrid, bqual);
  // Start the distributed transaction on the two connections.
  // The two connections do NOT need to be started and ended together.
  // They might be done in different threads, along with their SQL operations.
  xares1.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
  xares2.start(xid1, javax.transaction.xa.XAResource.TMNOFLAGS);
  ...
  // Do the SQL operations on connection 1.
  // Do the SQL operations on connection 2.
  ...
  // Now end the distributed transaction on the two connections.
  xares1.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);
  xares2.end(xid1, javax.transaction.xa.XAResource.TMSUCCESS);
  // If connection 2 work had been done in another thread,
  // a thread.join() call would be needed here to wait until the
  // connection 2 work is done.
}
try {
  // Now prepare both branches of the distributed transaction.
  // Both branches must prepare successfully before changes
  // can be committed.
  // If the distributed transaction fails, an XAException is thrown.
  if(rc1 == javax.transaction.xa.XAResourceXA_OK)
    { // Prepare was successful. Prepare the second connection.
         rc2 = xares2.prepare(xid1);
      if(rc2 == javax.transaction.xa.XAResourceXA_OK)
        { // Both connections prepared successfully and neither was read-only.
            xares1.commit(xid1, false);
            xares2.commit(xid1, false);
        }
      else if(rc2 == javax.transaction.xa.XAExceptionXA_RDONLY)
        { // The second connection is read-only, so just commit the
            // first connection.
            xares1.commit(xid1, false);
        }
else if(rc1 == javax.transaction.xa.XAException.XA_RDONLY)
{
  // SQL for the first connection is read-only (such as a SELECT).
  // The prepare committed it. Prepare the second connection.
  rc2 = xares2.prepare(xid1);
  if(rc2 == javax.transaction.xa.XAResource.XA_OK)
  { // The first connection is read-only but the second is not.
    // Commit the second connection.
    xares2.commit(xid1, false);
  }
  else if(rc2 == javax.transaction.xa.XAException.XA_RDONLY)
  { // Both connections are read-only, and both already committed,
    // so there is nothing more to do.
  }
}
}

try
{
  conn1.close();
  xaconn1.close();
}
catch (Exception e)
{
  System.out.println("Failed to close connection 1: "+ e.toString());
  e.printStackTrace();
}

try
{
  conn2.close();
  xaconn2.close();
}
catch (Exception e)
{
  System.out.println("Failed to close connection 2: "+ e.toString());
  e.printStackTrace();
}

catch (javax.transaction.xa.XAException xae)
{ // Distributed transaction failed, so roll it back.
  // Report XAException on prepare/commit.
  System.out.println("Distributed transaction prepare/commit failed. " +
                     "Rolling it back.");
  System.out.println("XAException error code = "+ xae.errorCode);
  System.out.println("XAException message = "+ xae.getMessage());
  try
  {
    xares1.rollback(xid1);
  }
  catch (javax.transaction.xa.XAException xael)
  { // Report failure of rollback.
    System.out.println("distributed Transaction rollback xares1 failed");
    System.out.println("XAException error code = "+ xael.errorCode);
    System.out.println("XAException message = "+ xael.getMessage());
  }
  try
  {
    xares2.rollback(xid1);
  }
  catch (javax.transaction.xa.XAException xae2)
  { // Report failure of rollback.
    System.out.println("distributed Transaction rollback xares2 failed");
    System.out.println("XAException error code = "+ xae2.errorCode);
    System.out.println("XAException message = "+ xae2.getMessage());
  }
}

try
{
  conn1.close();
  xaconn1.close();
}
catch (Exception e)
{ System.out.println("Failed to close connection 1: "+ e.toString());
  e.printStackTrace();
}
try
{ conn2.close();
  xaconn2.close();
}
catch (Exception e)
{ System.out.println("Failed to close connection 2: "+ e.toString());
  e.printStackTrace();
}
System.out.println("SQLException caught: " + sqe.getMessage());
sqe.printStackTrace();
}
catch (javax.transaction.xa.XAException xae)
{
    System.out.println("XA error is " + xae.getMessage());
    xae.printStackTrace();
}
catch (javax.naming.NamingException nme)
{
    System.out.println(" Naming Exception: " + nme.getMessage());
}

Recommendation: For better performance, complete a distributed transaction before you start another distributed or local transaction.

**Setting the transaction timeout value for an XAResource instance**

Use the XAResource.setTransactionTimeout method to reduce occurrences of deadlocks in a DB2 database that is the target of distributed transactions.

A distributed transaction to DB2 Database for Linux, UNIX, and Windows that ends, but cannot be prepared, is not an indoubt transaction. Therefore, the transaction manager cannot recover the transaction, and the DB2 resource manager does not put the transaction in its list of indoubt transactions. The DB2 resource manager does not roll back the transaction immediately, but waits until all connections to the database are released. During this period of inactivity, the transaction continues to hold locks on the database. If the transaction manager does not disconnect all connections to the database to allow rollback, the ended transaction continues to lock database records. If another application attempts to access those locked records, a deadlock can occur.

In a Java application that uses distributed transactions and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, you can prevent a transaction from holding locks on a database indefinitely by calling the XAResource.setTransactionTimeout method to set a timeout value on transactions. To do that, follow these steps:

1. On the DB2 Database for Linux, UNIX, and Windows instance, issue this command to cause the instance to check for timeout values.

   ```
   DB2 UPDATE DBM CFG USING RESYNC_INTERVAL seconds
   ```

   *seconds* needs to be less than the minimum timeout value that you set for a transaction.

2. In your application, after you create an XAResource object, call the XAResource.setTransactionTimeout method to set the timeout value. You can check the current timeout value by calling XAResource.getTransactionTimeout.

**Enterprise Java Beans**

The Enterprise Java beans architecture is a component architecture for the development and deployment of component-based distributed business applications.
Applications that are written using the Enterprise Java beans architecture can be written once, and then deployed on any server platform that supports the Enterprise Java beans specification. Java 2 Platform, Enterprise Edition (J2EE) applications implement server-side business components using Enterprise Java beans (EJBs) that include session beans and entity beans.

Session beans represent business services and are not shared between users. Entity beans are multi-user, distributed transactional objects that represent persistent data. The transactional boundaries of a EJB application can be set by specifying either container-managed or bean-managed transactions.

The sample program AccessEmployee.ear uses Enterprise Java beans to implement a J2EE application to access a data source. You can find this sample in the SQLLIB/samples/websphere directory.

The EJB sample application provides two business services. One service allows the user to access information about an employee (which is stored in the EMPLOYEE table of the sample database) through that employee's employee number. The other service allows the user to retrieve a list of the employee numbers, so that the user can obtain an employee number to use for querying employee data.

The following sample uses EJBs to implement a J2EE application to access a data source. The sample utilizes the Model-View-Controller (MVC) architecture, which is a commonly-used GUI architecture. The JSP is used to implement the view (the presentation component). A servlet acts as the controller in the sample. It controls the workflow and delegates the user's request to the model, which is implemented using EJBs. The model component of the sample consists of two EJBs, one session bean and one entity bean. The container-managed persistence (CMP) bean, Employee, represents the distributed transactional objects that represent the persistent data in the EMPLOYEE table of the sample database. The term container-managed persistence means that the EJB container handles all database access required by the entity bean. The bean's code contains no database access (SQL) calls. As a result, the bean's code is not tied to a specific persistent storage mechanism (database). The session bean, AccessEmployee, acts as the Façade of the entity bean and provides provide a uniform client access strategy. This Façade design reduces the network traffic between the EJB client and the entity bean and is more efficient in distributed transactions than if the EJB client accesses the entity bean directly. Access to the database server can be provided from the session bean or entity bean. The two services of the sample application demonstrate both approaches to accessing the database server. In the first service, the entity bean is used:

```java
//====================================================
// This method returns an employee's information by
// interacting with the entity bean located by the
// provided employee number
public EmployeeInfo getEmployeeInfo(String empNo)
throws java.rmi.RemoteException
{  
  Employee employee = null;
  try
  }
  employee = employeeHome.findByPrimaryKey(new EmployeeKey(empNo));
  EmployeeInfo empInfo = new EmployeeInfo(empNo);
  //set the employee's information to the dependent value object
  empInfo.setEmpno(employee.getEmpno());
  empInfo.setFirstName (employee.getFirstName());
  empInfo.setMidInit(employee.getMidInit());
  empInfo.setLastName(employee.getLastName());
  empInfo.setWorkDept(employee.getWorkDept());
```
empInfo.setPhoneNo(employee.getPhoneNo());
empInfo.setHireDate(employee.getHireDate());
empInfo.setJob(employee.getJob());
empInfo.setEdLevel(employee.getEdLevel());
empInfo.setSex(employee.getSex());
empInfo.setBirthDate(employee.getBirthDate());
empInfo.setSalary(employee.getSalary());
empInfo.setBonus(employee.getBonus());
empInfo.setComm(employee.getComm());
return empInfo;
}
catch (java.rmi.RemoteException rex)
{
......

In the second service, which displays employee numbers, the session bean, AccessEmployee, directly accesses the database table.

//====================================================================
/* Get the employee number list.  
 * @return Collection
 */
public Collection getEmpNoList()
{
ResultSet rs = null;
PreparedStatement ps = null;
Vector list = new Vector();
DataSource ds = null;
Connection con = null;
try
{
    ds = getDataSource();
    con = ds.getConnection();
    String schema = getEnvProps(DBschema);
    String query = "Select EMPNO from " + schema + ".EMPLOYEE";
    ps = con.prepareStatement(query);
    ps.executeQuery();
    rs = ps.getResultSet();
    EmployeeKey pk;
    while (rs.next())
    {
        pk = new EmployeeKey();
        pk.employeeId = rs.getString(1);
        list.addElement(pk.employeeId);
    }
    rs.close();
    return list;
}
Chapter 12. JDBC and SQLJ connection pooling support

Connection pooling is part of JDBC DataSource support, and is supported by the IBM Data Server Driver for JDBC and SQLJ.

The IBM Data Server Driver for JDBC and SQLJ provides a factory of pooled connections that are used by WebSphere Application Server or other application servers. The application server actually does the pooling. Connection pooling is completely transparent to a JDBC or SQLJ application.

Connection pooling is a framework for caching physical data source connections, which are equivalent to DB2 threads. When JDBC reuses physical data source connections, the expensive operations that are required for the creation and subsequent closing of java.sql.Connection objects are minimized.

Without connection pooling, each java.sql.Connection object represents a physical connection to the data source. When the application establishes a connection to a data source, DB2 creates a new physical connection to the data source. When the application calls the java.sql.Connection.close method, DB2 terminates the physical connection to the data source.

In contrast, with connection pooling, a java.sql.Connection object is a temporary, logical representation of a physical data source connection. The physical data source connection can be serially reused by logical java.sql.Connection instances. The application can use the logical java.sql.Connection object in exactly the same manner as it uses a java.sql.Connection object when there is no connection pooling support.

With connection pooling, when a JDBC application invokes the DataSource.getConnection method, the data source determines whether an appropriate physical connection exists. If an appropriate physical connection exists, the data source returns a java.sql.Connection instance to the application. When the JDBC application invokes the java.sql.Connection.close method, JDBC does not close the physical data source connection. Instead, JDBC closes only JDBC resources, such as Statement or ResultSet objects. The data source returns the physical connection to the connection pool for reuse.

Connection pooling can be homogeneous or heterogeneous.

With homogeneous pooling, all Connection objects that come from a connection pool should have the same properties. The first logical Connection that is created with the DataSource has the properties that were defined for the DataSource. However, an application can change those properties. When a Connection is returned to the connection pool, an application server or a pooling module should reset the properties to their original values. However, an application server or pooling module might not reset the changed properties. The JDBC driver does not modify the properties. Therefore, depending on the application server or pool module design, a reused logical Connection might have the same properties as those that are defined for the DataSource or different properties.

With heterogeneous pooling, Connection objects with different properties can share the same connection pool.
Chapter 13. JDBC and SQLJ reference information

The IBM implementations of JDBC and SQLJ provide a number of application programming interfaces, properties, and commands for developing JDBC and SQLJ applications.

Data types that map to database data types in Java applications

To write efficient JDBC and SQLJ programs, you need to use the best mappings between Java data types and table column data types.

The following tables summarize the mappings of Java data types to JDBC and database data types for a DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix (IDS) system.

Data types for updating table columns

The following table summarizes the mappings of Java data types to database data types for PreparedStatement.setXXX or ResultSet.updateXXX methods in JDBC programs, and for input host expressions in SQLJ programs. When more than one Java data type is listed, the first data type is the recommended data type.

<table>
<thead>
<tr>
<th>Java data type</th>
<th>Database data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>boolean</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int, java.lang.Integer</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long, java.lang.Long</td>
<td>BIGINT</td>
</tr>
<tr>
<td>float, java.lang.Float</td>
<td>REAL</td>
</tr>
<tr>
<td>double, java.lang.Double</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECIMAL((p, s))</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECFLOAT((n))</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CHAR((n))</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>GRAPHIC((m))</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR((n))</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARGRAPHIC((m))</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>XML</td>
</tr>
<tr>
<td>byte[]</td>
<td>CHAR((n)) FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARCHAR((n)) FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>BINARY((n))</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY((n))</td>
</tr>
<tr>
<td>byte[]</td>
<td>BLOB</td>
</tr>
<tr>
<td>byte[]</td>
<td>ROWID</td>
</tr>
<tr>
<td>byte[]</td>
<td>XML</td>
</tr>
</tbody>
</table>
Table 46. Mappings of Java data types to database server data types for updating database tables (continued)

<table>
<thead>
<tr>
<th>Java data type</th>
<th>Database data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.sql.Blob</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Blob</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>DBCLOB</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.io.StringReader</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.io.InputStream</td>
<td>XML</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2RowID (deprecated)</td>
<td>ROWID</td>
</tr>
<tr>
<td>java.sql.RowId</td>
<td>ROWID</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2Xml (deprecated)</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.SQLXML</td>
<td>XML</td>
</tr>
</tbody>
</table>

Notes:

1. The database server has no exact equivalent for the Java boolean or byte data types, but the best fit is SMALLINT.
2. \( p \) is the decimal precision and \( s \) is the scale of the table column.
   You should design financial applications so that java.math.BigDecimal columns map to DECIMAL columns. If you know the precision and scale of a DECIMAL column, updating data in the DECIMAL column with data in a java.math.BigDecimal variable results in better performance than using other combinations of data types.
3. \( n=16 \) or \( n=34 \).
4. DECFLOAT is valid for connections to DB2 Version 9.1 for z/OS, DB2 V9.5 for Linux, UNIX, and Windows, or DB2 for i V6R1, or later database servers. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.
5. \( n<=254 \).
6. \( m<=127 \).
7. \( n<=32672 \).
8. \( m<=16336 \).
9. This mapping is valid only if the database server can determine the data type of the column.
10. XML is valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 V9.1 for Linux, UNIX, and Windows or later database servers.
11. BIGINT is valid for connections to DB2 Version 9.1 for z/OS or later database servers, DB2 V9.1 for Linux, UNIX, and Windows or later database servers, and all supported DB2 for i database servers.
12. BINARY and VARBINARY are valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 for i5/OS® V5R3 and later database servers.

Data types for retrieval from table columns

The following table summarizes the mappings of DB2 or IDS data types to Java data types for ResultSet.getXXX methods in JDBC programs, and for iterators in SQLJ programs. This table does not list Java numeric wrapper object types, which are retrieved using ResultSet.getObject.
Table 47. Mappings of database server data types to Java data types for retrieving data from database server tables

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Recommended Java data type or Java object type</th>
<th>Other supported Java data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short</td>
<td>byte, int, long, float, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
<td>short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long</td>
<td>int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>DECIMAL(p,s) or NUMERIC(p,s)</td>
<td>java.math.BigDecimal</td>
<td>long, int, short, byte, float, double, boolean, java.lang.String</td>
</tr>
<tr>
<td>DECFLOAT(n)</td>
<td>java.math.BigDecimal</td>
<td>long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
<td>long, int, short, byte, double, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
<td>long, int, short, byte, float, java.math.BigDecimal, boolean, java.lang.String</td>
</tr>
<tr>
<td>CHAR(n) FOR BIT DATA</td>
<td>byte[]</td>
<td>java.lang.String, java.io.InputStream, java.io.Reader</td>
</tr>
<tr>
<td>VARCHAR(n) FOR BIT DATA</td>
<td>byte[]</td>
<td>java.lang.String, java.io.InputStream, java.io.Reader</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>byte[]</td>
<td>None</td>
</tr>
<tr>
<td>VARBINARY(n)</td>
<td>byte[]</td>
<td>None</td>
</tr>
<tr>
<td>CLOB(n)</td>
<td>java.sql.Clob</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>java.sql.Blob</td>
<td>byte[]</td>
</tr>
</tbody>
</table>
### Table 47. Mappings of database server data types to Java data types for retrieving data from database server tables (continued)

<table>
<thead>
<tr>
<th>SQL data type</th>
<th>Recommended Java data type or Java object type</th>
<th>Other supported Java data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBCLOB(n)</td>
<td>No exact equivalent. Use java.sql.Clob.</td>
<td></td>
</tr>
<tr>
<td>ROWID</td>
<td>java.sql.RowId</td>
<td>byte[], com.ibm.db2.jcc.DB2RowID (deprecated)</td>
</tr>
<tr>
<td>XML</td>
<td>java.sql.SQLXML</td>
<td>byte[], java.lang.String, java.io.InputStream, java.io.Reader</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
<td>java.sql.String, java.sql.Timestamp</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
<td>java.sql.String, java.sql.Timestamp</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
<td>java.sql.String, java.sql.Date, java.sql.Time, java.sql.Timestamp</td>
</tr>
</tbody>
</table>

#### Notes:
1. n = 16 or n = 34.
2. DECFLOAT is valid for connections to DB2 Version 9.1 for z/OS, DB2 V9.5 for Linux, UNIX, and Windows, or DB2 for i5/OS V6R1, or later database servers. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.
3. This mapping is valid only if the database server can determine the data type of the column.
4. XML is valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 V9.1 for Linux, UNIX, and Windows or later database servers.
5. BIGINT is valid for connections to DB2 Version 9.1 for z/OS or later database servers, DB2 V9.1 for Linux, UNIX, and Windows or later database servers, and all supported DB2 for i database servers.
6. BINARY and VARBINARY are valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 for i5/OS V5R3 or later database servers.

---

### Data types for calling stored procedures and user-defined functions

The following table summarizes mappings of Java data types to JDBC data types and DB2 or IDS data types for calling user-defined function and stored procedure parameters. The mappings of Java data types to JDBC data types are for CallableStatement.registerOutParameter methods in JDBC programs. The mappings of Java data types to database server data types are for parameters in stored procedure or user-defined function invocations.

If more than one Java data type is listed in the following table, the first data type is the recommended data type.

#### Table 48. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions

<table>
<thead>
<tr>
<th>Java data type</th>
<th>JDBC data type</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>BIT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>byte</td>
<td>TINYINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>short, java.lang.Short</td>
<td>SMALLINT</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>int, java.lang.Integer</td>
<td>INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>BIGINT</td>
<td>BIGINT</td>
</tr>
<tr>
<td>float, java.lang.Float</td>
<td>REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>float, java.lang.Float</td>
<td>FLOAT</td>
<td>REAL</td>
</tr>
<tr>
<td>double, java.lang.Double</td>
<td>DOUBLE</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
Table 48. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

<table>
<thead>
<tr>
<th>Java data type</th>
<th>JDBC data type</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.math.BigDecimal</td>
<td>NUMERIC</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>DECIMAL</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>java.types.OTHER</td>
<td>DECFLOAT</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>com.ibm.db2.jcc.DB2Types.DECFLOAT</td>
<td>DECFLOAT</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CHAR</td>
<td>CHAR</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CHAR</td>
<td>GRAPHIC</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR</td>
<td>VARGRAPHIC</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>VARCHAR</td>
<td>VARCHAR</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>LONGVARCHAR</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>byte[]</td>
<td>BINARY</td>
<td>CHAR FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY</td>
<td>VARCHAR FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>BINARY</td>
<td>BINARY</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY</td>
<td>VARBINARY</td>
</tr>
<tr>
<td>byte[]</td>
<td>LONGVARBINARY</td>
<td>VARCHAR FOR BIT DATA</td>
</tr>
<tr>
<td>byte[]</td>
<td>VARBINARY</td>
<td>BLOB</td>
</tr>
<tr>
<td>byte[]</td>
<td>LONGVARBINARY</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>DATE</td>
<td>DATE</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>TIME</td>
<td>TIME</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>TIMESTAMP</td>
<td>TIMESTAMP</td>
</tr>
<tr>
<td>java.sql.Blob</td>
<td>BLOB</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>CLOB</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.sql.Clob</td>
<td>CLOB</td>
<td>DBCLOB</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>None</td>
<td>BLOB</td>
</tr>
<tr>
<td>java.io.StringReader</td>
<td>None</td>
<td>CLOB</td>
</tr>
<tr>
<td>java.io.ByteArrayInputStream</td>
<td>None</td>
<td>CLOB</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2RowID (deprecated)</td>
<td>com.ibm.db2.jcc.DB2Types.ROWID</td>
<td>ROWID</td>
</tr>
<tr>
<td>java.sql.RowId</td>
<td>java.sql.Types.ROWID</td>
<td>ROWID</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2Xml (deprecated)</td>
<td>com.ibm.db2.jcc.DB2Types.XML</td>
<td>XML AS CLOB</td>
</tr>
<tr>
<td>java.sql.SQLXML</td>
<td>java.sql.Types.SQLXML</td>
<td>XML</td>
</tr>
<tr>
<td>java.sql.SQLXML</td>
<td>java.sql.Types.SQLXML</td>
<td>XML AS CLOB</td>
</tr>
<tr>
<td>java.sql.Array</td>
<td>java.sql.Types.ARRAY</td>
<td>ARRAY</td>
</tr>
<tr>
<td>java.sql.ResultSet</td>
<td>com.ibm.db2.jcc.DB2Types.CURSOR</td>
<td>CURSOR type</td>
</tr>
</tbody>
</table>
### Table 48. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

<table>
<thead>
<tr>
<th>Java data type</th>
<th>JDBC data type</th>
<th>SQL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Notes:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>A stored procedure or user-defined function that is defined with a SMALLINT parameter can be invoked with a boolean or byte parameter. However, this is not recommended.</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>DECIMAL parameters in Java routines are valid only for connections to DB2 Version 9.1 for z/OS or later database servers. DECIMAL parameters in Java routines are not supported for connections to for Linux, UNIX, and Windows or DB2 for i. Use of DECIMAL requires the SDK for Java Version 5 (1.5) or later.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>This mapping is valid only if the database server can determine the data type of the column.</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>BINARY and VARBINARY are valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 for i5/OS V5R3 and later database servers.</td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td>BIGINT is valid for connections to DB2 Version 9.1 for z/OS or later database servers, DB2 V9.1 for Linux, UNIX, and Windows or later database servers, and all supported DB2 for i database servers.</td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>ARRAY parameters are supported for stored procedures only.</td>
<td></td>
</tr>
</tbody>
</table>

### Data types in Java stored procedures and user-defined functions

The following table summarizes mappings of the SQL parameter data types in a CREATE PROCEDURE or CREATE FUNCTION statement to the data types in the corresponding Java stored procedure or user-defined function method.

For DB2 Database for Linux, UNIX, and Windows, if more than one Java data type is listed for an SQL data type, only the first Java data type is valid.

For DB2 for z/OS, if more than one Java data type is listed, and you use a data type other than the first data type as a method parameter, you need to include a method signature in the EXTERNAL clause of your CREATE PROCEDURE or CREATE FUNCTION statement that specifies the Java data types of the method parameters.

### Table 49. Mappings of SQL data types in a CREATE PROCEDURE or CREATE FUNCTION statement to data types in the corresponding Java stored procedure or user-defined function program

<table>
<thead>
<tr>
<th>SQL data type in CREATE PROCEDURE or CREATE FUNCTION</th>
<th>Data type in Java stored procedure or user-defined function method</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>short, java.lang.Integer</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int, java.lang.Integer</td>
</tr>
<tr>
<td>REAL</td>
<td>float, java.lang.Float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double, java.lang.Double</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>CHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>CHAR FOR BIT DATA</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARCHAR FOR BIT DATA</td>
<td>byte[]</td>
</tr>
<tr>
<td>BINARY[6]</td>
<td>byte[]</td>
</tr>
<tr>
<td>VARBINARY[6]</td>
<td>byte[]</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
</tbody>
</table>
### Table 49. Mappings of SQL data types in a CREATE PROCEDURE or CREATE FUNCTION statement to data types in the corresponding Java stored procedure or user-defined function program (continued)

<table>
<thead>
<tr>
<th>SQL data type in CREATE PROCEDURE or CREATE FUNCTION</th>
<th>Data type in Java stored procedure or user-defined function method</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>BLOB</td>
<td>java.sql.Blob</td>
</tr>
<tr>
<td>CLOB</td>
<td>java.sql.Clob</td>
</tr>
<tr>
<td>DBCLOB</td>
<td>java.sql.Clob</td>
</tr>
<tr>
<td>ROWID</td>
<td>java.sql.Types.ROWID</td>
</tr>
<tr>
<td>XML AS CLOB</td>
<td>java.sql.Types.SQLXML</td>
</tr>
</tbody>
</table>

#### Notes:
1. For a stored procedure or user-defined function on a DB2 Database for Linux, UNIX, and Windows server, only the first data type is valid.
2. BIGINT is valid for connections to DB2 Version 9.1 for z/OS or later database servers or DB2 V9.1 for Linux, UNIX, and Windows or later database servers.
3. DECIMAL parameters in Java routines are valid only for connections to DB2 Version 9.1 for z/OS or later database servers. DECIMAL parameters in Java routines are not supported for connections to for Linux, UNIX, and Windows or DB2 for i. Use of DECIMAL requires the SDK for Java Version 5 (1.5) or later.
4. BINARY and VARBINARY are valid for connections to DB2 Version 9.1 for z/OS or later database servers.

### Date, time, and timestamp values that can cause problems in JDBC and SQLJ applications

You might receive unexpected results in JDBC and SQLJ applications if you use date, time, and timestamp values that do not correspond to real dates and times.

The following items might cause problems:

- Use of the hour ‘24’ to represent midnight
- Use of a date between October 5, 1582, and October 14, 1582, inclusive

### Problems with using the hour '24' as midnight

The IBM Data Server Driver for JDBC and SQLJ uses Java data types for its internal processing of input and output parameters and ResultSet content in JDBC and SQLJ applications. The Java data type that is used by the driver is based on the best match for the corresponding SQL type when the target SQL type is known to the driver.

For values that are assigned to or retrieved from DATE, TIME, or TIMESTAMP SQL types, the IBM Data Server Driver for JDBC and SQLJ uses java.sql.Date for DATE SQL types, java.sql.Time for TIME SQL types, and java.sql.Timestamp for TIMESTAMP SQL types.

When you assign a string value to a DATE, TIME, or TIMESTAMP target, the IBM Data Server Driver for JDBC and SQLJ uses Java facilities to convert the string value to a java.sql.Date, java.sql.Time, or java.sql.Timestamp value. If a string representation of a date, time, or timestamp value does not correspond to a real date or time, Java adjusts the value to a real date or time value. In particular, Java
adjusts an hour value of '24' to '00' of the next day. This adjustment can result in an exception for a timestamp value of '9999-12-31 24:00:00.0', because the adjusted year value becomes '10000'.

Important: To avoid unexpected results when you assign or retrieve date, time, or timestamp values in JDBC or SQLJ applications, ensure that the values are real date, time, or timestamp values. In addition, do not use '24' as the hour component of a time or timestamp value.

If a value that does not correspond to a real date or time, such as a value with an hour component of '24', is stored in a TIME or TIMESTAMP column, you can avoid adjustment during retrieval by executing the SQL CHAR function against that column in the SELECT statement that defines a ResultSet. Executing the CHAR function converts the date or time value to a character string value on the database side. However, if you use the getTime or getTimestamp method to retrieve that value from the ResultSet, the IBM Data Server Driver for JDBC and SQLJ converts the value to a java.sql.Time or java.sql.Timestamp type, and Java adjusts the value. To avoid date adjustment, execute the CHAR function against the column value, and retrieve the value from the ResultSet with the getString method.

The following examples show the results of updating DATE, TIME, or TIMESTAMP columns in JDBC or SQLJ applications, when the application data does not represent real dates or times.

Table 50. Examples of updating DATE, TIME, or TIMESTAMP SQL values with Java date, time, or timestamp values that do not represent real dates or times

<table>
<thead>
<tr>
<th>String input value</th>
<th>Target type in database</th>
<th>Value sent to table column, or exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008-13-35</td>
<td>DATE</td>
<td>2009-02-04</td>
</tr>
<tr>
<td>25:00:00</td>
<td>TIME</td>
<td>01:00:00</td>
</tr>
<tr>
<td>24:00:00</td>
<td>TIME</td>
<td>00:00:00</td>
</tr>
<tr>
<td>2008-15-36 24:00:00</td>
<td>TIMESTAMP</td>
<td>2009-04-06 05:04:14.0</td>
</tr>
<tr>
<td>9999-12-31 24:00:00.0</td>
<td>TIMESTAMP</td>
<td>Exception, because the adjusted value (10000-01-01 00:00:00.0) exceeds the maximum year of 9999.</td>
</tr>
</tbody>
</table>

The following examples demonstrate the results of retrieving data from TIMESTAMP columns in JDBC or SQLJ applications, when the values in those columns do not represent real dates or times.

Table 51. Results of retrieving DATE, TIME, or TIMESTAMP SQL values that do not represent real dates or times into Java application variables

<table>
<thead>
<tr>
<th>SELECT statement</th>
<th>Value in TIMESTAMP column TS_COL</th>
<th>Target type in application (getXXX method for retrieval)</th>
<th>Value retrieved from table column</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT TS_COL FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>java.sql.Timestamp (getTimestamp)</td>
<td>2000-01-02 00:00:00.000000</td>
</tr>
<tr>
<td>SELECT TS_COL FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>String (getString)</td>
<td>2000-01-02 00:00:00.000000</td>
</tr>
<tr>
<td>SELECT CHAR(TS_COL) FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>java.sql.Timestamp (getTimestamp)</td>
<td>2000-01-02 00:00:00.000000</td>
</tr>
</tbody>
</table>
Table 51. Results of retrieving DATE, TIME, or TIMESTAMP SQL values that do not represent real dates or times into Java application variables (continued)

<table>
<thead>
<tr>
<th>SELECT statement</th>
<th>Value in TIMESTAMP column TS_COL</th>
<th>Target type in application (getXXX method for retrieval)</th>
<th>Value retrieved from table column</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT CHAR(TS_COL) FROM TABLE1</td>
<td>2000-01-01 24:00:00.000000</td>
<td>String (getString)</td>
<td>2000-01-01 24:00:00.000000 (no adjustment by Java)</td>
</tr>
</tbody>
</table>

Problems with using dates in the range October 5, 1582, through October 14, 1582

The Java java.util.Date and java.util.Timestamp classes use the Julian calendar for dates before October 4, 1582, and the Gregorian calendar for dates starting with October 4, 1582. In the Gregorian calendar, October 4, 1582, is followed by October 15, 1582. If a Java program encounters a java.util.Date or java.util.Timestamp value that is between October 5, 1582, and October 14, 1582, inclusive, Java adds 10 days to that date. Therefore, a DATE or TIMESTAMP value in a DB2 table that has a value between October 5, 1582, and October 14, 1582, inclusive, is retrieved in a Java program as a java.util.Date or java.util.Timestamp value between October 15, 1582, and October 24, 1582, inclusive. A java.util.Date or java.util.Timestamp value in a Java program that is between October 5, 1582, and October 14, 1582, inclusive, is stored in a DB2 table as a DATE or TIMESTAMP value between October 15, 1582, and October 24, 1582, inclusive.

Example: Retrieve October 10, 1582, from a DATE column.

```java
// DATETABLE has one date column with one row.
// Its value is 1582-10-10.
java.sql.ResultSet rs =
  statement.executeQuery(select * from DATETABLE);
rs.next();
System.out.println(rs.getDate(1)); // Value is retrieved as 1582-10-20
```

Example: Store October 10, 1582, in a DATE column.

```java
java.sql.Date d = java.sql.Date.valueOf("1582-10-10");
java.sql.PreparedStatement ps =
  c.prepareStatement("Insert into DATETABLE values(?)");
ps.setDate(1, d);
ps.executeUpdate(); // Value is inserted as 1582-10-20
```

To retrieve a value in the range October 5, 1582, to October 14, 1582, from a DB2 table without date adjustment, execute the SQL CHAR function against the DATE or TIMESTAMP column in the SELECT statement that defines a ResultSet. Executing the CHAR function converts the date or time value to a character string value on the database side.

To store a value in the range October 5, 1582, to October 14, 1582 in a DB2 table without date adjustment, you can use one of the following techniques:

- For a JDBC or an SQLJ application, use the setString method to assign the value to a String input parameter. Cast the input parameter as VARCHAR, and execute the DATE or TIMESTAMP function against the result of the cast. Then store the result of the DATE or TIMESTAMP function in the DATE or TIMESTAMP column.
- For a JDBC application, set the Connection or DataSource property sendDataAsIs to true, and use the setString method to assign the date or
timestamp value to the input parameter. Then execute an SQL statement to assign the String value to the DATE or TIMESTAMP column.

Example: Retrieve October 10, 1582, from a DATE column without date adjustment.

```java
// DATETABLE has one date column called DATECOL with one row.
// Its value is 1582-10-10.
java.sql.ResultSet rs =
    statement.executeQuery(SELECT CHAR(DATECOL) FROM DATETABLE);
rs.next();
System.out.println(rs.getString(1)); // Value is retrieved as 1582-10-10
```

Example: Store October 10, 1582, in a DATE column without date adjustment.

```java
String s = "1582-10-10";
java.sql.Statement stmt = c.createStatement;
java.sql.PreparedStatement ps =
    c.prepareStatement("Insert INTO DATETABLE VALUES " +
    "(" + DATE(CAST (? AS VARCHAR)) + ");
ps.setString(1, s);
ps.executeUpdate(); // Value is inserted as 1582-10-10
```

---

**Properties for the IBM Data Server Driver for JDBC and SQLJ**

IBM Data Server Driver for JDBC and SQLJ properties define how the connection to a particular data source should be made. Most properties can be set for a DataSource object or for a Connection object.

**Methods for setting the properties**

Properties can be set in one of the following ways:

- Using setXXX methods, where XXX is the unqualified property name, with the first character capitalized.
  Properties are applicable to the following IBM Data Server Driver for JDBC and SQLJ-specific implementations that inherit from com.ibm.db2.jcc.DB2BaseDataSource:
  - com.ibm.db2.jcc.DB2SimpleDataSource
  - com.ibm.db2.jcc.DB2ConnectionPoolDataSource
  - com.ibm.db2.jcc.DB2XADataSource

- In a java.util.Properties value in the `info` parameter of a DriverManager.getConnection call.

- In a java.lang.String value in the `url` parameter of a DriverManager.getConnection call.

Some properties with an int data type have predefined constant field values. You must resolve constant field values to their integer values before you can use those values in the `url` parameter. For example, you cannot use `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL` in a `url` parameter. However, you can build a URL string that includes `com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL`, and assign the URL string to a String variable. Then you can use the String variable in the `url` parameter:

```java
String url =
    "jdbc:db2://sysmvs1.stl.ibm.com:5021" +
    "user=dbadm;password=dbadm;" +
    "traceLevel=" +
    (com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL) + ";";
Connection con =
    java.sql.DriverManager.getConnection(url);
```
Common IBM Data Server Driver for JDBC and SQLJ properties for all supported database products

Most of the IBM Data Server Driver for JDBC and SQLJ properties apply to all database products that the driver supports.

Unless otherwise noted, all properties are in com.ibm.db2.jcc.DB2BaseDataSource.

Those properties are:

**affinityFailbackInterval**

Specifies the length of the interval, in seconds, that the IBM Data Server Driver for JDBC and SQLJ waits between attempts to fail back an existing connection to the primary server. A value that is less than or equal to 0 means that the connection does not fail back. The default is `DB2BaseDataSource.NOT_SET (0)`.

Attempts to fail back connections to the primary server are made at transaction boundaries, after the specified interval elapses.

`affinityFailbackInterval` is used only if the values of properties `enableSeamlessFailover` and `enableClientAffinitiesList` are `DB2BaseDataSource.YES`.

`affinityFailbackInterval` applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**allowNextOnExhaustedResultSet**

Specifies how the IBM Data Server Driver for JDBC and SQLJ handles a `ResultSet.next()` call for a forward-only cursor that is positioned after the last row of the `ResultSet`. The data type of this property is int.

Possible values are:

- **DB2BaseDataSource.YES (1)**
  - For a `ResultSet` that is defined as `TYPE_FORWARD_ONLY`, `ResultSet.next()` returns `false` if the cursor was previously positioned after the last row of the `ResultSet`. `false` is returned, regardless of whether the cursor is open or closed.

- **DB2BaseDataSource.NO (2)**
  - For a `ResultSet` that is defined as `TYPE_FORWARD_ONLY`, when `ResultSet.next()` is called, and the cursor was previously positioned after the last row of the `ResultSet`, the driver throws a `java.sql.SQLException` with error text "Invalid operation: result set is closed." This is the default.

**allowNullResultSetForExecuteQuery**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ returns null when `Statement.executeQuery`, `PreparedStatement.executeQuery`, or `CallableStatement.executeQuery` is used to execute a CALL statement for a stored procedure that does not return any result sets.

Possible values are:

- **DB2BaseDataSource.NOT_SET (0)**
  - The behavior is the same as for `DB2BaseDataSource.NO`.

- **DB2BaseDataSource.YES (1)**
  - The IBM Data Server Driver for JDBC and SQLJ returns null when `Statement.executeQuery`, `PreparedStatement.executeQuery`, or `CallableStatement.executeQuery` is used to execute a CALL statement
for a stored procedure that does not return any result sets. This behavior does not conform to the JDBC standard.

**DB2BaseDataSource.NO (2)**

The IBM Data Server Driver for JDBC and SQLJ throws a SQLException when Statement.executeQuery, PreparedStatement.executeQuery, or CallableStatement.executeQuery is used to execute a CALL statement for a stored procedure that does not return any result sets. This behavior conforms to the JDBC standard.

**atomicMultiRowInsert**

Specifies whether batch operations that use PreparedStatement methods to modify a table are atomic or non-atomic. The data type of this property is int.

For connections to DB2 for z/OS, this property applies only to batch INSERT operations.

For connections to DB2 Database for Linux, UNIX, and Windows or IBM Informix, this property applies to batch INSERT, MERGE, UPDATE or DELETE operations.

Possible values are:

**DB2BaseDataSource.YES (1)**

Batch operations are atomic. Insertion of all rows in the batch is considered to be a single operation. If insertion of a single row fails, the entire operation fails with a BatchUpdateException. Use of a batch statement that returns auto-generated keys fails with a BatchUpdateException.

If atomicMultiRowInsert is set to DB2BaseDataSource.YES (1):
- Execution of statements in a heterogeneous batch is not allowed.
- If the target data source is DB2 for z/OS the following operations are not allowed:
  - Insertion of more than 32767 rows in a batch results in a BatchUpdateException.
  - Calling more than one of the following methods against the same parameter in different rows results in a BatchUpdateException:
    - PreparedStatement.setAsciiStream
    - PreparedStatement.setCharacterStream
    - PreparedStatement.setUnicodeStream

**DB2BaseDataSource.NO (2)**

Batch inserts are non-atomic. Insertion of each row is considered to be a separate execution. Information on the success of each insert operation is provided by the int[] array that is returned by Statement.executeBatch.

**DB2BaseDataSource.NOT_SET (0)**

Batch inserts are non-atomic. Insertion of each row is considered to be a separate execution. Information on the success of each insert operation is provided by the int[] array that is returned by Statement.executeBatch. This is the default.

**blockingReadConnectionTimeout**

The amount of time in seconds before a connection socket read times out. This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and affects all requests that are sent to the data source after a connection is successfully established. The default is 0. A value of 0 means that there is no timeout.
**clientDebugInfo**
Specifies a value for the CLIENT DEBUGINFO connection attribute, to notify the data server that stored procedures and user-defined functions that are using the connection are running in debug mode. CLIENT DEBUGINFO is used by the DB2 Unified Debugger. The data type of this property is String. The maximum length is 254 bytes.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**clientRerouteAlternateServerName**
Specifies one or more server names for client reroute. The data type of this property is String.

When enableClientAffinitiesList=DB2BaseDataSource.YES (1), clientRerouteAlternateServerName must contain the name of the primary server as well as alternate server names. The server that is identified by serverName and portNumber is the primary server. That server name must appear at the beginning of the clientRerouteAlternateServerName list.

If more than one server name is specified, delimit the server names with commas (,) or spaces. The number of values that is specified for clientRerouteAlternateServerName must match the number of values that is specified for clientRerouteAlternatePortNumber.

clientRerouteAlternateServerName applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**clientRerouteAlternatePortNumber**
Specifies one or more port numbers for client reroute. The data type of this property is String.

When enableClientAffinitiesList=DB2BaseDataSource.YES (1), clientRerouteAlternatePortNumber must contain the port number for the primary server as well as port numbers for alternate servers. The server that is identified by serverName and portNumber is the primary server. That port number must appear at the beginning of the clientRerouteAlternatePortNumber list.

If more than one port number is specified, delimit the port numbers with commas (,) or spaces. The number of values that is specified for clientRerouteAlternatePortNumber must match the number of values that is specified for clientRerouteAlternateServerName.

clientRerouteAlternatePortNumber applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**clientRerouteServerListJNDIName**
Identifies a JNDI reference to a DB2ClientRerouteServerList instance in a JNDI repository of reroute server information. clientRerouteServerListJNDIName applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, and to connections that are established through the DataSource interface.

If the value of clientRerouteServerListJNDIName is not null, clientRerouteServerListJNDIName provides the following functions:

- Allows information about reroute servers to persist across JVMs
- Provides an alternate server location if the first connection to the data source fails
clientRerouteServerListJNDIContext
Specifies the JNDI context that is used for binding and lookup of the
DB2ClientRerouteServerList instance. clientRerouteServerListJNDIContext
applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity,
and to connections that are established through the DataSource interface.

If clientRerouteServerListJNDIContext is not set, the IBM Data Server Driver
for JDBC and SQLJ creates an initial context using system properties or the
jndi.properties file.

clientRerouteServerListJNDIContext can be set **only** by using the following
method:
```
public void setClientRerouteServerListJNDIContext(javax.naming.Context registry)
```

connectionCloseWithInFlightTransaction
Specifies whether the IBM Data Server Driver for JDBC and SQLJ throws an
SQLException or rolls back a transaction without throwing an SQLException
when a connection is closed in the middle of the transaction. Possible values
are:

**DB2BaseDataSource.NOT_SET (0)**
The behavior is the same as for
DB2BaseDataSource.CONNECTION_CLOSE_WITH_EXCEPTION.

**DB2BaseDataSource.CONNECTION_CLOSE_WITH_EXCEPTION (1)**
When a connection is closed in the middle of a transaction, an
SQLException with error -4471 is thrown.

**DB2BaseDataSource.CONNECTION_CLOSE_WITH_ROLLBACK (2)**
When a connection is closed in the middle of a transaction, the
transaction is rolled back, and no SQLException is thrown.

databaseName
Specifies the name for the data source. This name is used as the database
portion of the connection URL. The name depends on whether IBM Data
Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server
Driver for JDBC and SQLJ type 2 connectivity is used.

For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity:
- If the connection is to a DB2 for z/OS server, the databaseName value is the
  DB2 location name that is defined during installation. All characters in this
  value must be uppercase characters. You can determine the location name by
  executing the following SQL statement on the server:
    ```sql
    SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
    ```
- If the connection is to a DB2 Database for Linux, UNIX, and Windows
  server, the databaseName value is the database name that is defined during
  installation.
- If the connection is to an IDS server, database is the database name. The
  name is case-insensitive. The server converts the name to lowercase.
- If the connection is to an IBM Cloudscape server, the databaseName value is
  the fully-qualified name of the file that contains the database. This name
  must be enclosed in double quotation marks ("). For example:
  ```
  "c:/databases/testdb"
  ```

If this property is not set, connections are made to the local site.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity:
• The databaseName value is the database name that is defined during installation, if the value of the serverName connection property is null. If the value of serverName property is not null, the databaseName value is a database alias.

decimalSeparator

Specifies the decimal separator for input and output, for decimal, floating point, or decimal floating-point data values. The data type of this property is int.

If the value of the sendDataAsIs property is true, decimalSeparator affects only output values.

Possible values are:

- **DB2BaseDataSource.DECIMAL_SEPARATOR_NOT_SET (0)**
  A period is used as the decimal separator. This is the default.

- **DB2BaseDataSource.DECIMAL_SEPARATOR_PERIOD (1)**
  A period is used as the decimal separator.

- **DB2BaseDataSource.DECIMAL_SEPARATOR_COMMA (2)**
  A comma is used as the decimal separator.

  When DECIMAL_SEPARATOR_COMMA is set, the result of ResultSet.getString on a decimal, floating point, or decimal floating-point value has a comma as a separator. However, if the toString method is executed on a value that is retrieved with a ResultSet.getXXX method that returns a decimal, floating point, or decimal floating-point value, the result has a decimal point as the decimal separator.

decimalStringFormat

Specifies the string format for data that is retrieved from a DECIMAL or DECFLOAT column when the SDK for Java is Version 1.5 or later. The data type of this property is int. Possible values are:

- **DB2BaseDataSource.DECIMAL_STRING_FORMAT_NOT_SET (0)**
  The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toString method returns them. This is the default.

  For example, the value 0.0000000004 is returned as 4E-10.

- **DB2BaseDataSource.DECIMAL_STRING_FORMAT_TO_STRING (1)**
  The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toString method returns them.

  For example, the value 0.0000000004 is returned as 4E-10.

- **DB2BaseDataSource.DECIMAL_STRING_FORMAT_TO_PLAIN_STRING (2)**
  The IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toPlainString method returns them.

  For example, the value 0.0000000004 is returned as 0.0000000004.

This property has no effect for earlier versions of the SDK for Java. For those versions, the IBM Data Server Driver for JDBC and SQLJ returns decimal values in the format that the java.math.BigDecimal.toString method returns them.
**defaultIsolationLevel**

Specifies the default transaction isolation level for new connections. The data type of this property is int. When defaultIsolationLevel is set on a DataSource, all connections that are created from that DataSource have the default isolation level that is specified by defaultIsolationLevel.

For DB2 data sources, the default is `java.sql.Connection.TRANSACTION_READ_COMMITTED`.

For IBM Informix (IDS) databases, the default depends on the type of data source. The following table shows the defaults.

<table>
<thead>
<tr>
<th>Type of data source</th>
<th>Default isolation level</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI-compliant database with logging</td>
<td><code>java.sql.Connection.TRANSACTION_SERIALIZABLE</code></td>
</tr>
<tr>
<td>Database without logging</td>
<td><code>java.sql.Connection.TRANSACTION_READ_UNCOMMITTED</code></td>
</tr>
<tr>
<td>Non-ANSI-compliant database with logging</td>
<td><code>java.sql.Connection.TRANSACTION_READ_COMMITTED</code></td>
</tr>
</tbody>
</table>

**deferPrepares**

Specifies whether invocation of the Connection.prepareStatement method results in immediate preparation of an SQL statement on the data source, or whether statement preparation is deferred until the PreparedStatement.executeUpdate method is executed. The data type of this property is boolean.

deferPrepares is supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, and for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Possible values are:

- **true**: Statement preparation on the data source does not occur until the PreparedStatement.executeUpdate method is executed. This is the default.
- **false**: Statement preparation on the data source occurs when the Connection.prepareStatement method is executed.

Deferring prepare operations can reduce network delays. However, if you defer prepare operations, you need to ensure that input data types match table column types.

**description**

A description of the data source. The data type of this property is String.

**downgradeHoldCursorsUnderXa**

Specifies whether cursors that are defined WITH HOLD can be opened under XA connections.

downgradeHoldCursorsUnderXa applies to:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS servers.
- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows servers.

The default is `false`, which means that a cursor that is defined WITH HOLD cannot be opened under an XA connection. An exception is thrown when an attempt is made to open that cursor.
If `downgradeHoldCursorsUnderXa` is set to `true`, a cursor that is defined
WITH HOLD can be opened under an XA connection. However, the cursor has
the following restrictions:

- When the cursor is opened under an XA connection, the cursor does not
  have WITH HOLD behavior. The cursor is closed at XA End.
- A cursor that is open before XA Start on a local transaction is closed at XA
  Start.

**driverType**
For the `DataSource` interface, determines which driver to use for connections.
The data type of this property is `int`. Valid values are 2 or 4. 2 is the default.

**enableClientAffinitiesList**
Specifies whether the IBM Data Server Driver for JDBC and SQLJ enables
client affinities for cascaded failover support. The data type of this property is
int. Possible values are:

- **DB2BaseDataSource.YES (1)**
  The IBM Data Server Driver for JDBC and SQLJ enables client affinities
  for cascaded failover support. This means that only servers that are
  specified in the `clientRerouteAlternateServerName` and `clientRerouteAlternatePortNumber`
  properties are retried. The driver does not attempt to reconnect to any other servers.

  For example, suppose that `clientRerouteAlternateServerName` contains
  the following string:
  
  host1,host2,host3

  Also suppose that `clientRerouteAlternatePortNumber` contains the
  following string:
  
  port1,port2,port3

  When client affinities are enabled, the retry order is:
  1. host1:port1
  2. host2:port2
  3. host3:port3

- **DB2BaseDataSource.NO (2)**
  The IBM Data Server Driver for JDBC and SQLJ does not enable client
  affinities for cascaded failover support.

- **DB2BaseDataSource.NOT_SET (0)**
  The IBM Data Server Driver for JDBC and SQLJ does not enable client
  affinities for cascaded failover support. This is the default.

  The effect of the `maxRetriesForClientReroute` and `retryIntervalForClientReroute`
  properties differs depending on whether `enableClientAffinitiesList` is enabled.

  This property applies only to IBM Data Server Driver for JDBC and SQLJ type
  4 connectivity.

**enableNamedParameterMarkers**
Specifies whether support for named parameter markers is enabled in the IBM
Data Server Driver for JDBC and SQLJ. The data type of this property is `int`.
Possible values are:

- **DB2BaseDataSource.YES (1)**
  Named parameter marker support is enabled in the IBM Data Server
  Driver for JDBC and SQLJ.
Named parameter marker support is not enabled in the IBM Data Server Driver for JDBC and SQLJ.

The driver sends an SQL statement with named parameter markers to the target data source without modification. The success or failure of the statement depends on a number of factors, including the following ones:

- Whether the target data source supports named parameter markers
- Whether the deferPrepares property value is true or false
- Whether the sendDataAsIs property value is true or false

**Recommendation:** To avoid unexpected behavior in an application that uses named parameter markers, set enableNamedParameterMarkers to YES.

The behavior is the same as the behavior for DB2BaseDataSource.NO (2). This is the default.

**enableSeamlessFailover**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses seamless failover for client reroute. The data type of this property is int.

For connections to DB2 for z/OS, if enableSysplexWLB is set to true, enableSeamlessFailover has no effect. The IBM Data Server Driver for JDBC and SQLJ uses seamless failover regardless of the enableSeamlessFailover setting.

Possible values of enableSeamlessFailover are:

**DB2BaseDataSource.YES (1)**

The IBM Data Server Driver for JDBC and SQLJ uses seamless failover. This means that the driver does not throw an SQLException with error code -4498 after a failed connection has been successfully re-established if the following conditions are true:

- The connection was not being used for a transaction at the time the failure occurred.
- There are no outstanding global resources, such as global temporary tables or open, held cursors, or connection states that prevent a seamless failover to another server.

When seamless failover occurs, after the connection to a new data source has been established, the driver re-issues the SQL statement that was being processed when the original connection failed.

**Recommendation:** Set the queryCloseImplicit property to DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO (2) when you set enableSeamlessFailover to DB2BaseDataSource.YES, if the application uses held cursors.

**DB2BaseDataSource.NO (2)**

The IBM Data Server Driver for JDBC and SQLJ does not use seamless failover.

When this setting is in effect, if a server goes down, the driver tries to fail back or fail over to an alternate server. If failover or failback is successful, the driver throws an SQLException with error code -4498, which indicates that a connection failed but was successfully reestablished. An SQLException with error code -4498 informs the
application that it should retry the transaction during which the connection failure occurred. If the driver cannot reestablish a connection, it throws an SQLException with error code -4499.

DB2BaseDataSource.NOT_SET (0)
The IBM Data Server Driver for JDBC and SQLJ does not use seamless failover. This is the default.

fetchSize
Specifies the default fetch size for ResultSet objects that are generated from Statement objects. The data type of this property is int.

The fetchSize default can be overridden by the Statement.setFetchSize method. The fetchSize property does not affect Statement objects that already exist when fetchSize is set.

Possible values of fetchSize are:

0 or positive-integer
The default fetchSize value for newly created Statement objects. If the fetchSize property value is invalid, the IBM Data Server Driver for JDBC and SQLJ sets the default fetchSize value to 0.

DB2BaseDataSource.FETCHSIZE_NOT_SET (-1)
Indicates that the default fetchSize value for Statement objects is 0. This is the property default.

The fetchSize property differs from the queryDataSize property. fetchSize affects the number of rows that are returned, and queryDataSize affects the number of bytes that are returned.

floatingPointStringFormat
Specifies the format for data that is retrieved from a DOUBLE, FLOAT, or REAL column with the ResultSet.getString method. The data type of this property is int. Possible values are:

DB2BaseDataSource.NOT_SET (0)
The IBM Data Server Driver for JDBC and SQLJ returns double-precision floating point values in the string format that the java.lang.String.valueOf(double) method returns them. The IBM Data Server Driver for JDBC and SQLJ returns single-precision floating point values in the string format that the java.lang.String.valueOf(float) method returns them. This is the default.

For example, suppose that the value 71256.789 is retrieved from a DOUBLE column. If floatingPointStringFormat is not set, the string format of the retrieved value is 71256.789. If the value 71256.789 is retrieved from a REAL column, the string format of the retrieved value is 71256.79.

DB2BaseDataSource.JCC_DRIVER_FLOATING_POINT_STRING_FORMAT (1)
The IBM Data Server Driver for JDBC and SQLJ returns double-precision floating point values in the string format that the java.lang.String.valueOf(double) method returns them. The IBM Data Server Driver for JDBC and SQLJ returns single-precision floating point values in the string format that the java.lang.String.valueOf(float) method returns them. This is the default.

For example, suppose that the value 71256.789 is retrieved from a DOUBLE column. If floatingPointStringFormat is DB2BaseDataSource.JCC_DRIVER_FLOATING_POINT_STRING_FORMAT, the
string format of the retrieved value is 71256.789. If the value 71256.789 is retrieved from a REAL column, the string format of the retrieved value is 71256.79.

**DB2BaseDataSource.LUW_TYPE2_DRIVER_FLOATING_POINT_STRING_FORMAT (2)**
The IBM Data Server Driver for JDBC and SQLJ returns DOUBLE, FLOAT, or REAL values in the same format that the DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows returns.

For example, suppose that the value 71256.789 is retrieved from a DOUBLE column. If floatingPointStringFormat is `DB2BaseDataSource.LUW_TYPE2_DRIVER_FLOATING_POINT_STRING_FORMAT`, the string format of the retrieved value is 7.12567890000000E+004. If the value 71256.789 is retrieved from a REAL column, the string format of the retrieved value is 7.125679E+04.

**fullyMaterializeLobData**
Indicates whether the driver retrieves LOB locators for FETCH operations. The data type of this property is boolean.

The effect of fullyMaterializeLobData depends on whether the data source supports progressive streaming, which is also known as dynamic data format:

- If the data source does not support progressive streaming:
  - If the value of fullyMaterializeLobData is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If the value is `false`, LOB data is streamed. The driver uses locators internally to retrieve LOB data in chunks on an as-needed basis. It is highly recommended that you set this value to `false` when you retrieve LOBs that contain large amounts of data. The default is `true`.

- If the data source supports progressive streaming:
  The JDBC driver ignores the value of fullyMaterializeLobData if the progressiveStreaming property is set to `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`.

This property has no effect on stored procedure parameters or on LOBs that are fetched using scrollable cursors. LOB stored procedure parameters are always fully materialized. LOBs that are fetched using scrollable cursors use LOB locators if progressive streaming is not in effect.

**interruptProcessingMode**
Specifies the behavior of the IBM Data Server Driver for JDBC and SQLJ when an application executes the Statement.cancel method. Possible values are:

- **DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_DISABLED (0)**
  Interrupt processing is disabled. When an application executes Statement.cancel, the IBM Data Server Driver for JDBC and SQLJ does nothing.

- **DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL (1)**
  When an application executes Statement.cancel, the IBM Data Server Driver for JDBC and SQLJ cancels the currently executing statement, if the data server supports interrupt processing. If the data server does not support interrupt processing, the IBM Data Server Driver for JDBC and SQLJ throws an SQLException that indicates that the feature is not supported.

    INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL is the default.
For DB2 Database for Linux, UNIX, and Windows clients, when interruptProcessingMode is set to INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL, the DB2 Connect setting for INTERRUPT_ENABLED and the DB2 registry variable setting for DB2CONNECT_DISCONNECT_ON_INTERRUPT override this value.

**DB2BaseDataSource.INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET (2)**

When an application executes Statement.cancel, the IBM Data Server Driver for JDBC and SQLJ performs one of the following actions:

- If automatic client reroute or client affinities is not enabled, the IBM Data Server Driver for JDBC and SQLJ drops the underlying socket, closes the connection, and throws an SQLException that indicates that the application is being disconnected from the data server. Any subsequent operations that are invoked on any Statement objects that are created from the same connection receive an SQLException that indicates that the connection is closed.

- If automatic client reroute or client affinities is enabled, the IBM Data Server Driver for JDBC and SQLJ drops the underlying socket, closes the connection, and then attempts to re-establish the connection. If re-connection is successful, the driver throws an SQLException that indicates that the connection was re-established. the driver does not re-execute any SQL statements, even if the enableSeamlessFailover property is set to DB2BaseDataSource.YES.

**loginTimeout**

The maximum time in seconds to wait for a connection to a data source. After the number of seconds that are specified by loginTimeout have elapsed, the driver closes the connection to the data source. The data type of this property is int. The default is 0. A value of 0 means that the timeout value is the default system timeout value. This property is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

**logWriter**

The character output stream to which all logging and trace messages for the DataSource object are printed. The data type of this property is java.io.PrintWriter. The default value is null, which means that no logging or tracing for the DataSource is output.

**maxRetriesForClientReroute**

During automatic client reroute, limit the number of retries if the primary connection to the data source fails.

The data type of this property is int.

The IBM Data Server Driver for JDBC and SQLJ uses the maxRetriesForClientReroute property only if the retryIntervalForClientReroute property is also set.

If the enableClientAffinitiesList is set to DB2BaseDataSource.NO (2), an attempt to connect to the primary server and alternate servers counts as one retry. If enableClientAffinitiesList is set to DB2BaseDataSource.YES (1), each server that is specified by the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber values is retried the number of times that is specified by maxRetriesForClientReroute.

The default value for maxRetriesForClientReroute is 0 if enableClientAffinitiesList is DB2BaseDataSource.NO (2), or 3 if enableClientAffinitiesList is DB2BaseDataSource.YES (1).
If the value of maxRetriesForClientReroute is 0, client reroute processing does not occur.

**password**

The password to use for establishing connections. The data type of this property is String. When you use the DataSource interface to establish a connection, you can override this property value by invoking this form of the DataSource.getConnection method:

```java
getConnection(user, password);
```

**portNumber**

The port number where the DRDA server is listening for requests. The data type of this property is int.

**progressiveStreaming**

Specifies whether the JDBC driver uses progressive streaming when progressive streaming is supported on the data source.

DB2 for z/OS Version 9.1 and later supports progressive streaming for LOBs and XML objects. DB2 Database for Linux, UNIX, and Windows Version 9.5 and later, and IBM Informix (IDS) Version 11.50 and later support progressive streaming for LOBs.

With progressive streaming, also known as dynamic data format, the data source dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects. The value of the streamBufferSize parameter determines whether the data is materialized when it is returned.

The data type of progressiveStreaming is int. Valid values are `DB2BaseDataSource.YES` (1) and `DB2BaseDataSource.NO` (2). If the progressiveStreaming property is not specified, the progressiveStreaming value is `DB2BaseDataSource.NOT_SET` (0).

If the connection is to a data source that supports progressive streaming, and the value of progressiveStreaming is `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`, the JDBC driver uses progressive streaming to return LOBs and XML data.

If the value of progressiveStreaming is `DB2BaseDataSource.NO`, or the data source does not support progressive streaming, the way in which the JDBC driver returns LOB or XML data depends on the value of the `fullyMaterializeLobData` property.

**queryCloseImplicit**

Specifies whether cursors are closed immediately after all rows are fetched. `queryCloseImplicit` applies only to connections to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS Version 8 or later, and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity or IBM Data Server Driver for JDBC and SQLJ type 2 connectivityDB2 Database for Linux, UNIX, and Windows Version 9.7 or later. Possible values are:

```java
DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES (1)
```

Close cursors immediately after all rows are fetched.

A value of `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES` can provide better performance because this setting results in less network traffic.

```java
DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO (2)
```

Do not close cursors immediately after all rows are fetched.
**DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_COMMIT (3)**

Perform these actions:
- Implicitly close the cursor after all rows are fetched.
- If the application is in autocommit mode, implicitly send a commit request to the data source for the current unit of work.

**Important:** When this value is set, there might be impacts on other resources, just as an explicit commit operation might impact other resources. For example, other non-held cursors are closed, LOB locators go out of scope, progressive references are reset, and scrollable cursors lose their position.

**Restriction:** The following restrictions apply to QUERY_CLOSE_IMPLICIT_COMMIT behavior:
- This behavior applies only to SELECT statements that are issued by the application. It does not apply to SELECT statements that are generated by the IBM Data Server Driver for JDBC and SQLJ.
- If QUERY_CLOSE_IMPLICIT_COMMIT is set, and the application is not in autocommit mode, the driver uses the default behavior (QUERY_CLOSE_IMPLICIT_NOT_SET behavior). If QUERY_CLOSE_IMPLICIT_COMMIT is the default behavior, the driver uses QUERY_CLOSE_IMPLICIT_YES behavior.
- If QUERY_CLOSE_IMPLICIT_COMMIT is set, and the data source does not support QUERY_CLOSE_IMPLICIT_COMMIT behavior, the driver uses QUERY_CLOSE_IMPLICIT_YES behavior.
- This behavior is not supported for batched statements.
- This behavior is supported on an XA Connection only when the connection is in a local transaction.

**DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NOT_SET (0)**

This is the default. The following table describes the behavior for a connection to each type of data source.

<table>
<thead>
<tr>
<th>Data source</th>
<th>Version</th>
<th>Data sharing environment</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 with APAR PK68746</td>
<td>Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_COMMIT</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 without APAR PK68746</td>
<td>Non-data sharing, or in a data sharing group but not in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_YES</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 with APAR PK68746</td>
<td>In a data sharing group in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_COMMIT</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 9 without APAR PK68746</td>
<td>In a data sharing group in coexistence mode with Version 8 members</td>
<td>QUERY_CLOSE_IMPLICIT_YES</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 8 with or without APAR PK68746</td>
<td></td>
<td>QUERY_CLOSE_IMPLICIT_YES</td>
</tr>
<tr>
<td>DB2 Database for Linux, UNIX, and Windows</td>
<td>Version 9.7</td>
<td></td>
<td>QUERY_CLOSE_IMPLICIT_YES</td>
</tr>
</tbody>
</table>
queryDataSize

Specifies a hint that is used to control the amount of query data, in bytes, that is returned from the data source on each fetch operation. This value can be used to optimize the application by controlling the number of trips to the data source that are required to retrieve data.

Use of a larger value for queryDataSize can result in less network traffic, which can result in better performance. For example, if the result set size is 50 KB, and the value of queryDataSize is 32767 (32KB), two trips to the database server are required to retrieve the result set. However, if queryDataSize is set to 65535 (64 KB), only one trip to the data source is required to retrieve the result set.

The following table lists minimum, maximum, and default values of queryDataSize for each data source.

Table 53. Default, minimum, and maximum values of queryDataSize

<table>
<thead>
<tr>
<th>Data source</th>
<th>Product Version</th>
<th>Default</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Valid values</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2 Database for Linux, UNIX, and Windows</td>
<td>All</td>
<td>32767</td>
<td>4096</td>
<td>262143</td>
<td>4096-32767, 98303, 131071, 163839, 196607, 229375, 262143</td>
</tr>
<tr>
<td>IDS</td>
<td>All</td>
<td>32767</td>
<td>4096</td>
<td>10485760</td>
<td>4096-10485760</td>
</tr>
<tr>
<td>DB2 for i</td>
<td>V5R4</td>
<td>32767</td>
<td>4096</td>
<td>65535</td>
<td>4096-65535</td>
</tr>
<tr>
<td></td>
<td>V6R1</td>
<td>32767</td>
<td>4096</td>
<td>262143</td>
<td>4096-65535, 98303, 131071, 163839, 196607, 229375, 262143</td>
</tr>
<tr>
<td>DB2 for z/OS</td>
<td>Version 8</td>
<td>32767</td>
<td>32767</td>
<td>32767</td>
<td>32767</td>
</tr>
<tr>
<td></td>
<td>Version 9</td>
<td>32767</td>
<td>32767</td>
<td>65535</td>
<td>32767, 65535</td>
</tr>
</tbody>
</table>

Note:
1. If you specify a value between the minimum and maximum value that is not a valid value, the IBM Data Server Driver for JDBC and SQLJ sets queryDataSize to the nearest valid value.

resultSetHoldability

Specifies whether cursors remain open after a commit operation. The data type of this property is int. Valid values are:

DB2BaseDataSource.HOLD_CURSORS_OVER_COMMIT (1)

Leave cursors open after a commit operation.

This setting is not valid for a connection that is part of a distributed (XA) transaction.

DB2BaseDataSource.CLOSE_CURSORS_AT_COMMIT (2)

Close cursors after a commit operation.

DB2BaseDataSource.NOT_SET (0)

This is the default value. The behavior is:

- For connections that are part of distributed (XA) transactions, cursors are closed after a commit operation.
- For connections that are not part of a distributed transaction:
  - For connections to all versions of DB2 for z/OS, DB2 Database for Linux, UNIX, and Windows, or DB2 for i servers, or to Cloudscape Version 8.1 or later servers, cursors remain open after a commit operation.
For connections to all versions of IBM Informix, or to Cloudscape versions earlier than Version 8.1, cursors are closed after a commit operation.

**retrieveMessagesFromServerOnGetMessage**

Specifies whether JDBC SQLException.getMessage or SQLWarning.getMessage calls cause the IBM Data Server Driver for JDBC and SQLJ to invoke a DB2 for z/OS stored procedure that retrieves the message text for the error. The data type of this property is boolean. The default is false, which means that the full message text is not returned to the client.

For example, if retrieveMessagesFromServerOnGetMessage is set to true, a message similar to this one is returned by SQLException.getMessage after an attempt to perform an SQL operation on nonexistent table ADMF001.NO_TABLE:

```
ADMF001.NO_TABLE IS AN UNDEFINED NAME. SQLCODE=-204,
SQLSTATE=42704, DRIVER=3.50.54
```

If retrieveMessagesFromServerOnGetMessage is set to false, a message similar to this one is returned:

```
DB2 SQL Error: SQLCODE=-204, SQLSTATE=42704, DRIVER=3.50.54
```

An alternative to setting this property to true is to use the IBM Data Server Driver for JDBC and SQLJ-only DB2Sqlca.getMessage method in applications. Both techniques result in a stored procedure call, which starts a unit of work.

**retryIntervalForClientReroute**

For automatic client reroute, specifies the amount of time in seconds between connection retries.

The data type of this property is int.

The IBM Data Server Driver for JDBC and SQLJ uses the retryIntervalForClientReroute property only if the maxRetriesForClientReroute property is also set.

If maxRetriesForClientReroute or retryIntervalForClientReroute is not set, the IBM Data Server Driver for JDBC and SQLJ performs retries for 10 minutes.

If the enableClientAffinitiesList is set to DB2BaseDataSource.NO (2), an attempt to connect to the primary server and alternate servers counts as one retry. The driver waits the number of seconds that is specified by retryIntervalForClientReroute before retrying the connection. If enableClientAffinitiesList is set to DB2BaseDataSource.YES (1), each server that is specified by the clientRerouteAlternateServerName and clientRerouteAlternatePortNumber values is retried after the number of seconds that is specified by retryIntervalForClientReroute.

The default value for retryIntervalForClientReroute is DB2BaseDataSource.NOT_SET (-1). The default behavior is that there is no wait between retries.

**securityMechanism**

Specifies the DRDA security mechanism. The data type of this property is int.

Possible values are:

- **CLEAR_TEXT_PASSWORD_SECURITY (3)**
  - User ID and password

- **USER_ONLY_SECURITY (4)**
  - User ID only
ENCRYPTED PASSWORD SECURITY (7)
User ID, encrypted password

ENCRYPTED USER AND PASSWORD SECURITY (9)
Encrypted user ID and password

Kerberos. This value does not apply to connections to IDS.

ENCRYPTED USER AND DATA SECURITY (12)
Encrypted user ID and encrypted security-sensitive data. This value applies to connections to DB2 for z/OS only.

ENCRYPTED USER PASSWORD AND DATA SECURITY (13)
Encrypted user ID and password, and encrypted security-sensitive data. This value does not apply to connections to IDS.

PLUGIN SECURITY (15)
Plug-in security. This value applies to connections to DB2 Database for Linux, UNIX, and Windows only.

ENCRYPTED USER ONLY SECURITY (16)
Encrypted user ID. This value does not apply to connections to IDS.

If this property is specified, the specified security mechanism is the only mechanism that is used. If the security mechanism is not supported by the connection, an exception is thrown.

The default value for securityMechanism is CLEAR_TEXT_PASSWORD_SECURITY. If the server does not support CLEAR_TEXT_PASSWORD_SECURITY but supports ENCRYPTED_USER_AND_PASSWORD_SECURITY, the IBM Data Server Driver for JDBC and SQLJ driver updates the security mechanism to ENCRYPTED_USER_AND_PASSWORD_SECURITY and attempts to connect to the server. Any other mismatch in security mechanism support between the requester and the server results in an error.

sendDataAsIs
Specifies that the IBM Data Server Driver for JDBC and SQLJ does not convert input parameter values to the target column data types. The data type of this property is boolean. The default is false.

You should use this property only for applications that always ensure that the data types in the application match the data types in the corresponding database tables.

serverName
The host name or the TCP/IP address of the data source. The data type of this property is String.

sslConnection
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses an SSL socket to connect to the data source. If sslConnection is set to true, the connection uses an SSL socket. If sslConnection is set to false, the connection uses a plain socket.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

sslTrustStoreLocation
Specifies the name of the Java truststore on the client that contains the server certificate for an SSL connection.
The IBM Data Server Driver for JDBC and SQLJ uses this option only if the sslConnection property is set to true.

If sslTrustStore is set, and sslConnection is set to true, the IBM Data Server Driver for JDBC and SQLJ uses the sslTrustStoreLocation value instead of the value in the javax.net.ssl.trustStore Java property.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**sslTrustStorePassword**

Specifies the password for the Java truststore on the client that contains the server certificate for an SSL connection.

The IBM Data Server Driver for JDBC and SQLJ uses this option only if the sslConnection property is set to true.

If sslTrustStorePassword is set, and sslConnection is set to true, the IBM Data Server Driver for JDBC and SQLJ uses the sslTrustStorePassword value instead of the value in the javax.net.ssl.trustStorePassword Java property.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**stripTrailingZerosForDecimalNumbers**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ removes trailing zeroes when it retrieves data from a DECFLOAT, DECIMAL, or NUMERIC column. This property is meaningful only if the SDK for Java is Version 1.5 or later. The data type of this property is int. Possible values are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer value</th>
<th>Format</th>
</tr>
</thead>
</table>
| com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0) | 0 | yyyy-mm-dd-
| | | hh.mm.ss.mmmmm |
| com.ibm.db2.jcc.DB2BaseDataSource.YES (1) | 1 | yyyy-mm-dd
| | | hh:mm:ss.mmmmm |
| com.ibm.db2.jcc.DB2BaseDataSource.NO (2) | 2 | yyyy-mm-dd-
| | | hh:mm:ss.mmmmm |

For example, when the driver retrieves the value 234.04000, it returns the value 234.04 to the application.

**timestampFormat**

Specifies the format in which the result of the ResultSet.getString or CallableStatement.getString method against a TIMESTAMP column is returned. The data type of timestampFormat is int. Possible values of timestampFormat are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer value</th>
<th>Format</th>
</tr>
</thead>
</table>
| com.ibm.db2.jcc.DB2BaseDataSource.ISO | 1 | yyyy-mm-dd-
| | | hh:mm:ss.mmmmm |
| com.ibm.db2.jcc.DB2BaseDataSource.JDBC | 5 | yyyy-mm-dd
| | | hh:mm:ss.mmmmm |

Note:

The default is com.ibm.db2.jcc.DB2BaseDataSource.JDBC.
timestampFormat affects the format of output only.

**timestampPrecisionReporting**

Specifies whether trailing zeroes are truncated in the result of a ResultSet.getString call for a TIMESTAMP value. The data type of this property is int. Possible values are:

**TIMESTAMP_JDBC_STANDARD (1)**

Trailing zeroes are truncated in the result of a ResultSet.getString call for a TIMESTAMP value. This is the default.

For example:

- A TIMESTAMP value of 2009-12-01-11.30.00.100000 is truncated to 2009-12-01-11.30.00.1 after retrieval.

**TIMESTAMP_ZERO_PADDING (2)**

Trailing zeroes are not truncated in the result of a ResultSet.getString call for a TIMESTAMP value.

**traceDirectory**

Specifies a directory into which trace information is written. The data type of this property is String. When traceDirectory is specified, trace information for multiple connections on the same DataSource is written to multiple files.

When traceDirectory is specified, a connection is traced to a file named traceFile_origin_n.

n is the nth connection for a DataSource.

origin indicates the origin of the log writer that is in use. Possible values of origin are:

- **cpds** The log writer for a DB2ConnectionPoolDataSource object.
- **driver** The log writer for a DB2Driver object.
- **global** The log writer for a DB2TraceManager object.
- **sds** The log writer for a DB2SimpleDataSource object.
- **xads** The log writer for a DB2XADatasource object.

If the traceFile property is also specified, the traceDirectory value is not used.

**traceFile**

Specifies the name of a file into which the IBM Data Server Driver for JDBC and SQLJ writes trace information. The data type of this property is String. The traceFile property is an alternative to the logWriter property for directing the output trace stream to a file.

**traceFileAppend**

Specifies whether to append to or overwrite the file that is specified by the traceFile property. The data type of this property is boolean. The default is false, which means that the file that is specified by the traceFile property is overwritten.

**traceLevel**

Specifies what to trace. The data type of this property is int.

You can specify one or more of the following traces with the traceLevel property:

- `com.ibm.db2.jcc.DB2BaseDataSource.TRACENONE (X'00')`
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity for DB2 Database for Linux, UNIX, and Windows only) (X'800')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR (X'20000')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS (X'40000')
com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:

- Use bitwise OR (|) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:
  ```
  TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
  ```
- Use a bitwise complement (~) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:
  ```
  ~TRACE_DRDA_FLOWS
  ```

**user**

The user ID to use for establishing connections. The data type of this property is String. When you use the DataSource interface to establish a connection, you can override this property value by invoking this form of the DataSource.getConnection method:

```java
getConnection(user, password);
```

**xaNetworkOptimization**

Specifies whether XA network optimization is enabled for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity. You might need to disable XA network optimization in an environment in which an XA Start and XA End are issued from one Java process, and an XA Prepare and an XA Commit are issued from another Java process. With XA network optimization, the XA Prepare can reach the data source before the XA End, which results in an XAER_PROTO error. To prevent the XAER_PROTO error, disable XA network optimization.

The default is true, which means that XA network optimization is enabled. If xaNetworkOptimization is false, which means that XA network optimization is disabled, the driver closes any open cursors at XA End time.

xaNetworkOptimization can be set on a DataSource object, or in the url parameter in a getConnection call. The value of xaNetworkOptimization cannot be changed after a connection is obtained.

**com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements**

Controls an internal statement cache that is associated with a PooledConnection. The data type of this property is int. Possible values are:
positive integer
   Enables the internal statement cache for a PooledConnection, and
   specifies the number of statements that the IBM Data Server Driver for
   JDBC and SQLJ keeps open in the cache.

0 or negative integer
   Disables internal statement caching for the PooledConnection. 0 is the
   default.

maxStatements controls the internal statement cache that is associated with a
PooledConnection only when the PooledConnection object is created.
maxStatements has no effect on caching in an already existing
PooledConnection object.

maxStatements applies only to IBM Data Server Driver for JDBC and SQLJ
type 4 connectivity.

Common IBM Data Server Driver for JDBC and SQLJ
properties for DB2 servers

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to DB2
for z/OS and DB2 Database for Linux, UNIX, and Windows only.

Unless otherwise noted, all properties are in com.ibm.db2.jcc.DB2BaseDataSource.

Those properties are:

clientAccountingInformation
   Specifies accounting information for the current client for the connection. This
   information is for client accounting purposes. This value can change during a
   connection. The data type of this property is String. The maximum length is
   255 bytes. A Java empty string ("") is valid for this value, but a Java null value
   is not valid.

clientApplicationInformation
   Specifies the application or transaction name of the end user's application. You
   can use this property to provide the identity of the client end user for
   accounting and monitoring purposes. This value can change during a
   connection. The data type of this property is String. For a DB2 for z/OS server,
   the maximum length is 32 bytes. For a DB2 Database for Linux, UNIX, and
   Windows server, the maximum length is 255 bytes. A Java empty string ("") is
   valid for this value, but a Java null value is not valid.

clientProgramId
   Specifies a value for the client program ID that can be used to identify the end
   user. The data type of this property is String, and the length is 80 bytes. If the
   program ID value is less than 80 bytes, the value must be padded with blanks.

clientProgramName
   Specifies an application ID that is fixed for the duration of a physical
   connection for a client. The value of this property becomes the correlation ID
   on a DB2 for z/OS server. Database administrators can use this property to
   correlate work on a DB2 for z/OS server to client applications. The data type
   of this property is String. The maximum length is 12 bytes. If this value is
   null, the IBM Data Server Driver for JDBC and SQLJ supplies a value of
db2jccthread-name.

   This property applies only to IBM Data Server Driver for JDBC and SQLJ type
   4 connectivity.
**currentDegree**

Specifies the degree of parallelism for the execution of queries that are dynamically prepared. The type of this property is String. The currentDegree value is used to set the CURRENT DEGREE special register on the data source. If currentDegree is not set, no value is passed to the data source.

**currentFunctionPath**

Specifies the SQL path that is used to resolve unqualified data type names and function names in SQL statements that are in JDBC programs. The data type of this property is String. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 254 bytes. For a DB2 for z/OS server, the maximum length is 2048 bytes. The value is a comma-separated list of schema names. Those names can be ordinary or delimited identifiers.

**currentMaintainedTableTypesForOptimization**

Specifies a value that identifies the types of objects that can be considered when the data source optimizes the processing of dynamic SQL queries. This register contains a keyword representing table types. The data type of this property is String.

Possible values of currentMaintainedTableTypesForOptimization are:

- **ALL**
  - Indicates that all materialized query tables will be considered.

- **NONE**
  - Indicates that no materialized query tables will be considered.

- **SYSTEM**
  - Indicates that only system-maintained materialized query tables that are refresh deferred will be considered.

- **USER**
  - Indicates that only user-maintained materialized query tables that are refresh deferred will be considered.

**currentPackagePath**

Specifies a comma-separated list of collections on the server. The database server searches these collections for JDBC and SQLJ packages.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.

**currentPackageSet**

Specifies the collection ID to search for JDBC and SQLJ packages. The data type of this property is String. The default is NULLID. If currentPackageSet is set, its value overrides the value of jdbcCollection.

Multiple instances of the IBM Data Server Driver for JDBC and SQLJ can be installed at a database server by running the DB2Binder utility multiple times. The DB2binder utility includes a -collection option that lets the installer specify the collection ID for each IBM Data Server Driver for JDBC and SQLJ instance. To choose an instance of the IBM Data Server Driver for JDBC and SQLJ for a connection, you specify a currentPackageSet value that matches the collection ID for one of the IBM Data Server Driver for JDBC and SQLJ instances.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.
**currentRefreshAge**
Specifies a timestamp duration value that is the maximum duration since a
REFRESH TABLE statement was processed on a system-maintained REFRESH
DEFERRED materialized query table such that the materialized query table can
be used to optimize the processing of a query. This property affects dynamic
statement cache matching. The data type of this property is long.

**currentSchema**
Specifies the default schema name that is used to qualify unqualified database
objects in dynamically prepared SQL statements. The value of this property
sets the value in the CURRENT SCHEMA special register on the database
server. The schema name is case-sensitive, and must be specified in uppercase
characters.

**cursorSensitivity**
Specifies whether the java.sql.ResultSet.TYPE_SCROLL_SENSITIVE value for a
JDBC ResultSet maps to the SENSITIVE DYNAMIC attribute, the SENSITIVE
STATIC attribute, or the ASENSITIVE attribute for the underlying database
cursor. The data type of this property is int. Possible values are
TYPE_SCROLL_SENSITIVE_STATICO (0), TYPE_SCROLL_SENSITIVE_DYNAMIC (1), or
TYPE_SCROLL_ASENSITIVE (2). The default is TYPE_SCROLL_SENSITIVE_STATIC.

If the data source does not support sensitive dynamic scrollable cursors, and
TYPE_SCROLL_SENSITIVE_DYNAMIC is requested, the JDBC driver accumulates a
warning and maps the sensitivity to SENSITIVE STATIC. For DB2 for i
database servers, which do not support sensitive static cursors,
java.sql.ResultSet.TYPE_SCROLL_SENSITIVE always maps to SENSITIVE
DYNAMIC.

**dateFormat**
Specifies:
- The format in which the String argument of the PreparedStatement.setString
  method against a DATE column must be specified.
- The format in which the result of the ResultSet.getString or
  CallableStatement.getString method against a DATE column is returned.

The data type of dateFormat is int.
Possible values of dateFormat are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.ISO</td>
<td>1</td>
<td>yyyy-mm-dd</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.USA</td>
<td>2</td>
<td>mm/dd/yyyy</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.EUR</td>
<td>3</td>
<td>dd.mm.yyyy</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.JIS</td>
<td>4</td>
<td>yyyy-mm-dd</td>
</tr>
</tbody>
</table>

The default is com.ibm.db2.jcc.DB2BaseDataSource.ISO.

**decimalRoundingMode**
Specifies the rounding mode for decimal floating-point values on DB2 for
z/OS Version 9 or later, or DB2 Database for Linux, UNIX, and Windows
database servers.

Possible values are:

**DB2BaseDataSource.ROUND_DOWN (1)**
Rounds the value towards 0 (truncation). The discarded digits are
ignored.
DB2BaseDataSource.ROUND_CEILING (2)
Rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by 1.

DB2BaseDataSource.ROUND_HALF_EVEN (3)
Rounds the value to the nearest value; if the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represent greater than half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 if its rightmost digit is odd (to make an even digit).

DB2BaseDataSource.ROUND_HALF_UP (4)
Rounds the value to the nearest value; if the values are equidistant, rounds the value away from zero. If the discarded digits represent greater than or equal to half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored.

DB2BaseDataSource.ROUND_FLOOR (6)
Rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1.

DB2BaseDataSource.ROUND_UNSET (-2147483647)
No rounding mode was explicitly set. The IBM Data Server Driver for JDBC and SQLJ does not use the decimalRoundingMode to set the rounding mode on the data source.

The IBM Data Server Driver for JDBC and SQLJ uses the following values for its rounding mode:
• For DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows database servers, the rounding mode is ROUND_HALF_EVEN for decimal floating-point values.

If decimalRoundingMode is set, the decimalRoundingMode value is used to set the CURRENT DECFLOAT Rounding MODE special register on DB2 for z/OS database servers.

enableRowsetSupport
Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH for forward-only cursors or scrollable cursors, if the data source supports multiple-row FETCH. The data type of this property is int.

When enableRowsetSupport is set, its value overrides the useRowsetCursor property value.

Possible values are:

DB2BaseDataSource.YES (1)
Specifies that:
• For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row FETCH is used for scrollable cursors and forward-only cursors, if the data source supports multiple-row FETCH.
For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, multiple-row fetch is used for scrollable cursors, if the data source supports multiple-row FETCH.

DB2BaseDataSource.NO (2)
   Specifies that multiple-row fetch is not used.

DB2BaseDataSource.NOT_SET (0)
   Specifies that if the enableRowsetSupport property is not set:
   • For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row fetch is not used.
   • For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, the useRowsetCursor property determines whether multiple-row fetch is used for scrollable cursors.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS, multiple-row fetch is not used.
Therefore, if progressive streaming is used for a FETCH operation, multiple-row FETCH is not used.

encryptionAlgorithm
   Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses 56-bit DES (weak) encryption or 256-bit AES (strong) encryption. The data type of this property is int. Possible values are:
   1   The driver uses 56-bit DES encryption.
   2   The driver uses 256-bit AES encryption, if the database server supports it. 256-bit AES encryption is available for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity only.

encryptionAlgorithm can be specified only if the securityMechanism value is ENCRYPTED_PASSWORD_SECURITY (7) or ENCRYPTED_USER_AND_PASSWORD_SECURITY (9).

fullyMaterializeInputStreams
   Indicates whether streams are fully materialized before they are sent from the client to a data source. The data type of this property is boolean. The default is false.

   If the value of fullyMaterializeInputStreams is true, the JDBC driver fully materialized the streams before sending them to the server.

gssCredential
   For a data source that uses Kerberos security, specifies a delegated credential that is passed from another principal. The data type of this property is org.ietf.jgss.GSSCredential. Delegated credentials are used in multi-tier environments, such as when a client connects to WebSphere Application Server, which, in turn, connects to the data source. You obtain a value for this property from the client, by invoking the GSSContext.getDelegCred method. GSSContext is part of the IBM Java Generic Security Service (GSS) API. If you set this property, you also need to set the Mechanism and KerberosServerPrincipal properties.

   This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
For more information on using Kerberos security with the IBM Data Server Driver for JDBC and SQLJ, see "Using Kerberos security under the IBM Data Server Driver for JDBC and SQLJ".

**kerberosServerPrincipal**
For a data source that uses Kerberos security, specifies the name that is used for the data source when it is registered with the Kerberos Key Distribution Center (KDC). The data type of this property is String.

This property is applicable only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**pdqProperties**
Specifies properties that control the interaction between the IBM Data Server Driver for JDBC and SQLJ and the client optimization feature of pureQuery.

The data type of this property is String.

Set the pdqProperties property only if you are using the client optimization feature of pureQuery. See the Integrated Data Management Information Center for information about valid values for pdqProperties.

**readOnly**
Specifies whether the connection is read-only. The data type of this property is boolean. The default is `false`.

**resultSetHoldabilityForCatalogQueries**
Specifies whether cursors for queries that are executed on behalf of DatabaseMetaData methods remain open after a commit operation. The data type of this property is int.

When an application executes DatabaseMetaData methods, the IBM Data Server Driver for JDBC and SQLJ executes queries against the catalog of the target data source. By default, the holdability of those cursors is the same as the holdability of application cursors. To use different holdability for catalog queries, use the resultSetHoldabilityForCatalogQueries property. Possible values are:

- **DB2BaseDataSource.HOLD_CURSORS_OVER_COMMIT (1)**
  Leave cursors for catalog queries open after a commit operation, regardless of the resultSetHoldability setting.

- **DB2BaseDataSource.CLOSE_CURSORS_AT_COMMIT (2)**
  Close cursors for catalog queries after a commit operation, regardless of the resultSetHoldability setting.

- **DB2BaseDataSource.NOT_SET (0)**
  Use the resultSetHoldability setting for catalog queries. This is the default value.

**returnAlias**
Specifies whether the JDBC driver returns rows for table aliases and synonyms for DatabaseMetaData methods that return table information, such as getTables. The data type of returnAlias is int. Possible values are:

- **0**
  Do not return rows for aliases or synonyms of tables in output from DatabaseMetaData methods that return table information.

- **1**
  For tables that have aliases or synonyms, return rows for aliases and synonyms of those tables, as well as rows for the tables, in output from DatabaseMetaData methods that return table information. This is the default.
streamBufferSize
Specifies the size, in bytes, of the JDBC driver buffers for chunking LOB or XML data. The JDBC driver uses the streamBufferSize value whether or not it uses progressive streaming. The data type of streamBufferSize is int. The default is 1048576.

If the JDBC driver uses progressive streaming, LOB or XML data is materialized if it fits in the buffers, and the driver does not use the fullyMaterializeLobData property.

DB2 for z/OS Version 9.1 and later supports progressive streaming for LOBs and XML objects. DB2 Database for Linux, UNIX, and Windows Version 9.5 and later, and IBM Informix (IDS) Version 11.50 and later support progressive streaming for LOBs.

supportsAsynchronousXARollback
Specifies whether the IBM Data Server Driver for JDBC and SQLJ supports asynchronous XA rollback operations. The data type of this property is int. The default is DB2BaseDataSource.NO (2). If the application runs against a BEA WebLogic Server application server, set supportsAsynchronousXARollback to DB2BaseDataSource.YES (1).

sysSchema
Specifies the schema of the shadow catalog tables or views that are searched when an application invokes a DatabaseMetaData method. The sysSchema property was formerly called cliSchema.

timeFormat
Specifies:
- The format in which the String argument of the PreparedStatement.setString method against a TIME column must be specified.
- The format in which the result of the ResultSet.getString or CallableStatement.getString method against a TIME column is returned.

The data type of timeFormat is int.

Possible values of timeFormat are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer value</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.ISO</td>
<td>1</td>
<td>hh:mm:ss</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.USA</td>
<td>2</td>
<td>hh:mm am or hh:mm pm</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.EUR</td>
<td>3</td>
<td>hh:mm:ss</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.JIS</td>
<td>4</td>
<td>hh:mm:ss</td>
</tr>
</tbody>
</table>

The default is com.ibm.db2.jcc.DB2BaseDataSource.ISO.

useCachedCursor
Specifies whether the underlying cursor for PreparedStatement objects is cached and reused on subsequent executions of the PreparedStatement. The data type of useCachedCursor is boolean.

If useCachedCursor is set to true, the cursor for PreparedStatement objects is cached, which can improve performance. true is the default.

Set useCachedCursor to false if PreparedStatement objects access tables whose column types or lengths change between executions of those PreparedStatement objects.
useJDBC4ColumnNameAndLabelSemantics

Specifies how the IBM Data Server Driver for JDBC and SQLJ handles column labels in ResultSetMetaData.getColumnName, ResultSetMetaData.getColumnLabel, and ResultSet.findColumn method calls.

Possible values are:

DB2BaseDataSource.YES (1)

The IBM Data Server Driver for JDBC and SQLJ uses the following rules, which conform to the JDBC 4.0 specification, to determine the value that ResultSetMetaData.getColumnName, ResultSetMetaData.getColumnLabel, and ResultSet.findColumn return:

- The column name that is returned by ResultSetMetaData.getColumnName is its name from the database.
- The column label that is returned by ResultSetMetaData.getColumnLabel is the label that is specified with the SQL AS clause. If the SQL AS clause is not specified, the label is the name of the column.
- ResultSet.findColumn takes the label for the column, as specified with the SQL AS clause, as input. If the SQL AS clause was not specified, the label is the column name.
- The IBM Data Server Driver for JDBC and SQLJ does not use a column label that is assigned by the SQL LABEL ON statement.

These rules apply to IBM Data Server Driver for JDBC and SQLJ version 3.50 and later, for connections to the following database systems:

- DB2 for z/OS Version 8 or later
- DB2 Database for Linux, UNIX, and Windows Version 8.1 or later
- DB2 UDB for iSeries® V5R3 or later

For earlier versions of the driver or the database systems, the rules for a useJDBC4ColumnNameAndLabelSemantics value of DB2BaseDataSource.NO apply, even if useJDBC4ColumnNameAndLabelSemantics is set to DB2BaseDataSource.YES.

DB2BaseDataSource.NO (2)

The IBM Data Server Driver for JDBC and SQLJ uses the following rules to determine the values that ResultSetMetaData.getColumnName, ResultSetMetaData.getColumnLabel, and ResultSet.findColumn return:

If the data source does not support the LABEL ON statement, or the source column is not defined with the LABEL ON statement:

- The value that is returned by ResultSetMetaData.getColumnName is its name from the database, if no SQL AS clause is specified. If the SQL AS clause is specified, the value that is returned is the column label.
- The value that is returned by ResultSetMetaData.getColumnLabel is the label that is specified with the SQL AS clause. If the SQL AS clause is not specified, the value that is returned is the name of the column.
- ResultSet.findColumn takes the column name as input.

If the source column is defined with the LABEL ON statement:

- The value that is returned by ResultSetMetaData.getColumnName is the column name from the database, if no SQL AS clause is
specified. If the SQL AS clause is specified, the value that is returned is the column label that is specified in the AS clause.

- The value that is returned by ResultSetMetaData.getColumnLabel is the label that is specified in the LABEL ON statement.
- ResultSet.findColumn takes the column name as input.

These rules conform to the behavior of the IBM Data Server Driver for JDBC and SQLJ before Version 3.50.

**DB2BaseDataSource.NOT_SET (0)**

This is the default behavior.

For the IBM Data Server Driver for JDBC and SQLJ version 3.50 and earlier, the default behavior for useJDBC4ColumnNameAndLabelSemantics is the same as the behavior for DB2BaseDataSource.NO.

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later:

- The default behavior for useJDBC4ColumnNameAndLabelSemantics is the same as the behavior for DB2BaseDataSource.YES, for connections to the following database systems:
  - DB2 for z/OS Version 8 or later
  - DB2 Database for Linux, UNIX, and Windows Version 8.1 or later
  - DB2 UDB for iSeries V5R3 or later
- For connections to earlier versions of these database systems, the default behavior for useJDBC4ColumnNameAndLabelSemantics is DB2BaseDataSource.NO.

**com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements**

Controls an internal statement cache that is associated with a PooledConnection. The data type of this property is int. Possible values are:

- **positive integer**
  
  Enables the internal statement cache for a PooledConnection, and specifies the number of statements that the IBM Data Server Driver for JDBC and SQLJ keeps open in the cache.

- **0 or negative integer**
  
  Disables internal statement caching for the PooledConnection. 0 is the default.

maxStatements controls the internal statement cache that is associated with a PooledConnection only when the PooledConnection object is created. maxStatements has no effect on caching in an already existing PooledConnection object.

maxStatements applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, and toIBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**Common IBM Data Server Driver for JDBC and SQLJ properties for DB2 for z/OS and IDS**

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to IBM Informix (IDS) and DB2 for z/OS database servers.

Properties that apply to IDS and DB2 for z/OS are:
enableConnectionConcentrator
Indicates whether the connection concentrator function of the IBM Data Server Driver for JDBC and SQLJ is enabled.

The data type of enableConnectionConcentrator is boolean. The default is false. However, if enableSysplexWLB is set to true, the default is true.

enableConnectionConcentrator applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

enableSysplexWLB
Indicates whether the Sysplex workload balancing function of the IBM Data Server Driver for JDBC and SQLJ is enabled. The data type of enableSysplexWLB is boolean. The default is false.

enableSysplexWLB applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

keepDynamic
Specifies whether the data source keeps already prepared dynamic SQL statements in the dynamic statement cache after commit points so that those prepared statements can be reused. The data type of this property is int. Valid values are DB2BaseDataSource.YES (1) and DB2BaseDataSource.NO (2).

If the keepDynamic property is not specified, the keepDynamic value is DB2BaseDataSource.NOT_SET (0). If the connection is to a DB2 for z/OS server, caching of dynamic statements for a connection is not done if the property is not set. If the connection is to an IDS data source, caching of dynamic statements for a connection is done if the property is not set.

keepDynamic is used with the DB2Binder -keepdynamic option. The keepDynamic property value that is specified must match the -keepdynamic value that was specified when DB2Binder was run.

For a DB2 for z/OS database server, dynamic statement caching can be done only if the EDM dynamic statement cache is enabled on the data source. The CACHEDYN subsystem parameter must be set to DB2BaseDataSource.YES to enable the dynamic statement cache.

maxTransportObjects
Specifies the maximum number of transport objects that can be used for all connections with the associated DataSource object. The IBM Data Server Driver for JDBC and SQLJ uses transport objects and a global transport objects pool to support the connection concentrator and Sysplex workload balancing. There is one transport object for each physical connection to the data source.

The data type of this property is int.

The maxTransportObjects value is ignored if the enableConnectionConcentrator or enableSysplexWLB properties are not set to enable the use of the connection concentrator or Sysplex workload balancing.

If the maxTransportObjects value has not been reached, and a transport object is not available in the global transport objects pool, the pool creates a new transport object. If the maxTransportObjects value has been reached, the application waits for the amount of time that is specified by the db2.jcc.maxTransportObjectWaitTime configuration property. After that amount of time has elapsed, if there is still no available transport object in the pool, the pool throws an SQLException.

maxTransportObjects does not override the db2.jcc.maxTransportObjects configuration property. maxTransportObjects has no effect on connections from...
other DataSource objects. If the maxTransportObjects value is larger than the db2.jcc.maxTransportObjects value, maxTransportObjects does not increase the db2.jcc.maxTransportObjects value.

The default value for maxTransportObjects is -1, which means that the number of transport objects for the DataSource is limited only by the db2.jcc.maxTransportObjects value for the driver.

Common IBM Data Server Driver for JDBC and SQLJ properties for IDS and DB2 Database for Linux, UNIX, and Windows

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply to IBM Informix (IDS) and DB2 Database for Linux, UNIX, and Windows database servers.

Properties that apply to IDS and DB2 Database for Linux, UNIX, and Windows are:

currentLockTimeout
  Specifies whether DB2 Database for Linux, UNIX, and Windows servers wait for a lock when the lock cannot be obtained immediately. The data type of this property is int. Possible values are:

  integer  Wait for integer seconds. integer is between -1 and 32767, inclusive.

  LOCK_TIMEOUT_NO_WAIT
    Do not wait for a lock. This is the default.

  LOCK_TIMEOUT_WAIT_INDEFINITELY
    Wait indefinitely for a lock.

  LOCK_TIMEOUT_NOT_SET
    Use the default for the data source.

IBM Data Server Driver for JDBC and SQLJ properties for DB2 Database for Linux, UNIX, and Windows

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to DB2 Database for Linux, UNIX, and Windows servers.

Those properties are:

currentExplainMode
  Specifies the value for the CURRENT EXPLAIN MODE special register. The CURRENT EXPLAIN MODE special register enables and disables the Explain facility. The data type of this property is String. The maximum length is 254 bytes. This property applies only to connections to data sources that support the CURRENT EXPLAIN MODE special register.

currentExplainSnapshot
  Specifies the value for the CURRENT EXPLAIN SNAPSHOT special register. The CURRENT EXPLAIN SNAPSHOT special register enables and disables the Explain snapshot facility. The data type of this property is String. The maximum length is eight bytes. This property applies only to connections to
data sources that support the CURRENT EXPLAIN SNAPSHOT special register, such as DB2 Database for Linux, UNIX, and Windows.

**currentQueryOptimization**

Specifies a value that controls the class of query optimization that is performed by the database manager when it binds dynamic SQL statements. The data type of this property is int. The possible values of currentQueryOptimization are:

- 0 Specifies that a minimal amount of optimization is performed to generate an access plan. This class is most suitable for simple dynamic SQL access to well-indexed tables.
- 1 Specifies that optimization roughly comparable to DB2 Database for Linux, UNIX, and Windows Version 1 is performed to generate an access plan.
- 2 Specifies a level of optimization higher than that of DB2 Database for Linux, UNIX, and Windows Version 1, but at significantly less optimization cost than levels 3 and above, especially for very complex queries.
- 3 Specifies that a moderate amount of optimization is performed to generate an access plan.
- 5 Specifies a significant amount of optimization is performed to generate an access plan. For complex dynamic SQL queries, heuristic rules are used to limit the amount of time spent selecting an access plan. Where possible, queries will use materialized query tables instead of the underlying base tables.
- 7 Specifies a significant amount of optimization is performed to generate an access plan. This value is similar to 5 but without the heuristic rules.
- 9 Specifies the maximum amount of optimization is performed to generate an access plan. This can greatly expand the number of possible access plans that are evaluated. This class should be used to determine if a better access plan can be generated for very complex and very long-running queries using large tables. Explain and performance measurements can be used to verify that a better plan has been generated.

**optimizationProfile**

Specifies an optimization profile that is used during SQL optimization. The data type of this property is String. The optimizationProfile value is used to set the OPTIMIZATION PROFILE special register. The default is null.

optimizationProfile applies to DB2 Database for Linux, UNIX, and Windows servers only.

**optimizationProfileToFlush**

Specifies the name of an optimization profile that is to be removed from the optimization profile cache. The data type of this property is String. The default is null.

**plugin**

The name of a client-side JDBC security plug-in. This property has the `Object` type and contains a new instance of the JDBC security plug-in method.

**pluginName**

The name of a server-side security plug-in module.
retryWithAlternativeSecurityMechanism
Specifies whether the IBM Data Server Driver for JDBC and SQLJ retries a
connection with an alternative security mechanism if the security mechanism
that is specified by property securityMechanism is not supported by the data
source. The data type of this property is int. Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)
  Retry the connection using an alternative security mechanism. The IBM
  Data Server Driver for JDBC and SQLJ issues warning code +4222 and
  retries the connection with the most secure available security
  mechanism.

com.ibm.db2.jcc.DB2BaseDataSource.NO (2) or
com.ibm.db2.jcc.DB2BaseDataSource.NOT_SET (0)
  Do not retry the connection using an alternative security mechanism.

retryWithAlternativeSecurityMechanism applies to IBM Data Server Driver for
JDBC and SQLJ type 4 connectivity connections to DB2 Database for Linux,
UNIX, and Windows only.

useTransactionRedirect
Specifies whether the DB2 system directs SQL statements to different database
partitions for better performance. The data type of this property is boolean.
The default is false.

This property is applicable only under the following conditions:
• The connection is to a DB2 Database for Linux, UNIX, and Windows server
  that uses the Database Partitioning Feature (DPF).
• The partitioning key remains constant throughout a transaction.

If useTransactionRedirect is true, the IBM Data Server Driver for JDBC and
SQLJ sends connection requests to the DPF node that contains the target data
of the first directable statement in the transaction. DB2 Database for Linux,
UNIX, and Windows then directs the SQL statement to different partitions as
needed.

IBM Data Server Driver for JDBC and SQLJ properties for DB2
for z/OS
Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to
DB2 for z/OS servers.

Those properties are:

accountingInterval
Specifies whether DB2 accounting records are produced at commit points or on
termination of the physical connection to the data source. The data type of this
property is String.

If the value of accountingInterval is "COMMIT", and there are no open, held
cursors, DB2 writes an accounting record each time that the application
commits work. If the value of accountingInterval is "COMMIT", and the
application performs a commit operation while a held cursor is open, the
accounting interval spans that commit point and ends at the next valid
accounting interval end point. If the value of accountingInterval is not
"COMMIT", accounting records are produced on termination of the physical
connection to the data source.
The accountingInterval property sets the *accounting-interval* parameter for an underlying RRSAF signon call. If the value of subsystem parameter ACCUMACC is not NO, the ACCUMACC value overrides the accountingInterval setting.

accountingInterval applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. accountingInterval is not applicable to connections under CICS or IMS, or for Java stored procedures.

The accountingInterval property overrides the db2.jcc.accountingInterval configuration property.

**charOutputSize**

Specifies the maximum number of bytes to use for INOUT or OUT stored procedure parameters that are registered as Types.CHAR. charOutputSize applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers.

Because DESCRIBE information for stored procedure INOUT and OUT parameters is not available at run time, by default, the IBM Data Server Driver for JDBC and SQLJ sets the maximum length of each character INOUT or OUT parameter to 32767. For stored procedures with many Types.CHAR parameters, this maximum setting can result in allocation of much more storage than is necessary.

To use storage more efficiently, set charOutputSize to the largest expected length for any Types.CHAR INOUT or OUT parameter.

charOutputSize has no effect on INOUT or OUT parameters that are registered as Types.VARCHAR or Types.LONGVARCHAR. The driver uses the default length of 32767 for Types.VARCHAR and Types.LONGVARCHAR parameters.

The value that you choose for charOutputSize needs to take into account the possibility of expansion during character conversion. Because the IBM Data Server Driver for JDBC and SQLJ has no information about the server-side CCSID that is used for output parameter values, the driver requests the stored procedure output data in UTF-8 Unicode. The charOutputSize value needs to be the maximum number of bytes that are needed after the parameter value is converted to UTF-8 Unicode. UTF-8 Unicode characters can require up to three bytes. (The euro symbol is an example of a three-byte UTF-8 character.) To ensure that the value of charOutputSize is large enough, if you have no information about the output data, set charOutputSize to three times the defined length of the largest CHAR parameter.

**clientUser**

Specifies the current client user name for the connection. This information is for client accounting purposes. Unlike the JDBC connection user name, this value can change during a connection. For a DB2 for z/OS server, the maximum length is 16 bytes.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

**clientWorkstation**

Specifies the workstation name for the current client for the connection. This information is for client accounting purposes. This value can change during a connection. The data type of this property is String. For a DB2 for z/OS server, the maximum length is 18 bytes. A Java empty string (""") is valid for this value, but a Java null value is not valid.
This property applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

**currentSQLID**

Specifies:
- The authorization ID that is used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
- The owner of a table space, database, storage group, or synonym that is created by a dynamically issued CREATE statement.
- The implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements.

`currentSQLID` sets the value in the `CURRENT SQLID` special register on a DB2 for z/OS server. If the `currentSQLID` property is not set, the default schema name is the value in the `CURRENT SQLID` special register.

**enableMultiRowInsertSupport**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT for batched INSERT or MERGE operations, when the target data server is a DB2 for z/OS server that supports multi-row INSERT. The batch operations must be PreparedStatement calls with parameter markers. The data type of this property is boolean. The default is `true`.

The `enableMultiRowInsertSupport` value cannot be changed for the duration of a connection. `enableMultiRowInsertSupport` must be set to false if INSERT FROM SELECT statements are executed in a batch. Otherwise, the driver throws a BatchUpdateException.

**jdbcCollection**

Specifies the collection ID for the packages that are used by an instance of the IBM Data Server Driver for JDBC and SQLJ at run time. The data type of `jdbcCollection` is `String`. The default is `NULLID`.

This property is used with the `DB2Binder -collection` option. The `DB2Binder` utility must have previously bound IBM Data Server Driver for JDBC and SQLJ packages at the server using a `-collection` value that matches the `jdbcCollection` value.

The `jdbcCollection` setting does not determine the collection that is used for SQLJ applications. For SQLJ, the collection is determined by the `-collection` option of the SQLJ customizer.

`jdbcCollection` does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

**maxRowsetSize**

Specifies the maximum number of bytes that are used for rowset buffering for each statement, when the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH for cursors. The data type of this property is int. The default is `32767`.

`maxRowsetSize` applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

**reportLongTypes**

Specifies whether `DatabaseMetaData` methods report LONG VARCHAR and LONG VARGRAPHIC column data types as long data types. The data type of this property is short. Possible values are:
Specifies that DatabaseMetaData methods that return information about a LONG VARCHAR or LONG VARGRAPHIC column return java.sql.Types.VARCHAR in the DATA_TYPE column and VARCHAR or VARGRAPHIC in the TYPE_NAME column of the result set. This is the default for DB2 for z/OS Version 9 or later.

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)

Specifies that DatabaseMetaData methods that return information about a LONG VARCHAR or LONG VARGRAPHIC column return java.sql.Types.LONGVARCHAR in the DATA_TYPE column and LONG VARCHAR or LONG VARGRAPHIC in the TYPE_NAME column of the result set.

sendCharInputsUTF8

Specifies whether the IBM Data Server Driver for JDBC and SQLJ converts character input data to the CCSID of the DB2 for z/OS database server, or sends the data in UTF-8 encoding for conversion by the database server. sendCharInputsUTF8 applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers only. The data type of this property is int. If this property is also set at the driver level (db2.jdbc.sendCharInputsUTF8), this value overrides the driver-level value.

Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.NO (2)

Specifies that the IBM Data Server Driver for JDBC and SQLJ converts character input data to the target encoding before the data is sent to the DB2 for z/OS database server. com.ibm.db2.jcc.DB2BaseDataSource.NO is the default.

com.ibm.db2.jcc.DB2BaseDataSource.YES (1)

Specifies that the IBM Data Server Driver for JDBC and SQLJ sends character input data to the DB2 for z/OS database server in UTF-8 encoding. The database server converts the data from UTF-8 encoding to the target CCSID.

Specify com.ibm.db2.jcc.DB2BaseDataSource.YES only if conversion to the target CCSID by the SDK for Java causes character conversion problems. The most common problem occurs when you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to insert a Unicode line feed character (U+000A) into a table column that has CCSID 37, and then retrieve that data from a non-z/OS client. If the SDK for Java does the conversion during insertion of the character into the column, the line feed character is converted to the EBCDIC new line character X'15'. However, during retrieval, some SDKs for Java on operating systems other than z/OS convert the X'15' character to the Unicode next line character (U+0085) instead of the line feed character (U+000A). The next line character causes unexpected behavior for some XML parsers. If you set sendCharInputsUTF8 to com.ibm.db2.jcc.DB2BaseDataSource.YES, the DB2 for z/OS database server converts the U+000A character to the EBCDIC line feed character X'25' during insertion into the column, so the character is always retrieved as a line feed character.

Conversion of data to the target CCSID on the database server might cause the IBM Data Server Driver for JDBC and SQLJ to use more memory than conversion by the driver. The driver allocates memory
for conversion of character data from the source encoding to the encoding of the data that it sends to the database server. The amount of space that the driver allocates for character data that is sent to a table column is based on the maximum possible length of the data. UTF-8 data can require up to three bytes for each character. Therefore, if the driver sends UTF-8 data to the database server, the driver needs to allocate three times the maximum number of characters in the input data. If the driver does the conversion, and the target CCSID is a single-byte CCSID, the driver needs to allocate only the maximum number of characters in the input data.

**sqljEnableClassLoaderSpecificProfiles**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ allows using and loading of SQLJ profiles with the same Java name in multiple J2EE application (.ear) files. The data type of this property is boolean. The default is false. sqljEnableClassLoaderSpecificProfiles is a DataSource property. This property is primarily intended for use with WebSphere Application Server.

**ssid**

Specifies the name of the local DB2 for z/OS subsystem to which a connection is established using IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. The data type of this property is String.

The ssid property overrides the db2.jcc ssid configuration property.

ssid can be the subsystem name for a local subsystem or a group attachment name or subgroup attachment name.

Specification of a single local subsystem name allows more than one subsystem on a single LPAR to be accessed as a local subsystem for connections that use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

ssid applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS.

**useRowsetCursor**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ always uses multiple-row FETCH for scrollable cursors if the data source supports multiple-row fetch. The data type of this property is boolean.

This property applies only to IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, or to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS. If the enableRowsetSupport property is not set, the default for useRowsetCursor is true. If the enableRowsetSupport property is set, the useRowsetCursor property is not used.

Applications that use the JDBC 1 technique for performing positioned update or delete operations should set useRowSetCursor to false. Those applications do not operate properly if the IBM Data Server Driver for JDBC and SQLJ uses multiple-row FETCH.

---

**IBM Data Server Driver for JDBC and SQLJ properties for IDS**

Some of the IBM Data Server Driver for JDBC and SQLJ properties apply only to IBM Informix (IDS) databases. Those properties correspond to IDS environment variables.

Properties that are shown in uppercase characters in the following information must be specified in uppercase. For those properties, getXXX and setXXX methods are formed by prepending the uppercase property name with get or set. For example:
boolean dbDate = DB2BaseDateSource.getDBDATE();

The IDS-specific properties are:

**DBANSIWARN**
Specifies whether the IBM Data Server Driver for JDBC and SQLJ instructs the IDS database to return an SQLWarning to the application if an SQL statement does not use ANSI-standard syntax. The data type of this property is boolean. Possible values are:

- **false or 0**
  - Do not send a value to the IDS database that instructs the database to return an SQLWarning to the application if an SQL statement does not use ANSI-standard syntax. This is the default.

- **true or 1**
  - Send a value to the IDS database that instructs the database to return an SQLWarning to the application if an SQL statement does not use ANSI-standard syntax.

You can use the DBANSIWARN IBM Data Server Driver for JDBC and SQLJ property to set the DBANSIWARN IDS property, but you cannot use the DBANSIWARN IBM Data Server Driver for JDBC and SQLJ property to reset the DBANSIWARN IDS property.

**DBDATE**
Specifies the end-user format of DATE values. The data type of this property is String. Possible values are in the description of the DBDATE environment variable in *IBM Informix Guide to SQL: Reference*.

The default value is "Y4MD-".

**DBPATH**
Specifies a colon-separated list of values that identify the database servers that contain databases. The data type of this property is String. Each value can be:

- A full path name
- A relative path name
- The server name of an IDS database server
- A server name and full path name

The default ".".

**DBSPACETEMP**
Specifies a comma-separated or colon-separated list of existing dbspaces in which temporary tables are placed. The data type of this property is String. If this property is not set, no value is sent to the server. The value for the DBSPACETEMP environment variable is used.

**DBTEMP**
Specifies the full path name of an existing directory in which temporary files and temporary tables are placed. The data type of this property is String. The default is "/tmp".

**DBUPSPACE**
Specifies the maximum amount of system disk space and maximum amount of memory, in kilobytes, that the UPDATE STATISTICS statement can use when it constructs multiple column distributions simultaneously. The data type of this property is String.

The format of DBUPSPACE is "maximum-disk-space:maximum-memory".
If this property is not set, no value is sent to the server. The value for the DBUPSPACE environment variable is used.

**DB_LOCALE**
Specifies the database locale, which the database server uses to process locale-sensitive data. The data type of this property is String. Valid values are the same as valid values for the DB_LOCALE environment variable. The default value is null.

**DELIMIDENT**
Specifies whether delimited SQL identifiers can be used in an application. The data type of this property is boolean. Possible values are:

- **false**
  The application cannot contain delimited SQL identifiers. Double quotation marks (") or single quotation marks (') delimit literal strings. This is the default.

- **true**
  The application can contain delimited SQL identifiers. Delimited SQL identifiers must be enclosed in double quotation marks ("'). Single quotation marks (') delimit literal strings.

**IFX_DIRECTIVES**
Specifies whether the optimizer allows query optimization directives from within a query. The data type of this property is String. Possible values are:

- **"1" or "ON"**
  Optimization directives are accepted.

- **"0" or "OFF"**
  Optimization directives are not accepted.

If this property is not set, no value is sent to the server. The value for the IFX_DIRECTIVES environment variable is used.

**IFX_EXTDIRECTIVES**
Specifies whether the optimizer allows external query optimization directives from the sysdirectives system catalog table to be applied to queries in existing applications. Possible values are:

- **"1" or "ON"**
  External query optimization directives are accepted.

- **"0" or "OFF"**
  External query optimization are not accepted.

If this property is not set, no value is sent to the server. The value for the IFX_EXTDIRECTIVES environment variable is used.

**IFX_UPDDDESC**
Specifies whether a DESCRIBE of an UPDATE statement is permitted. The data type of this property is String.

Any non-null value indicates that a DESCRIBE of an UPDATE statement is permitted. The default is "1".

**IFX_XASTDCOMPLIANCE_XAEND**
Specifies whether global transactions are freed only after an explicit rollback, or after any rollback. The data type of this property is String. Possible values are:

- **"0"**
  Global transactions are freed only after an explicit rollback. This behavior conforms to the X/Open XA standard.

- **"1"**
  Global transactions are freed after any rollback.
If this property is not set, no value is sent to the server. The value for the IFX_XASTDCOMPLIANCE_XAEND environment variable is used.

**INFORMIXOPCACHE**

Specifies the size of the memory cache, in kilobytes, for the staging-area blobspace of the client application. The data type of this property is String. A value of "0" indicates that the cache is not used.

If this property is not set, no value is sent to the server. The value for the INFORMIXOPCACHE environment variable is used.

**INFORMIXSTACKSIZE**

Specifies the stack size, in kilobytes, that the database server uses for the primary thread of a client session. The data type of this property is String.

If this property is not set, no value is sent to the server. The value for the INFORMIXSTACKSIZE environment variable is used.

**NODEFDAC**

Specifies whether the database server prevents default table privileges (SELECT, INSERT, UPDATE, and DELETE) from being granted to PUBLIC when a new table is created during the current session, in a database that is not ANSI compliant. The data type of this property is String. Possible values are:

"yes"  
The database server prevents default table privileges from being granted to PUBLIC when a new table is created during the current session, in a database that is not ANSI compliant.

"no"  
The database server does not prevent default table privileges from being granted to PUBLIC when a new table is created during the current session, in a database that is not ANSI compliant. This is the default.

**OPTCOMPIND**

Specifies the preferred method for performing a join operation on an ordered pair of tables. The data type of this property is String. Possible values are:

"0"  
The optimizer chooses a nested-loop join, where possible, over a sort-merge join or a hash join.

"1"  
When the isolation level is repeatable read, the optimizer chooses a nested-loop join, where possible, over a sort-merge join or a hash join. When the isolation level is not repeatable read, the optimizer chooses a join method based on costs.

"2"  
The optimizer chooses a join method based on costs, regardless of the transaction isolation mode.

If this property is not set, no value is sent to the server. The value for the OPTCOMPIND environment variable is used.

**OPTOFC**

Specifies whether to enable optimize-OPEN-FETCH-CLOSE functionality. The data type of this property is String. Possible values are:

"0"  
Disable optimize-OPEN-FETCH-CLOSE functionality for all threads of applications.

"1"  
Enable optimize-OPEN-FETCH-CLOSE functionality for all cursors in all threads of applications.

If this property is not set, no value is sent to the server. The value for the OPTOFCD environment variable is used.
PDQ_PRIORITY
Specifies the degree of parallelism that the database server uses. The PDQ_PRIORITY value affects how the database server allocates resources, including memory, processors, and disk reads. The data type of this property is String. Possible values are:

"HIGH"
When the database server allocates resources among all users, it gives as many resources as possible to queries.

"LOW" or "1"
The database server fetches values from fragmented tables in parallel.

"OFF" or "0"
Parallel processing is disabled.

If this property is not set, no value is sent to the server. The value for the PDQ_PRIORITY environment variable is used.

PSORT_DBTEMP
Specifies the full path name of a directory in which the database server writes temporary files that are used for a sort operation. The data type of this property is String.

If this property is not set, no value is sent to the server. The value for the PSORT_DBTEMP environment variable is used.

PSORT_NPROCS
Specifies the maximum number of threads that the database server can use to sort a query. The data type of this property is String. The maximum value of PSORT_NPROCS is "10".

If this property is not set, no value is sent to the server. The value for the PSORT_NPROCS environment variable is used.

STMT_CACHE
Specifies whether the shared-statement cache is enabled. The data type of this property is String. Possible values are:

"0"
The shared-statement cache is disabled.

"1"A 512 KB shared-statement cache is enabled.

If this property is not set, no value is sent to the server. The value for the STMT_CACHE environment variable is used.

dumpPool
Specifies the types of statistics on global transport pool events that are written, in addition to summary statistics. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

The data type of dumpPool is int. dumpPoolStatisticsOnSchedule and dumpPoolStatisticsOnScheduleFile must also be set for writing statistics before any statistics are written.

You can specify one or more of the following types of statistics with the db2.jcc.dumpPool property:

- DUMP_REMOVE_OBJECT (hexadecimal: X'01', decimal: 1)
- DUMP_GET_OBJECT (hexadecimal: X'02', decimal: 2)
- DUMP_WAIT_OBJECT (hexadecimal: X'04', decimal: 4)
- DUMP_SET_AVAILABLE_OBJECT (hexadecimal: X'08', decimal: 8)
- DUMP_CREATE_OBJECT (hexadecimal: X'10', decimal: 16)
- DUMP_SYSPLEX_MSG (hexadecimal: X'20', decimal: 32)
To trace more than one type of event, add the values for the types of events that you want to trace. For example, suppose that you want to trace DUMP_GET_OBJECT and DUMP_CREATE_OBJECT events. The numeric equivalents of these values are 2 and 16, so you specify 18 for the dumpPool value.

The default is 0, which means that only summary statistics for the global transport pool are written.

This property does not have a setXXX or a getXXX method.

dumpPoolStatisticsOnSchedule
Specifies how often, in seconds, global transport pool statistics are written to the file that is specified by dumpPoolStatisticsOnScheduleFile. The global transport object pool is used for the connection concentrator and Sysplex workload balancing.

The default is -1. -1 means that global transport pool statistics are not written.

This property does not have a setXXX or a getXXX method.

dumpPoolStatisticsOnScheduleFile
Specifies the name of the file to which global transport pool statistics are written. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

If dumpPoolStatisticsOnScheduleFile is not specified, global transport pool statistics are not written.

This property does not have a setXXX or a getXXX method.

maxTransportObjectIdleTime
Specifies the amount of time in seconds that an unused transport object stays in a global transport object pool before it can be deleted from the pool. Transport objects are used for the connection concentrator and Sysplex workload balancing.

The default value for maxTransportObjectIdleTime is 60. Setting maxTransportObjectIdleTime to a value less than 0 causes unused transport objects to be deleted from the pool immediately. Doing this is not recommended because it can cause severe performance degradation.

This property does not have a setXXX or a getXXX method.

maxTransportObjectWaitTime
Specifies the maximum amount of time in seconds that an application waits for a transport object if the maxTransportObjects value has been reached. Transport objects are used for the connection concentrator and Sysplex workload balancing. When an application waits for longer than the maxTransportObjectWaitTime value, the global transport object pool throws an SQLException.

The default value for maxTransportObjectWaitTime is -1. Any negative value means that applications wait forever.

This property does not have a setXXX or a getXXX method.

minTransportObjects
Specifies the lower limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When a JVM is created, there are no transport objects in the pool. Transport objects are added to the pool as they are needed. After the
minTransportObjects value is reached, the number of transport objects in the
global transport object pool never goes below the minTransportObjects value
for the lifetime of that JVM.

The default value for minTransportObjects is 0. Any value that is less than or
equal to 0 means that the global transport object pool can become empty.
This property does not have a setXXX or a getXXX method.

IBM Data Server Driver for JDBC and SQLJ configuration properties

The IBM Data Server Driver for JDBC and SQLJ configuration properties have
driver-wide scope.

The following table summarizes the configuration properties and corresponding
Connection or DataSource properties, if they exist.

<table>
<thead>
<tr>
<th>Configuration property name</th>
<th>Connection or DataSource property name: com.ibm.db2.jcc.DB2BaseDataSource. ...</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2.jcc.accountingInterval</td>
<td>accountingInterval</td>
<td>1 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.allowSqljDuplicateStaticQueries</td>
<td></td>
<td>4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.charOutputSize</td>
<td>charOutputSize</td>
<td>1 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.currentSchema</td>
<td>currentSchema</td>
<td>1 on page 313, 4 on page 313, 6 on page 313</td>
</tr>
<tr>
<td>db2.jcc.override.currentSchema</td>
<td>currentSchema</td>
<td>2 on page 313, 4 on page 313, 6 on page 313</td>
</tr>
<tr>
<td>db2.jcc.currentSQLID</td>
<td>currentSQLID</td>
<td>1 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.override.currentSQLID</td>
<td>currentSQLID</td>
<td>2 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.decimalRoundingMode</td>
<td>decimalRoundingMode</td>
<td>1 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.override.decimalRoundingMode</td>
<td>decimalRoundingMode</td>
<td>2 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.defaultSQLState</td>
<td></td>
<td>4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.disableSqljProfileCaching</td>
<td></td>
<td>4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.dumpPool</td>
<td>dumpPool</td>
<td>1 on page 313, 3 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.dumpPoolStatisticsOnSchedule</td>
<td>dumpPoolStatisticsOnSchedule</td>
<td>1 on page 313, 5 on page 313, 7 on page 313</td>
</tr>
<tr>
<td>db2.jcc.dumpPoolStatisticsOnScheduleFile</td>
<td>dumpPoolStatisticsOnScheduleFile</td>
<td>1 on page 313, 5 on page 313, 7 on page 313</td>
</tr>
<tr>
<td>db2.jcc.jmxEnabled</td>
<td></td>
<td>4 on page 313, 5 on page 313</td>
</tr>
<tr>
<td>db2.jcc.lobOutputSize</td>
<td></td>
<td>4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectIdleTime</td>
<td>maxTransportObjectIdleTime</td>
<td>1 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjectWaitTime</td>
<td>maxTransportObjectWaitTime</td>
<td>1 on page 313, 4 on page 313</td>
</tr>
<tr>
<td>db2.jcc.maxTransportObjects</td>
<td>maxTransportObjects</td>
<td>1 on page 313, 4 on page 313</td>
</tr>
</tbody>
</table>

Developing Java Applications
Table 54. Summary of Configuration properties and corresponding Connection and DataSource properties (continued)

<table>
<thead>
<tr>
<th>Configuration property name</th>
<th>Connection or DataSource property name:</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>db2.jcc.minTransportObjects</td>
<td>minTransportObjects</td>
<td>1, 4, 5</td>
</tr>
<tr>
<td>db2.jcc.pkList</td>
<td>pkList</td>
<td>1, 4</td>
</tr>
<tr>
<td>db2.jcc.planName</td>
<td>planName</td>
<td>1, 4</td>
</tr>
<tr>
<td>db2.jcc.progressiveStreaming</td>
<td>progressiveStreaming</td>
<td>1, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.override.progressiveStreaming</td>
<td>progressiveStreaming</td>
<td>1, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.rollbackOnShutdown</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>db2.jcc.sendCharInputsUTF8</td>
<td>sendCharInputsUTF8</td>
<td>1, 4</td>
</tr>
<tr>
<td>db2.jcc.sqljUncustomizedWarningOrException</td>
<td></td>
<td>1, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.ssid</td>
<td>ssid</td>
<td>1, 4</td>
</tr>
<tr>
<td>db2.jcc.traceDirectory</td>
<td>traceDirectory</td>
<td>1, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.override.traceDirectory</td>
<td>traceDirectory</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.traceFile</td>
<td>traceFile</td>
<td>1, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.override.traceFile</td>
<td>traceFile</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.traceFileAppend</td>
<td>traceFileAppend</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.override.traceFileAppend</td>
<td>traceFileAppend</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.traceLevel</td>
<td>traceLevel</td>
<td>1, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.override.traceLevel</td>
<td>traceLevel</td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.tracePolling</td>
<td></td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.tracePollingInterval</td>
<td></td>
<td>2, 4, 5, 6</td>
</tr>
<tr>
<td>db2.jcc.t2zosTraceFile</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>db2.jcc.t2zosTraceBufferSize</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>db2.jcc.t2zosTraceWrap</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>db2.jcc.useCcsid420ShapedConverter</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

Note:
1. The Connection or DataSource property setting overrides the configuration property setting. The configuration property provides a default value for the Connection or DataSource property.
2. The configuration property setting overrides the Connection or DataSource property.
3. The corresponding Connection or DataSource property is defined only for IBM Informix.
4. The configuration property applies to DB2 for z/OS.
5. The configuration property applies to IBM Informix.
6. The configuration property applies to DB2 Database for Linux, UNIX, and Windows.

The meanings of the configuration properties are:

**db2.jcc.accountingInterval**

Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. If the value of db2.jcc.accountingInterval is COMMIT, DB2 accounting records are produced at commit points. For example:

```
db2.jcc.accountingInterval=COMMIT
```

Otherwise, accounting records are produced on termination of the physical connection to the data source.

**db2.jcc.accountingInterval** applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. db2.jcc.accountingInterval is not applicable to connections under CICS or IMS, or for Java stored procedures.
You can override `db2.jcc.accountingInterval` by setting the `accountingInterval` property for a Connection or DataSource object.

This configuration property applies only to DB2 for z/OS.

**db2.jcc.allowSqljDuplicateStaticQueries**

Specifies whether multiple open iterators on a single SELECT statement in an SQLJ application are allowed under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity.

To enable this support, set `db2.jcc.allowSqljDuplicateStaticQueries` to YES or true.

**db2.jcc.charOutputSize**

Specifies the maximum number of bytes to use for INOUT or OUT stored procedure parameters that are registered as `Types.CHAR`.

Because DESCRIBE information for stored procedure INOUT and OUT parameters is not available at run time, by default, the IBM Data Server Driver for JDBC and SQLJ sets the maximum length of each character INOUT or OUT parameter to 32767. For stored procedures with many `Types.CHAR` parameters, this maximum setting can result in allocation of much more storage than is necessary.

To use storage more efficiently, set `db2.jcc.charOutputSize` to the largest expected length for any `Types.CHAR` INOUT or OUT parameter.

`db2.jcc.charOutputSize` has no effect on INOUT or OUT parameters that are registered as `Types.VARCHAR` or `Types.LONGVARCHAR`. The driver uses the default length of 32767 for `Types.VARCHAR` and `Types.LONGVARCHAR` parameters.

The value that you choose for `db2.jcc.charOutputSize` needs to take into account the possibility of expansion during character conversion. Because the IBM Data Server Driver for JDBC and SQLJ has no information about the server-side CCSID that is used for output parameter values, the driver requests the stored procedure output data in UTF-8 Unicode. The `db2.jcc.charOutputSize` value needs to be the maximum number of bytes that are needed after the parameter value is converted to UTF-8 Unicode. UTF-8 Unicode characters can require up to three bytes. (The euro symbol is an example of a three-byte UTF-8 character.) To ensure that the value of `db2.jcc.charOutputSize` is large enough, if you have no information about the output data, set `db2.jcc.charOutputSize` to three times the defined length of the largest CHAR parameter.

This configuration property applies only to DB2 for z/OS.

**db2.jcc.currentSchema or db2.jcc.override.currentSchema**

Specifies the default schema name that is used to qualify unqualified database objects in dynamically prepared SQL statements. This value of this property sets the value in the CURRENT SCHEMA special register on the database server. The schema name is case-sensitive, and must be specified in uppercase characters.

This configuration property applies only to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows.

**db2.jcc.currentSQLID or db2.jcc.override.currentSQLID**

Specifies:

- The authorization ID that is used for authorization checking on dynamically prepared CREATE, GRANT, and REVOKE SQL statements.
The owner of a table space, database, storage group, or synonym that is created by a dynamically issued CREATE statement.

The implicit qualifier of all table, view, alias, and index names specified in dynamic SQL statements.

currentSQLID sets the value in the CURRENT SQLID special register on a DB2 for z/OS server. If the currentSQLID property is not set, the default schema name is the value in the CURRENT SQLID special register.

This configuration property applies only to DB2 for z/OS.

db2.jcc.decimalRoundingMode or db2.jcc.override.decimalRoundingMode
Specifies the rounding mode for decimal or decimal floating-point values on DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows database servers, and for decimal values on all other data sources that support the decimal data type.

Possible values are:

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_DOWN (1)
   Rounds the value towards 0 (truncation). The discarded digits are ignored.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_CEILING (2)
   Rounds the value towards positive infinity. If all of the discarded digits are zero or if the sign is negative the result is unchanged other than the removal of the discarded digits. Otherwise, the result coefficient is incremented by 1.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_HALF_EVEN (3)
   Rounds the value to the nearest value; if the values are equidistant, rounds the value so that the final digit is even. If the discarded digits represent greater than half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. If they represent less than half, then the result coefficient is not adjusted (that is, the discarded digits are ignored). Otherwise the result coefficient is unaltered if its rightmost digit is even, or is incremented by 1 if its rightmost digit is odd (to make an even digit).

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_HALF_UP (4)
   Rounds the value to the nearest value; if the values are equidistant, rounds the value away from zero. If the discarded digits represent greater than or equal to half (0.5) of the value of one in the next left position then the result coefficient is incremented by 1. Otherwise the discarded digits are ignored.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_FLOOR (6)
   Rounds the value towards negative infinity. If all of the discarded digits are zero or if the sign is positive the result is unchanged other than the removal of discarded digits. Otherwise, the sign is negative and the result coefficient is incremented by 1.

com.ibm.db2.jcc.DB2BaseDataSource.ROUND_UNSET (-2147483647)
   No rounding mode was explicitly set. The IBM Data Server Driver for JDBC and SQLJ does not use the decimalRoundingMode to set the rounding mode on the data source.

The IBM Data Server Driver for JDBC and SQLJ uses the following values for its rounding mode:

• If the data source is DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows, the rounding mode is ROUND_HALF_EVEN for decimal or decimal floating-point values.
• For any other data source, the rounding mode is ROUND_DOWN for decimal values.

This configuration property applies only to DB2 for z/OS Version 9 or later, or DB2 Database for Linux, UNIX, and Windows Version 9.1 or later.

db2.jcc.defaultSQLState
Specifies the SQLSTATE value that the IBM Data Server Driver for JDBC and SQLJ returns to the client for SQLException or SQLWarning objects that have null SQLSTATE values. This configuration property can be specified in the following ways:

db2.jcc.defaultSQLState
If db2.jcc.defaultSQLState is specified with no value, the IBM Data Server Driver for JDBC and SQLJ returns 'FFFFF'.

db2.jcc.defaultSQLState=xxxxx
xxxxx is the value that the IBM Data Server Driver for JDBC and SQLJ returns when the SQLSTATE value is null. If xxxxx is longer than five bytes, the driver truncates the value to five bytes. If xxxxx is shorter than five bytes, the driver pads xxxxx on the right with blanks.

If db2.jcc.defaultSQLState is not specified, the IBM Data Server Driver for JDBC and SQLJ returns a null SQLSTATE value.

This configuration property applies only to DB2 for z/OS.

db2.jcc.disableSQLJProfileCaching
Specifies whether serialized profiles are cached when the JVM under which their application is running is reset. db2.jcc.disableSQLJProfileCaching applies only to applications that run in a resettable JVM (applications that run in the CICS, IMS, or Java stored procedure environment), and use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. Possible values are:

YES SQLJ serialized profiles are not cached every time the JVM is reset, so that new versions of the serialized profiles are loaded when the JVM is reset. Use this option when an application is under development, and new versions of the application and its serialized profiles are produced frequently.

NO SQLJ serialized profiles are cached when the JVM is reset. NO is the default.

This configuration property applies only to DB2 for z/OS.

db2.jcc.dumpPool
Specifies the types of statistics on global transport pool events that are written, in addition to summary statistics. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

db2.jcc.dumpPoolStatisticsOnSchedule and db2.jcc.dumpPoolStatisticsOnScheduleFile must also be set for writing statistics before any statistics are written.

You can specify one or more of the following types of statistics with the db2.jcc.dumpPool property:
• DUMP_REMOVE_OBJECT (hexadecimal: X'01', decimal: 1)
• DUMP_GET_OBJECT (hexadecimal: X'02', decimal: 2)
• DUMP_WAIT_OBJECT (hexadecimal: X'04', decimal: 4)
• DUMP_SET_AVAILABLE_OBJECT (hexadecimal: X'08', decimal: 8)
• DUMP_CREATE_OBJECT (hexadecimal: X'10', decimal: 16)
• DUMP_SYSPLEX_MSG (hexadecimal: X'20', decimal: 32)
• DUMP_POOL_ERROR (hexadecimal: X'80', decimal: 128)

To trace more than one type of event, add the values for the types of events that you want to trace. For example, suppose that you want to trace DUMP_GET_OBJECT and DUMP_CREATE_OBJECT events. The numeric equivalents of these values are 2 and 16, so you specify 18 for the db2.jcc.dumpPool value.

The default is 0, which means that only summary statistics for the global transport pool are written.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.dumpPoolStatisticsOnSchedule
Specifies how often, in seconds, global transport pool statistics are written to the file that is specified by db2.jcc.dumpPoolStatisticsOnScheduleFile. The global transport object pool is used for the connection concentrator and Sysplex workload balancing.

The default is -1. -1 means that global transport pool statistics are not written.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.dumpPoolStatisticsOnScheduleFile
Specifies the name of the file to which global transport pool statistics are written. The global transport pool is used for the connection concentrator and Sysplex workload balancing.

If db2.jcc.dumpPoolStatisticsOnScheduleFile is not specified, global transport pool statistics are not written.

This configuration property applies only to DB2 for z/OS or IBM Informix.

db2.jcc.jmxEnabled
Specifies whether the Java Management Extensions (JMX) is enabled for the IBM Data Server Driver for JDBC and SQLJ instance. JMX must be enabled before applications can use the remote trace controller.

Possible values are:

true or yes
Indicates that JMX is enabled.

Any other value
Indicates that JMX is disabled. This is the default.

db2.jcc.lobOutputSize
Specifies the number of bytes of storage that the IBM Data Server Driver for JDBC and SQLJ needs to allocate for output LOB values when the driver cannot determine the size of those LOBs. This situation occurs for LOB stored procedure output parameters. db2.jcc.lobOutputSize applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The default value for db2.jcc.lobOutputSize is 1048576. For systems with storage limitations and smaller LOBs, set the db2.jcc.lobOutputSize value to a lower number.

For example, if you know that the output LOB size is at most 64000, set db2.jcc.lobOutputSize to 64000.
This configuration property applies only to DB2 for z/OS.

**db2.jcc.maxTransportObjectIdleTime**

Specifies the amount of time in seconds that an unused transport object stays in a global transport object pool before it can be deleted from the pool. Transport objects are used for the connection concentrator and Sysplex workload balancing.

The default value for db2.jcc.maxTransportObjectIdleTime is 60. Setting db2.jcc.maxTransportObjectIdleTime to a value less than 0 causes unused transport objects to be deleted from the pool immediately. Doing this is not recommended because it can cause severe performance degradation.

This configuration property applies only to DB2 for z/OS or IBM Informix.

**db2.jcc.maxTransportObjects**

Specifies the upper limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When the number of transport objects in the pool reaches the db2.jcc.maxTransportObjects value, transport objects that have not been used for longer than the db2.jcc.maxTransportObjectIdleTime value are deleted from the pool.

The default value for db2.jcc.maxTransportObjects is -1. Any value that is less than or equal to 0 means that there is no limit to the number of transport objects in the global transport object pool.

This configuration property applies only to DB2 for z/OS or IBM Informix.

**db2.jcc.maxTransportObjectWaitTime**

Specifies the maximum amount of time in seconds that an application waits for a transport object if the db2.jcc.maxTransportObjects value has been reached. Transport objects are used for the connection concentrator and Sysplex workload balancing. When an application waits for longer than the db2.jcc.maxTransportObjectWaitTime value, the global transport object pool throws an SQLException.

The default value for db2.jcc.maxTransportObjectWaitTime is -1. Any negative value means that applications wait forever.

This configuration property applies only to DB2 for z/OS or IBM Informix.

**db2.jcc.minTransportObjects**

Specifies the lower limit for the number of transport objects in a global transport object pool for the connection concentrator and Sysplex workload balancing. When a JVM is created, there are no transport objects in the pool. Transport objects are added to the pool as they are needed. After the db2.jcc.minTransportObjects value is reached, the number of transport objects in the global transport object pool never goes below the db2.jcc.minTransportObjects value for the lifetime of that JVM.

The default value for db2.jcc.minTransportObjects is 0. Any value that is less than or equal to 0 means that the global transport object pool can become empty.

This configuration property applies only to DB2 for z/OS or IBM Informix.

**db2.jcc.pkList**

Specifies a package list that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. Specify this property if you do not bind plans for your SQLJ programs or for the JDBC driver. If you specify this property, **do not specify db2.jcc.planName**.
db2.jcc.pkList applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. db2.jcc.pkList does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the db2.jcc.pkList setting in those cases.

Recommendation: Use db2.jcc.pkList instead of db2.jcc.planName.

The format of the package list is:

```
/collection-ID.*
```

The default value of db2.jcc.pkList is NULLID.*.

If you specify the -collection parameter when you run `com.ibm.db2.jcc.DB2Binder`, the collection ID that you specify for IBM Data Server Driver for JDBC and SQLJ packages when you run `com.ibm.db2.jcc.DB2Binder` must also be in the package list for the db2.jcc.pkList property.

You can override db2.jcc.pkList by setting the pkList property for a Connection or DataSource object.

The following example specifies a package list for a IBM Data Server Driver for JDBC and SQLJ instance whose packages are in collection JDBCCID. SQLJ applications that are prepared under this driver instance are bound into collections SQLJCID1, SQLJCID2, or SQLJCID3.

```
db2.jcc.pkList=JDBCCID.*,SQLJCID1.*,SQLJCID2.*,SQLJCID3.*
```

This configuration property applies only to DB2 for z/OS.

**db2.jcc.planName**

Specifies a DB2 for z/OS plan name that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. Specify this property if you bind plans for your SQLJ programs and for the JDBC driver packages. If you specify this property, do not specify db2.jcc.pkList.

db2.jcc.planName applies only to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS. db2.jcc.planName does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the db2.jcc.planName setting in those cases.

If you do not specify this property or the db2.jcc.pkList property, the IBM Data Server Driver for JDBC and SQLJ uses the db2.jcc.pkList default value of NULLID.*.

If you specify db2.jcc.planName, you need to bind the packages that you produce when you run `com.ibm.db2.jcc.DB2Binder` into a plan whose name is the value of this property. You also need to bind all SQLJ packages into a plan whose name is the value of this property.

You can override db2.jcc.planName by setting the planName property for a Connection or DataSource object.

The following example specifies a plan name of MYPLAN for the IBM Data Server Driver for JDBC and SQLJ JDBC packages and SQLJ packages.

```
db2.jcc.planName=MYPLAN
```

This configuration property applies only to DB2 for z/OS.
**db2.jcc.progressiveStreaming** or **db2.jcc.override.progressiveStreaming**

Specifies whether the JDBC driver uses progressive streaming when progressive streaming is supported on the data source.

With progressive streaming, also known as dynamic data format, the data source dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects.

Valid values are:

1. Use progressive streaming, if the data source supports it.
2. Do not use progressive streaming.

**db2.jcc.rollbackOnShutdown**

Specifies whether DB2 for z/OS forces a rollback operation and disables further operations on JDBC connections that are in a unit of work during processing of JVM shutdown hooks.

db2.jcc.rollbackOnShutdown applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity only.

db2.jcc.rollbackOnShutdown does not apply to the CICS, IMS, stored procedure, or WebSphere Application Server environments.

Possible values are:

**yes or true**

The IBM Data Server Driver for JDBC and SQLJ directs DB2 for z/OS to force a rollback operation and disables further operations on JDBC connections that are in a unit of work during processing of JVM shutdown hooks.

**Any other value**

The IBM Data Server Driver for JDBC and SQLJ takes no action with respect to rollback processing during processing of JVM shutdown hooks. This is the default.

This configuration property applies only to DB2 for z/OS.

**db2.jcc.sendCharInputsUTF8**

Specifies whether the IBM Data Server Driver for JDBC and SQLJ converts character input data to the CCSID of the DB2 for z/OS database server, or sends the data in UTF-8 encoding for conversion by the database server.

db2.jcc.sendCharInputsUTF8 applies to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS database servers only. If this property is also set at the connection level, the connection-level setting overrides this value.

Possible values are:

**no, false, or 2**

Specifies that the IBM Data Server Driver for JDBC and SQLJ converts character input data to the target encoding before the data is sent to the DB2 for z/OS database server. This is the default.

**yes, true, or 1**

Specifies that the IBM Data Server Driver for JDBC and SQLJ sends character input data to the DB2 for z/OS database server in UTF-8 encoding. The data source converts the data from UTF-8 encoding to the target CCSID.

Specify yes, true, or 1 only if conversion to the target CCSID by the SDK for Java causes character conversion problems. The most common
problem occurs when you use IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to insert a Unicode line feed character (U+000A) into a table column that has CCSID 37, and then retrieve that data from a non-z/OS client. If the SDK for Java does the conversion during insertion of the character into the column, the line feed character is converted to the EBCDIC new line character X’15’. However, during retrieval, some SDKs for Java on operating systems other than z/OS convert the X’15’ character to the Unicode next line character (U+0085) instead of the line feed character (U+000A). The next line character causes unexpected behavior for some XML parsers.

If you set db2.jcc.sendCharInputsUTF8 to yes, the DB2 for z/OS database server converts the U+000A character to the EBCDIC line feed character X’25’ during insertion into the column, so the character is always retrieved as a line feed character.

Conversion of data to the target CCSID on the data source might cause the IBM Data Server Driver for JDBC and SQLJ to use more memory than conversion by the driver. The driver allocates memory for conversion of character data from the source encoding to the encoding of the data that it sends to the data source. The amount of space that the driver allocates for character data that is sent to a table column is based on the maximum possible length of the data. UTF-8 data can require up to three bytes for each character. Therefore, if the driver sends UTF-8 data to the data source, the driver needs to allocate three times the maximum number of characters in the input data. If the driver does the conversion, and the target CCSID is a single-byte CCSID, the driver needs to allocate only the maximum number of characters in the input data.

For example, any of the following settings for db2.jcc.sendCharInputsUTF8 causes the IBM Data Server Driver for JDBC and SQLJ to convert input character strings to UTF-8, rather than the target encoding, before sending the data to the data source:

- `db2.jcc.sendCharInputsUTF8=yes`
- `db2.jcc.sendCharInputsUTF8=true`
- `db2.jcc.sendCharInputsUTF8=1`

This configuration property applies only to DB2 for z/OS.

**db2.jcc.sqljUncustomizedWarningOrException**

Specifies the action that the IBM Data Server Driver for JDBC and SQLJ takes when an uncustomized SQLJ application runs.

- `db2.jcc.sqljUncustomizedWarningOrException` can have the following values:
  - **0** The IBM Data Server Driver for JDBC and SQLJ does not throw a Warning or Exception when an uncustomized SQLJ application is run. This is the default.
  - **1** The IBM Data Server Driver for JDBC and SQLJ throws a Warning when an uncustomized SQLJ application is run.
  - **2** The IBM Data Server Driver for JDBC and SQLJ throws an Exception when an uncustomized SQLJ application is run.

This configuration property applies only to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows.

**db2.jcc.traceDirectory or db2.jcc.override.traceDirectory**

Enables the IBM Data Server Driver for JDBC and SQLJ trace for Java driver code, and specifies a directory into which trace information is written. When
When db2.jcc.override.traceDirectory is specified, a connection is traced to a file named file-name_origin_n.

- n is the nth connection for a DataSource.
- If neither db2.jcc.traceFileName nor db2.jcc.override.traceFileName is specified, file-name is traceFile. If db2.jcc.traceFileName or db2.jcc.override.traceFileName is also specified, file-name is the value of db2.jcc.traceFileName or db2.jcc.override.traceFileName.

- origin indicates the origin of the log writer that is in use. Possible values of origin are:
  - cpds The log writer for a DB2ConnectionPoolDataSource object.
  - driver The log writer for a DB2Driver object.
  - global The log writer for a DB2TraceManager object.
  - sds The log writer for a DB2SimpleDataSource object.
  - xads The log writer for a DB2XADatasource object.

The db2.jcc.override.traceDirectory property overrides the traceDirectory property for a Connection or DataSource object.

For example, specifying the following setting for db2.jcc.override.traceDirectory enables tracing of the IBM Data Server Driver for JDBC and SQLJ Java code to files in a directory named /SYSTEM/tmp:

db2.jcc.override.traceDirectory=/SYSTEM/tmp

You should set the trace properties under the direction of IBM Software Support.

**db2.jcc.traceLevel or db2.jcc.override.traceLevel**

Specifies what to trace.

The db2.jcc.override.traceLevel property overrides the traceLevel property for a Connection or DataSource object.

You specify one or more trace levels by specifying a decimal value. The trace levels are the same as the trace levels that are defined for the traceLevel property on a Connection or DataSource object.

To specify more than one trace level, do an OR (|) operation on the values, and specify the result in decimal in the db2.jcc.traceLevel or db2.jcc.override.traceLevel specification.

For example, suppose that you want to specify TRACE_DRDA_FLOWS and TRACE_CONNECTIONS for db2.jcc.override.traceLevel.

TRACE_DRDA_FLOWS has a hexadecimal value of X'40'.

TRACE_CONNECTION_CALLS has a hexadecimal value of X'01'. To specify both traces, do a bitwise OR operation on the two values, which results in X'41'. The decimal equivalent is 65, so you specify:

db2.jcc.override.traceLevel=65

**db2.jcc.ssid**

Specifies the DB2 for z/OS subsystem to which applications make connections with IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The db2.jcc.ssid value can be the name of the local DB2 subsystem or a group attachment name.
For example:

db2.jcc.ssid=DB2A

The ssid Connection and DataSource property overrides db2.jcc.ssid.

If you specify a group attachment name, and the DB2 subsystem to which an application is connected fails, the connection terminates. However, when new connections use that group attachment name, DB2 for z/OS uses group attachment processing to find an active DB2 subsystem to which to connect.

If you do not specify the db2.jcc.ssid property, the IBM Data Server Driver for JDBC and SQLJ uses the SSID value from the application defaults load module. When you install DB2 for z/OS, an application defaults load module is created in the prefix.SDSNEXIT data set and the prefix.SDSNLOAD data set. Other application defaults load modules might be created in other data sets for selected applications.

The IBM Data Server Driver for JDBC and SQLJ must load an application defaults load module before it can read the SSID value. z/OS searches data sets in the following places, and in the following order, for the application defaults load module:

1. Job pack area (JPA)
2. TASKLIB
3. STEPLIB or JOBLIB
4. LPA
5. Libraries in the link list

You need to ensure that if your system has more than one copy of the application defaults load module, z/OS finds the data set that contains the correct copy for the IBM Data Server Driver for JDBC and SQLJ first.

This configuration property applies only to DB2 for z/OS.

**db2.jcc.traceFile or db2.jcc.override.traceFile**

Enables the IBM Data Server Driver for JDBC and SQLJ trace for Java driver code, and specifies the name on which the trace file names are based.

Specify a fully qualified z/OS UNIX System Services file name for the db2.jcc.override.traceFile property value.

The db2.jcc.override.traceFile property overrides the traceFile property for a Connection or DataSource object.

For example, specifying the following setting for db2.jcc.override.traceFile enables tracing of the IBM Data Server Driver for JDBC and SQLJ Java code to a file named /SYSTEM/tmp/jdbctrace:

db2.jcc.override.traceFile=/SYSTEM/tmp/jdbctrace

You should set the trace properties under the direction of IBM Software Support.

**db2.jcc.traceFileAppend or db2.jcc.override.traceFileAppend**

Specifies whether to append to or overwrite the file that is specified by the db2.jcc.override.traceFile property. Valid values are true or false. The default is false, which means that the file that is specified by the traceFile property is overwritten.

The db2.jcc.override.traceFileAppend property overrides the traceFileAppend property for a Connection or DataSource object.

For example, specifying the following setting for db2.jcc.override.traceFileAppend causes trace data to be added to the existing trace file:
db2.jcc.override.traceFileAppend=true

You should set the trace properties under the direction of IBM Software Support.

db2.jcc.tracePolling
Indicates whether the IBM Data Server Driver for JDBC and SQLJ polls the global configuration file for changes in trace directives and modifies the trace behavior to match the new trace directives. The driver modifies the trace behavior at the beginning of the next polling interval after the configuration properties file is changed. Possible values are true or false. False is the default. For trace polling to be enabled, the db2.jcc.tracePolling property must be enabled before the driver is loaded and initialized.

db2.jcc.tracePolling polls the following global configuration properties:
- db2.jcc.override.traceLevel
- db2.jcc.override.traceFile
- db2.jcc.override.traceDirectory
- db2.jcc.override.traceFileAppend

db2.jcc.tracePollingInterval
Specifies the interval, in seconds, for polling the IBM Data Server Driver for JDBC and SQLJ global configuration file for changes in trace directives. The property value is a positive integer. The default is 60. For the specified trace polling interval to be used, the db2.jcc.tracePollingInterval property must be set before the driver is loaded and initialized. Changes to db2.jcc.tracePollingInterval after the driver is loaded and initialized have no effect.

db2.jcc.t2zosTraceFile
Enables the IBM Data Server Driver for JDBC and SQLJ trace for C/C++ native driver code for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, and specifies the name on which the trace file names are based. This property is required for collecting trace data for C/C++ native driver code. Specify a fully qualified z/OS UNIX System Services file name for the db2.jcc.t2zosTraceFile property value.

For example, specifying the following setting for db2.jcc.t2zosTraceFile enables tracing of the IBM Data Server Driver for JDBC and SQLJ C/C++ native code to a file named /SYSTEM/tmp/jdbctraceNative:

db2.jcc.t2zosTraceFile=/SYSTEM/tmp/jdbctraceNative

You should set the trace properties under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

db2.jcc.t2zosTraceBufferSize
Specifies the size, in kilobytes, of a trace buffer in virtual storage that is used for tracing the processing that is done by the C/C++ native driver code. This value is also the maximum amount of C/C++ native driver trace information that can be collected.

Specify an integer between 64 (64 KB) and 4096 (4096 KB). The default is 256 (256 KB).

The JDBC driver determines the trace buffer size as shown in the following table:
Specified value \((n)\) & Trace buffer size (KB) \\

\(<64\) & 64 \\
\(64\leq n<128\) & 64 \\
\(128\leq n<256\) & 128 \\
\(256\leq n<512\) & 256 \\
\(512\leq n<1024\) & 512 \\
\(1024\leq n<2048\) & 1024 \\
\(2048\leq n<4096\) & 2048 \\
\(n\geq=4096\) & 4096 \\

\(\text{db2.jcc.t2zosTraceBufferSize}\) is used only if the \(\text{db2.jcc.t2zosTraceFile}\) property is set.

**Recommendation:** To avoid a performance impact, specify a value of 1024 or less.

For example, to set a trace buffer size of 1024 KB, use this setting:

\(\text{db2.jcc.t2zosTraceBufferSize}=1024\)

You should set the trace properties under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

**db2.jcc.t2zosTraceWrap**

Enables or disables wrapping of the SQLJ trace. \(\text{db2.jcc.t2zosTraceWrap}\) can have one of the following values:

- \(1\) Wrap the trace
- \(0\) Do not wrap the trace

The default is 1. This parameter is optional. For example:

\(\text{DB2SQLJ_TRACE_WRAP}=0\)

You should set \(\text{db2.jcc.t2zosTraceWrap}\) only under the direction of IBM Software Support.

This configuration property applies only to DB2 for z/OS.

**db2.jcc.useCcsid420ShapedConverter**

Specifies whether Arabic character data that is in EBCDIC CCSID 420 maps to Cp420S encoding.

\(\text{db2.jcc.useCcsid420ShapedConverter}\) applies only to connections to DB2 for z/OS database servers.

If the value of \(\text{db2.jcc.useCcsid420ShapedConverter}\) is \text{true}, CCSID 420 maps to Cp420S encoding. If the value of \(\text{db2.jcc.useCcsid420ShapedConverter}\) is \text{false}, CCSID 420 maps to Cp420 encoding. \text{false} is the default.

This configuration property applies only to DB2 for z/OS.

**Driver support for JDBC APIs**

The JDBC drivers that are supported by DB2 and IBM Informix (IDS) database systems have different levels of support for JDBC methods.
The following tables list the JDBC interfaces and indicate which drivers support them. The drivers and their supported platforms are:

**Table 55. JDBC drivers for DB2 and IDS database systems**

<table>
<thead>
<tr>
<th>JDBC driver name</th>
<th>Associated data source</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM Data Server Driver for JDBC and SQLJ</td>
<td>DB2 Database for Linux, UNIX, and Windows, DB2 for z/OS, or IBM Informix (IDS)</td>
</tr>
<tr>
<td>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows</td>
<td>DB2 Database for Linux, UNIX, and Windows</td>
</tr>
<tr>
<td>IBM Informix JDBC Driver (IDS JDBC Driver)</td>
<td>IDS</td>
</tr>
</tbody>
</table>

If a method has JDBC 2.0 and JDBC 3.0 forms, the IBM Data Server Driver for JDBC and SQLJ supports all forms. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows supports only the JDBC 2.0 forms.

**Table 56. Support for Array methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getArray</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getBaseType</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getBaseTypeName</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSet</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. Under the IBM Data Server Driver for JDBC and SQLJ, Array methods are supported for connections to DB2 Database for Linux, UNIX, and Windows data sources only.
2. This is a JDBC 4.0 method.

**Table 57. Support for BatchUpdateException methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>java.lang.Exception</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getUpdateCounts</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table 58. Support for Blob methods**

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>length</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>position</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBinaryStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 58. Support for Blob methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBytes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>truncate</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:
1. This is a JDBC 4.0 method.
2. Supported forms of this method include the following JDBC 4.0 form:
   `getBinaryStream(long pos, long length)`
3. For versions of the IBM Data Server Driver for JDBC and SQLJ before version 3.50, these methods cannot be used if a Blob is passed to a stored procedure as an IN or INOUT parameter, and the methods are used on the Blob in the stored procedure.

Table 59. Support for CallableStatement methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.sql.Statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.sql.PreparedStatement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getArray</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getBigDecimal</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBlob</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBoolean</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getByte</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBlob</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBoolean</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getByte</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDouble</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFloat</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getInt</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLong</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getObject</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getRef</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getRowId</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getShort</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDouble</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTimestamp</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getURL</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>registerOutParameter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setAsciiStream</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setBigDecimal</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Chapter 13. JDBC and SQLJ reference information 327
### Table 59. Support for CallableStatement methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>setBinaryStream</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setBoolean</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setByte</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setBytes</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setCharacterStream</td>
<td></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setDate</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setDouble</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setFloat</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setInt</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setLong</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setNull</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setObject</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setShort</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setString</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setTime</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setTimestamp</td>
<td>Yes, No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setURL</td>
<td>Yes, No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>wasNull</td>
<td>Yes, Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. The inherited getParameterMetaData method is not supported if the data source is DB2 for z/OS.
2. This is a JDBC 4.0 method.
3. The following forms of CallableStatement.getMethod methods are not supported if the data source is DB2 for z/OS:
   ```java
   getXXX(String parameterName)
   ```
4. The database server does not timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from the server if you specify a form of the `getDateTime`, `getTimestamp`, or `getTemporal` method that includes a `java.util.Calendar` parameter.
5. The following form of the `getObject` method is not supported:
   ```java
   getobject(int parameterIndex, java.util.Map<key, value>)
   ```
6. The following form of the `registerOutParameter` method is not supported:
   ```java
   registerOutParameter(int parameterIndex, int jdbcType, String typeName)
   ```
7. The following forms of CallableStatement.setXXX methods are not supported if the data source is DB2 for z/OS:
   ```java
   setXXX(String parameterName, int parameterIndex, int jdbcType, String typeName)
   ```
8. The following form of setNull is not supported:
   ```java
   setNull(int parameterIndex, int jdbcType, String typeName)
   ```

### Table 60. Support for Clob methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
### Table 60. Support for Clob methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCharacterStream</td>
<td>Yes[	extsuperscript{2}]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSubString</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>length</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>position</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setAsciiStream[	extsuperscript{1}]</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setCharacterStream</td>
<td>Yes[	extsuperscript{1}]</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>setString[	extsuperscript{1}]</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>truncate[	extsuperscript{1}]</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. This is a JDBC 4.0 method.
2. Supported forms of this method include the following JDBC 4.0 form:
   
   getCharacterStream(long pos, long length)

3. For versions of the IBM Data Server Driver for JDBC and SQLJ before version 3.50, these methods cannot be used if a Clob is passed to a stored procedure as an IN or INOUT parameter, and the methods are used on the Clob in the stored procedure.

### Table 61. Support for Connection methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearWarnings</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>commit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>createStatement[	extsuperscript{1}]</td>
<td>Yes</td>
<td>Yes[	extsuperscript{2}]</td>
<td>Yes</td>
</tr>
<tr>
<td>createBlob[	extsuperscript{2}]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>createClob[	extsuperscript{2}]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getAutoCommit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalog</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getClientInfo[	extsuperscript{4}]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getHoldability</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getMetaData</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTransactionIsolation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTypeMap</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>getWarnings</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isOpen</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isReadonly</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isValid[	extsuperscript{4}]</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>nativeSQL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>prepareCall</td>
<td>Yes[	extsuperscript{3}]</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Table 61. Support for Connection methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>preparedStatement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>releaseSavepoint</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>rollback</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setAutoCommit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setCatalog</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>setClientInfo</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setReadOnly</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setSavepoint</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setTransactionIsolation</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTypeMap</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**
1. This is a JDBC 4.0 method.
2. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 forms of this method.
3. This is a JDBC 4.0 method.
4. Under IBM Data Server Driver for JDBC and SQLJ type 4 connectivity, an SQLException is thrown if the `timeout` parameter value is less than 0. Under IBM Data Server Driver for JDBC and SQLJ type 2 connectivity, an SQLException is thrown if the `timeout` parameter value is not 0.
5. If the stored procedure in the CALL statement is on DB2 for z/OS, the parameters of the CALL statement cannot be expressions.
6. The driver does not use the setting. For the IBM Data Server Driver for JDBC and SQLJ, a connection can be set as read-only through the `readOnly` property for a Connection or DataSource object.

### Table 62. Support for ConnectionEvent methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.util.EventObject</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSQLException</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 63. Support for ConnectionEventListener methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectionClosed</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>connectionErrorOccurred</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 64. Support for ConnectionPoolDataSource methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Table 64. Support for ConnectionPoolDataSource methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPooledConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Note:**
1. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

### Table 65. Support for DatabaseMetaData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>allProceduresAreCallable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>allTablesAreSelectable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>dataDefinitionCausesTransactionCommit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>dataDefinitionIgnoredInTransactions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>deletesAreDetected</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>doesMaxRowSizeIncludeBlobs</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getAttributes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getBestRowIdentifier</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalogs</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalogSeparator</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCatalogTerm</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getClientInfoProperties</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getCatalogTerm</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnPrivileges</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCrossReference</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDatabaseMajorVersion</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getDatabaseMinorVersion</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getDatabaseProductName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDatabaseProductVersion</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDefaultTransactionIsolation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriverMajorVersion</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriverMinorVersion</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriverName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriverVersion</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getExportedKeys</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFunctionColumns</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>JDBC method</td>
<td>IBM Data Server Driver for JDBC and SQLJ support</td>
<td>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</td>
<td>IDS JDBC Driver support</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-------------------------------------------------</td>
<td>----------------------------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>getFunction</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getExtraNameCharacters</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getIdentifierQuoteString</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getImportedKeys</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getIndexInfo</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getJDBCMajorVersion</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getJDBCMinorVersion</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getMaxBinaryLiteralLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxCatalogNameLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxCharLiteralLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnNameLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInGroupBy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInIndex</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInOrderBy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInSelect</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxColumnsInTable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxConnections</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxCursorNameLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxIndexLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxProcedureNameLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxRowSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxSchemaNameLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxStatementLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxStatements</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxTableNameLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxTablesInSelect</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxUserNameLength</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getNumericFunctions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getPrimaryKeys</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getProcedureColumns</td>
<td>Yes on page 336</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getProcedures</td>
<td>Yes on page 336</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getProcedureTerm</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSetHoldability</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getRowIdLifetime</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 65. Support for DatabaseMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver support for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSchemas</td>
<td>Yes 9 on page 336</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSchemaTerm</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSearchStringEscape</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSQLKeywords</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSQLStateType</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getStringFunctions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSuperTables</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getSuperTypes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getSystemFunctions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTablePrivileges</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTables</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTableTypes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDateFunctions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTypeInfo</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getUDTs</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getURL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getUserName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getVersionColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>insertsAreDetected</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isCatalogAtStart</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isReadOnly</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>locatorsUpdateCopy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullPlusNonNullIsNull</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedAtEnd</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedAtStart</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedHigh</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nullsAreSortedLow</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>othersDeletesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>othersInsertsAreVisible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>othersUpdatesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ownDeletesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ownInsertsAreVisible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ownUpdatesAreVisible</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesLowerCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesLowerCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesMixedCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Table 65. Support for DatabaseMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>storesMixedCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesUpperCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>storesUpperCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsAlterTableWithAddColumn</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsAlterTableWithDropColumn</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsANSI92EntryLevelSQL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsANSI92FullSQL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsANSI92IntermediateSQL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsBatchUpdates</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInDataManipulation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInIndexDefinitions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInPrivilegeDefinitions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInProcedureCalls</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCatalogsInTableDefinitions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SupportsColumnAliasing</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsConvert</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCoreSQLGrammar</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsCorrelatedSubqueries</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsDataDefinitionAndDataManipulationTransactions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsDataManipulationTransactionsOnly</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsDifferentTableCorrelationNames</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsExpressionsInOrderBy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsExtendedSQLGrammar</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsFullOuterJoins</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsGetGeneratedKeys</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>supportsGroupBy</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsGroupByBeyondSelect</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsIntegrityEnhancementFacility</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsLikeEscapeClause</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsLimitedOuterJoins</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMinimumSQLGrammar</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMixedCaseIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMixedCaseQuotedIdentifiers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMultipleOpenResults</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsMultipleResultSets</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 65. Support for DatabaseMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>supportsMultipleTransactions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsNamedParameters</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>supportsNonNullColumns</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenCursorsAcrossCommit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenCursorsAcrossRollback</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenStatementsAcrossCommit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOpenStatementsAcrossRollback</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOrderByUnrelated</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsOuterJoins</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsPositionedDelete</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsPositionedUpdate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsResultSetConcurrency</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsResultSetHoldability</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>supportsResultSetType</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSavepoints</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInDataManipulation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInIndexDefinitions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInPrivilegeDefinitions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInProcedureCalls</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSchemasInTableDefinitions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSelectForUpdate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsStoredProcedures</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInComparisons</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInExists</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInIns</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSubqueriesInQuantifieds</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsSuperTables</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>supportsSuperTypes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>supportsTableCorrelationNames</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsTransactionIsolationLevel</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsTransactions</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsUnion</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>supportsUnionAll</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updatesAreDetected</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>usesLocalFilePerTable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>usesLocalFiles</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Table 65. Support for DatabaseMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. DB2 data sources return `false` for this method. IDS data sources return `true`.
2. This method is supported for connections to DB2 Database for Linux, UNIX, and Windows and IDS only.
3. Under the IBM Data Server Driver for JDBC and SQLJ, DB2 data sources and IDS data sources return `true` for this method. Under the IDS JDBC Driver, IDS data sources return `false`.
4. Under the IBM Data Server Driver for JDBC and SQLJ, DB2 data sources and IDS data sources return `false` for this method. Under the IDS JDBC Driver, IDS data sources return `true`.
5. DB2 data sources return `true` for this method. IDS data sources return `false`.
6. This is a JDBC 4.0 method.
7. This method returns the additional column that is described by the JDBC 4.0 specification.
8. JDBC 3.0 and earlier implementations of the IBM Data Server Driver for JDBC and SQLJ return "IBM DB2 JDBC Universal Driver Architecture."
   The JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ returns "IBM Data Server Driver for JDBC and SQLJ."
9. The JDBC 4.0 form and previous forms of this method are supported.
10. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.
11. The method can be executed, but it returns an empty ResultSet.

### Table 66. Support for DataSource methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

### Table 67. Support for DataTruncation methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.sql.SQLException</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Methods inherited from java.sql.SQLWarning</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.
### Table 67. Support for DataTruncation methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDataSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getIndex</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getRead</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTransferSize</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 68. Support for Driver methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>acceptsURL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>connect</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMajorVersion</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMinorVersion</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>get PropertyInfo</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>jdbcCompliant</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 69. Support for DriverManager methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>deregisterDriver</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDriver</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDrivers</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLoginTimeout</td>
<td></td>
<td>Yes(^1)</td>
<td>Yes(^1)</td>
</tr>
<tr>
<td>getLogStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>println</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>registerDriver</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td></td>
<td>Yes(^1)</td>
<td>Yes(^1)</td>
</tr>
<tr>
<td>setLogStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Notes:**

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.
### Table 70. Support for ParameterMetaData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getParameterClassName</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getParameterCount</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getParameterMode</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getParameterType</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getParameterTypeName</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getPrecision</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getScale</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>isNullable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>isSigned</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 71. Support for PooledConnection methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>addConnectionEventListener</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>addStatementEventListener</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>removeConnectionEventListener</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>removeStatementEventListener</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:
1. This is a JDBC 4.0 method.

### Table 72. Support for PreparedStatement methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.sql.Statement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>addBatch</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>clearParameters</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>execute</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeQuery</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeUpdate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMetaData</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getParameterMetaData</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setArray</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBigDecimal</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBinaryStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JDBC method</td>
<td>IBM Data Server Driver for JDBC and SQLJ support</td>
<td>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</td>
<td>IDS JDBC Driver support</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>setBlob</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBoolean</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setByte</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setBytes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setClob</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setDate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setDouble</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFloat</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setInt</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLong</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setNull</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setObject</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setRef</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setRowId</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setShort</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setString</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTime</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTimestamp</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setUnicodeStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setURL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Table 72. Support for PreparedStatement methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
</table>

**Notes:**

1. If the value of the `length` parameter is -1, all of the data from the `InputStream` or `Reader` is read and sent to the data source.
2. Supported forms of this method include the following JDBC 4.0 forms:
   
   - `setAsciiStream(int parameterIndex, InputStream x, long length)`
   - `setAsciiStream(int parameterIndex, InputStream x)`
3. Supported forms of this method include the following JDBC 4.0 forms:
   
   - `setBinaryStream(int parameterIndex, InputStream x, long length)`
   - `setBinaryStream(int parameterIndex, InputStream x)`
4. Supported forms of this method include the following JDBC 4.0 form:
   
   - `setBlob(int parameterIndex, InputStream inputStream, long length)`
5. Supported forms of this method include the following JDBC 4.0 forms:
   
   - `setCharacterStream(int parameterIndex, Reader reader, long length)`
   - `setCharacterStream(int parameterIndex, Reader reader)`
6. Supported forms of this method include the following JDBC 4.0 form:
   
   - `setClob(int parameterIndex, Reader reader, long length)`
7. This is a JDBC 4.0 method.
8. The database server does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone before sending the value to the server if you specify a form of the `setDate`, `setTime`, or `setTimestamp` method that includes a `java.util.Calendar` parameter.
9. The following form of `setNull` is not supported:
   
   - `setNull(int parameterIndex, int jdbcType, String typeName)`
10. `setString` is not supported if the column has the FOR BIT DATA attribute or the data type is BLOB.

### Table 73. Support for Ref methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBaseTypeName</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 74. Support for ResultSet methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>afterLast</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>beforeFirst</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cancelRowUpdates</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>clearWarnings</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>deleteRow</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>findColumn</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>first</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getArray</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 74. Support for ResultSet methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAsciiStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBigDecimal</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBlob</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBoolean</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getByte</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCharacterStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getCursorName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getDouble</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFloat</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getInt</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLong</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMetaData</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getObject</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getRef</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getRow</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getRowId</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getShort</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getStatement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getString</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTime</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTimestamp</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getType</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getUnicodeStream</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getURL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getWarnings</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>insertRow</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>isAfterLast</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isBeforeFirst</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isFirst</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isLast</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>last</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 74. Support for ResultSet methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>moveToCurrentRow</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>moveToInsertRow</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>next</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>previous</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>refreshRow</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>relative</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>rowDeleted</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>rowInserted</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>rowUpdated</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFetchSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>updateArray</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateAsciiStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateBigDecimal</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateBinaryStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateBlob</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateBoolean</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateByte</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateBytes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateCharacterStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateClob</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateDate</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateDouble</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateFloat</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateInt</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateLong</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateNull</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateObject</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateRef</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateRow</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateRowId</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateShort</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateString</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateTime</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>updateTimestamp</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>wasNull</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 74. Support for ResultSet methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JdbcGetBoolean</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetByte</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetBytes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetChar</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetChars</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetDate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetDate2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetDouble</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetDouble2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetFloat</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetFloat2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetInt</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetInt2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetObject</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetObject2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetRowId</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetRowId2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetShort</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetShort2</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetSQLXML</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetSQLXML2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetString</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetString2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetUnicode</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetUnicode2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetURL</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetURL2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetURLType</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JdbcGetURLType2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Notes:
1. `getBinaryStream` is not supported for CLOB columns.
2. `getMetaData` pads the schema name, if the returned schema name is less than 8 characters, to fill 8 characters.
3. The database server does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from the server if you specify a form of the `getDate`, `getTime`, or `getTimestamp` method that includes a `java.util.Calendar` parameter.
4. The following form of the `getObject` method is not supported:
   ```java
   getobject(int parameterIndex, java.util.Map map)
   ```
5. Supported forms of this method include the following JDBC 4.0 forms:
   ```java
   updateAsciiStream(int columnIndex, InputStream x)
   updateAsciiStream(String columnLabel, InputStream x)
   updateAsciiStream(int columnIndex, InputStream x, long length)
   updateAsciiStream(String columnLabel, InputStream x, long length)
   ```
6. Supported forms of this method include the following JDBC 4.0 forms:
   ```java
   updateBinaryStream(int columnIndex, InputStream x)
   updateBinaryStream(String columnLabel, InputStream x)
   updateBinaryStream(int columnIndex, InputStream x, long length)
   updateBinaryStream(String columnLabel, InputStream x, long length)
   ```
7. Supported forms of this method include the following JDBC 4.0 forms:
   ```java
   updateBlob(int columnIndex, InputStream x)
   updateBlob(String columnLabel, InputStream x)
   updateBlob(int columnIndex, InputStream x, long length)
   updateBlob(String columnLabel, InputStream x, long length)
   ```
8. Supported forms of this method include the following JDBC 4.0 forms:
   ```java
   updateCharacterStream(int columnIndex, Reader reader)
   updateCharacterStream(String columnLabel, Reader reader)
   updateCharacterStream(int columnIndex, Reader reader, long length)
   updateCharacterStream(String columnLabel, Reader reader, long length)
   ```
9. Supported forms of this method include the following JDBC 4.0 forms:
   ```java
   updateClob(int columnIndex, Reader reader)
   updateClob(String columnLabel, Reader reader)
   updateClob(int columnIndex, Reader reader, long length)
   updateClob(String columnLabel, Reader reader, long length)
   ```
10. This is a JDBC 4.0 method.

Table 75. Support for ResultSetMetaData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getCatalogName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnClassName</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnCount</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnDisplaySize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnLabel</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnType</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getColumnTypeName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Chapter 13. JDBC and SQLJ reference information 343
Table 75. Support for ResultSetMetaData methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getPrecision</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getScale</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSchemaName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTableName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isAutoIncrement</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isCaseSensitive</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isCurrency</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isDefinitelyWritable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isNullable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isReadOnly</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isSearchable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isSigned</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isWritable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Notes:
1. For IDS data sources, getTableName does not return a value.
2. getSchemaName pads the schema name, if the returned schema name is less than 8 characters, to fill 8 characters.

Table 76. Support for RowId methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getBytes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>hashCode</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>toString</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Notes:
1. These methods are JDBC 4.0 methods.
2. These methods are supported for connections to DB2 for z/OS, DB2 for i, and IDS data sources.

Table 77. Support for SQLClientInfoException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getFailedProperties</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.
### Table 78. Support for SQLData methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getSQLTypeName</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readSQL</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeSQL</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 79. Support for SQLDataException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

### Table 80. Support for SQLException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getSQLState</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getErrorCode</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getNextException</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setNextException</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 81. Support for SQLFeatureNotSupported methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.
### Table 82. Support for SQLInput methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>readArray</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readAsciiStream</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readBigDecimal</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readBinaryStream</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readBlob</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readBoolean</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readByte</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readBytes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readCharacterStream</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readClob</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readDate</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readDouble</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readFloat</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readInt</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readLong</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readObject</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readRef</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readShort</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readString</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readTime</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>readTimestamp</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>wasNull</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 83. Support for SQLIntegrityConstraintViolationException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.
### Table 84. Support for SQLInvalidAuthorizationSpecException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

### Table 85. Support for SQLNonTransientConnectionException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

### Table 86. Support for SQLNonTransientException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

### Table 87. Support for SQLOutput methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeArray</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeAsciiStream</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBigDecimal</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBinaryStream</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBlob</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 87. Support for SQLOutput methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeBoolean</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeByte</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeBytes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeCharacterStream</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeClob</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeDate</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeDouble</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeFloat</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeInt</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeLong</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeObject</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeRef</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeShort</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeString</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeStruct</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeTime</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>writeTimestamp</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 88. Support for SQLRecoverableException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Note:
1. This is a JDBC 4.0 class.

Table 89. Support for SQLSyntaxErrorException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Table 89. Support for SQLSyntaxErrorException methods\(^1\) (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Note:</td>
<td>1. This is a JDBC 4.0 class.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 90. Support for SQLTimeoutException methods\(^1\)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Note:</td>
<td>1. This is a JDBC 4.0 class.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 91. Support for SQLTransientConnectionException methods\(^1\)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Note:</td>
<td>1. This is a JDBC 4.0 class.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 92. Support for SQLTransientException methods\(^1\)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Note:</td>
<td>1. This is a JDBC 4.0 class.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 93. Support for SQLTransientRollbackException methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from java.lang.Exception</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Throwable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Methods inherited from java.lang.Object</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Note:**
1. This is a JDBC 4.0 class.

Table 94. Support for SQLXML methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>free</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getBinaryStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getCharacterStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getSource</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getString</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setBinaryStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setCharacterStream</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setResult</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setString</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**Notes:**
1. These are JDBC 4.0 methods. These methods are not supported for connections to IBM Informix servers.

Table 95. Support for Statement methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>addBatch</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>cancel</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>clearBatch</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>clearWarnings</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>close</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>execute</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeBatch</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeQuery</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>executeUpdate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getFetchSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>JDBC method</td>
<td>IBM Data Server Driver for JDBC and SQLJ support</td>
<td>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</td>
<td>IDS JDBC Driver support</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------</td>
<td>----------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>getGeneratedKeys</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>getMaxFieldSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMaxRows</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getMoreResults</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getQueryTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getResultSet</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setCursorName</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setEscapeProcessing</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFetchDirection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setFetchSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setMaxFieldSize</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setMaxRows</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setPoolable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>setQueryTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Table 95. Support for Statement methods (continued)

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
</table>

Notes:
1. For the IBM Data Server Driver for JDBC and SQLJ, Statement.cancel is supported only in the following environments:
   - Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later
   - Type 2 and type 4 connectivity from a Linux, UNIX, or Windows client to a DB2 for z/OS server, Version 9 or later
   - Type 4 connectivity from a z/OS client to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later
   - Type 4 connectivity from a z/OS client to a DB2 for z/OS server, Version 8 or later
   The action that the IBM Data Server Driver for JDBC and SQLJ takes when the application executes Statement.cancel is also dependent on the setting of the DB2BaseDataSource.interruptProcessingMode property.

2. For the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows, Statement.cancel is supported only in the following environments:
   - Connections to a DB2 Database for Linux, UNIX, and Windows server, Version 8 or later
   - Connections to a DB2 for z/OS server, Version 9 or later

3. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.

4. Not supported for stored procedure ResultSets.

5. For DB2 for i and for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, this method is supported only for a seconds value of 0.

6. For the IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later, Statement.setQueryTimeout is supported for the following methods:
   - Statement.execute
   - Statement.executeUpdate
   - Statement.executeQuery
   Statement.setQueryTimeout is not supported for the Statement.executeBatch method.

7. This is a JDBC 4.0 method.

Table 96. Support for Struct methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
</table>

getSQLTypeName | No | No | No |
getAttributes | No | No | No |

Table 97. Support for Wrapper methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
</table>

isWrapperFor | Yes | No | No |
unWrap | Yes | No | No |

Notes:
1. These are JDBC 4.0 methods.
Table 98. Support for javax.sql.XAConnection methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods inherited from javax.sql.PooledConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getXAResource</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Notes:
1. These methods are supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Table 99. Support for XADataSource methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getXAConnection</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLoginTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setLogWriter</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 100. Support for javax.transaction.xa.XAResource methods

<table>
<thead>
<tr>
<th>JDBC method</th>
<th>IBM Data Server Driver for JDBC and SQLJ support</th>
<th>DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support</th>
<th>IDS JDBC Driver support</th>
</tr>
</thead>
<tbody>
<tr>
<td>commit</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>end</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>forget</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>getTransactionTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isSameRM</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>prepare</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>recover</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>rollback</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>setTransactionTimeout</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>start</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Notes:
1. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server or IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.
2. This method is supported for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows Version 9.1 or later.

IBM Data Server Driver for JDBC and SQLJ support for SQL escape syntax

The IBM Data Server Driver for JDBC and SQLJ supports SQL escape syntax, as described in the JDBC 1.0 specification.
This is the same syntax that is used in vendor escape clauses in ODBC and CLI applications.

SQL escape syntax is supported in JDBC and SQLJ applications.

---

**SQLJ statement reference information**

SQLJ statements are used for transaction control and SQL statement execution.

**SQLJ clause**

The SQL statements in an SQLJ program are in SQLJ clauses.

**Syntax**

```
//sql connection-declaration-clause ;
iterator-declaration-clause
executable-clause
```

**Usage notes**

Keywords in an SQLJ clause are case sensitive, unless those keywords are part of an SQL statement in an executable clause.

**SQLJ host-expression**

A host expression is a Java variable or expression that is referenced by SQLJ clauses in an SQLJ application program.

**Syntax**

```
: simple-variable IN (complex-expression)
IN OUT INOUT
```

**Description**

- Indicates that the variable or expression that follows is a host expression. The colon must immediately precede the variable or expression.

- **IN | OUT | INOUT**
  - For a host expression that is used as a parameter in a stored procedure call, identifies whether the parameter provides data to the stored procedure (IN), retrieves data from the stored procedure (OUT), or does both (INOUT). The default is IN.

- **simple-variable**
  - Specifies a Java unqualified identifier.

- **complex-expression**
  - Specifies a Java expression that results in a single value.

**Usage notes**

- A complex expression must be enclosed in parentheses.
- ANSI/ISO rules govern where a host expression can appear in a static SQL statement.
SQLJ implements-clause

The implements clause derives one or more classes from a Java interface.

Syntax

```
implements <interface-element>
```

**interface-element:**

```
sqlj.runtime.ForUpdate
sqlj.runtime.Scrollable
user-specified-interface-class
```

Description

**interface-element**
 Specifies a user-defined Java interface, the SQLJ interface
sqlj.runtime.ForUpdate or the SQLJ interface sqlj.runtime.Scrollable.

You need to implement sqlj.runtime.ForUpdate when you declare an iterator
for a positioned UPDATE or positioned DELETE operation. See "Perform
positioned UPDATE and DELETE operations in an SQLJ application" for
information on performing a positioned UPDATE or positioned DELETE
operation in SQLJ.

You need to implement sqlj.runtime.Scrollable when you declare a scrollable
iterator. See "Use scrollable iterators in an SQLJ application" for information on
scrollable iterators.

SQLJ with-clause

The with clause specifies a set of one or more attributes for an iterator or a
connection context.

Syntax

```
with (<with-element>)
```

**with-element:**
**Description**

**holdability**
For an iterator, specifies whether an iterator keeps its position in a table after a COMMIT is executed. The value for holdability must be `true` or `false`.

**sensitivity**
For an iterator, specifies whether changes that are made to the underlying table can be visible to the iterator after it is opened. The value must be `INSENSITIVE`, `SENSITIVE`, or `ASENSITIVE`. The default is `ASENSITIVE`.

For connections to IBM Informix (IDS), only `INSENSITIVE` is supported.

**dynamic**
For an iterator that is defined with sensitivity=SENSITIVE, specifies whether the following cases are true:

- When the application executes positioned UPDATE and DELETE statements with the iterator, those changes are visible to the iterator.
- When the application executes INSERT, UPDATE, and DELETE statements within the application but outside the iterator, those changes are visible to the iterator.

The value for dynamic must be `true` or `false`. The default is `false`.

DB2 Database for Linux, UNIX, and Windows servers do not support dynamic scrollable cursors. Specify `true` only if your application accesses data on DB2 for z/OS servers, at Version 9 or later.

For connections to IDS, only `false` is supported. IDS does not support dynamic cursors.

**updateColumns**
For an iterator, specifies the columns that are to be modified when the iterator is used for a positioned UPDATE statement. The value for updateColumns must be a literal string that contains the column names, separated by commas.

**column-name**
For an iterator, specifies a column of the result table that is to be updated using the iterator.

**Java-ID**
For an iterator or connection context, specifies a Java variable that identifies a user-defined attribute of the iterator or connection context. The value of `java-constant-expression` is also user-defined.

**dataSource**
For a connection context, specifies the logical name of a separately-created
DataSource object that represents the data source to which the application will connect. This option is available only for the IBM Data Server Driver for JDBC and SQLJ.

Usage notes

- The value on the left side of a with element must be unique within its with clause.
- If you specify updateColumns in a with element of an iterator declaration clause, the iterator declaration clause must also contain an implements clause that specifies the sqlj.runtime.ForUpdate interface.
- If you do not customize your SQLJ program, the JDBC driver ignores the value of holdability that is in the with clause. Instead, the driver uses the JDBC driver setting for holdability.

SQLJ connection-declaration-clause

The connection declaration clause declares a connection to a data source in an SQLJ application program.

Syntax

```
context
```

Description

Java-modifiers

Specifies modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name

Specifies a valid Java identifier. During the program preparation process, SQLJ generates a connection context class whose name is this identifier.

implements-clause

See “SQLJ implements-clause” for a description of this clause. In a connection declaration clause, the interface class to which the implements clause refers must be a user-defined interface class.

with-clause

See “SQLJ with-clause” for a description of this clause.

Usage notes

- SQLJ generates a connection class declaration for each connection declaration clause you specify. SQLJ data source connections are objects of those generated connection classes.
- You can specify a connection declaration clause anywhere that a Java class definition can appear in a Java program.

SQLJ iterator-declaration-clause

An iterator declaration clause declares a positioned iterator class or a named iterator class in an SQLJ application program.

An iterator contains the result table from a query. SQLJ generates an iterator class for each iterator declaration clause you specify. An iterator is an object of an iterator class.
An iterator declaration clause has a form for a positioned iterator and a form for a named iterator. The two kinds of iterators are distinct and incompatible Java types that are implemented with different interfaces.

**Syntax**

```
Java-modifiers iterator Java-class-name implements-clause with-clause
({\hspace{1ex}positioned-iterator-column-declarations\hspace{1ex}})
\{\hspace{1ex}named-iterator-column-declarations\hspace{1ex}}
```

**Description**

**Java-modifiers**
Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

**Java-class-name**
Any valid Java identifier. During the program preparation process, SQLJ generates an iterator class whose name is this identifier.

**implements-clause**
See "SQLJ implements-clause" for a description of this clause. For an iterator declaration clause that declares an iterator for a positioned UPDATE or positioned DELETE operation, the implements clause must specify interface sqlj.runtime.ForUpdate. For an iterator declaration clause that declares a scrollable iterator, the implements clause must specify interface sqlj.runtime.Scrollable.

**with-clause**
See "SQLJ with-clause" for a description of this clause.

**positioned-iterator-column-declarations**
Specifies a list of Java data types, which are the data types of the columns in the positioned iterator. The data types in the list must be separated by commas. The order of the data types in the positioned iterator declaration is the same as the order of the columns in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See "Java, JDBC, and SQL data types" for a list of compatible data types.
named-iterator-column-declarations

Specifies a list of Java data types and Java identifiers, which are the data types and names of the columns in the named iterator. Pairs of data types and names must be separated by commas. The name of a column in the iterator must match, except for case, the name of a column in the result table. For online checking during serialized profile customization to succeed, the data types of the columns in the iterator must be compatible with the data types of the columns in the result table. See "Java, JDBC, and SQL data types" for a list of compatible data types.

Usage notes

- An iterator declaration clause can appear anywhere in a Java program that a Java class declaration can appear.
- When a named iterator declaration contains more than one pair of Java data types and Java IDs, all Java IDs within the list must be unique. Two Java IDs are not unique if they differ only in case.

SQLJ executable-clause

An executable clause contains an SQL statement or an assignment statement. An assignment statement assigns the result of an SQL operation to a Java variable.

This topic describes the general form of an executable clause.

Syntax

```plaintext
context-clause
statement-clause
assignment-clause
```

Usage notes

- An executable clause can appear anywhere in a Java program that a Java statement can appear.
- SQLJ reports negative SQL codes from executable clauses through class `java.sql.SQLException`.
  
  If SQLJ raises a run-time exception during the execution of an executable clause, the value of any host expression of type OUT or INOUT is undefined.

SQLJ context-clause

A context clause specifies a connection context, an execution context, or both. You use a connection context to connect to a data source. You use an execution context to monitor and modify SQL statement execution.

Syntax

```plaintext
[ connection-context ]
```

Description

`connection-context`

Specifies a valid Java identifier that is declared earlier in the SQLJ program.
That identifier must be declared as an instance of the connection context class that SQLJ generates for a connection declaration clause.

**execution-context**

Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of class `sqlj.runtime.ExecutionContext`.

**Usage notes**

- If you do not specify a connection context in an executable clause, SQLJ uses the default connection context.
- If you do not specify an execution context, SQLJ obtains the execution context from the connection context of the statement.

**SQLJ statement-clause**

A statement clause contains an SQL statement or a SET TRANSACTION clause.

**Syntax**

```plaintext
{ SQL-statement 
  | SET-TRANSACTION-clause }
```

**Description**

**SQL-statement**

You can include SQL statements in [Table 101](#) in a statement clause.

**SET-TRANSACTIONS-clause**

Sets the isolation level for SQL statements in the program and the access mode for the connection. The SET TRANSACTION clause is equivalent to the SET TRANSACTION statement, which is described in the ANSI/ISO SQL standard of 1992 and is supported in some implementations of SQL.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Applicable data sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALTER DATABASE</td>
<td>1 on page 362 2 on page 362</td>
</tr>
<tr>
<td>ALTER FUNCTION</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>ALTER INDEX</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>ALTER PROCEDURE</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>ALTER STOGROUP</td>
<td>1 on page 362 2 on page 362</td>
</tr>
<tr>
<td>ALTER TABLE</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>ALTER TABLESPACE</td>
<td>1 on page 362 2 on page 362</td>
</tr>
<tr>
<td>CALL</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>COMMENT ON</td>
<td>1 on page 362 2 on page 362</td>
</tr>
<tr>
<td>COMMIT</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>Compound SQL (BEGIN ATOMIC...END)</td>
<td>2 on page 362</td>
</tr>
<tr>
<td>CREATE ALIAS</td>
<td>1 on page 362 2 on page 362</td>
</tr>
<tr>
<td>CREATE DATABASE</td>
<td>1 on page 362 2 on page 362 3a on page 362</td>
</tr>
<tr>
<td>CREATE DISTINCTTYPE</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>CREATE FUNCTION</td>
<td>1 on page 362 2 on page 362 3 on page 362</td>
</tr>
<tr>
<td>Statement</td>
<td>Applicable data sources</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>CREATE GLOBAL TEMPORARY TABLE</td>
<td>1 on page 362, 2 on page 362</td>
</tr>
<tr>
<td>CREATE TEMP TABLE</td>
<td>3 on page 362</td>
</tr>
<tr>
<td>CREATE INDEX</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>CREATE PROCEDURE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>CREATE STOGROUP</td>
<td>1 on page 362, 2 on page 362</td>
</tr>
<tr>
<td>CREATE SYNONYM</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>CREATE TABLE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>CREATE TABLESPACE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>CREATE TYPE (cursor)</td>
<td>2 on page 362</td>
</tr>
<tr>
<td>CREATE TRIGGER</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>CREATE VIEW</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DECLARE GLOBAL TEMPORARY TABLE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DELETE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP ALIAS</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP DATABASE</td>
<td>1 on page 362, 2 on page 362, 3a on page 362</td>
</tr>
<tr>
<td>DROP DISTINCT TYPE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP TYPE</td>
<td>3 on page 362</td>
</tr>
<tr>
<td>DROP FUNCTION</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP INDEX</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP PACKAGE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP PROCEDURE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP STOGROUP</td>
<td>1 on page 362, 2 on page 362</td>
</tr>
<tr>
<td>DROP SYNONYM</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP TABLE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP TABLESPACE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP TRIGGER</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>DROP VIEW</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>FETCH</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>GRANT</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>INSERT</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>LOCK TABLE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>MERGE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>REVOKE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>ROLLBACK</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>SAVEPOINT</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>SELECT INTO</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
<tr>
<td>SET CURRENT APPLICATION ENCODING SCHEME</td>
<td>1 on page 362</td>
</tr>
<tr>
<td>SET CURRENT DEBUG MODE</td>
<td>1 on page 362</td>
</tr>
<tr>
<td>SET CURRENT DEFAULT TRANSFORM GROUP</td>
<td>2 on page 362</td>
</tr>
<tr>
<td>SET CURRENT DEGREE</td>
<td>1 on page 362, 2 on page 362, 3 on page 362</td>
</tr>
</tbody>
</table>
Table 101. Valid SQL statements in an SQLJ statement clause (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Applicable data sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET CURRENT EXPLAIN MODE</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT EXPLAIN SNAPSHOT</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT ISOLATION</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT LOCALE LC_TYPE</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT OPTIMIZATION HINT</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT PACKAGE PATH</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT PACKAGESET (USER is not supported)</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT PRECISION</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT QUERY OPTIMIZATION</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT REFRESH AGE</td>
<td>1, 2</td>
</tr>
<tr>
<td>SET CURRENT ROUTINE VERSION</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT RULES</td>
<td>1</td>
</tr>
<tr>
<td>SET CURRENT SCHEMA</td>
<td>2</td>
</tr>
<tr>
<td>SET CURRENT SQLID</td>
<td>1</td>
</tr>
<tr>
<td>SET PATH</td>
<td>1, 2</td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>1</td>
</tr>
<tr>
<td>UPDATE</td>
<td>1, 2, 3</td>
</tr>
</tbody>
</table>

Note: The SQL statement applies to connections to the following data sources:
1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix
   a. IBM Informix, for the SYSMASTER database only.

Usage notes
- SQLJ supports both positioned and searched DELETE and UPDATE operations.
- For a FETCH statement, a positioned DELETE statement, or a positioned UPDATE statement, you must use an iterator to refer to rows in a result table.

SQLJ SET-TRANSACTION-clause

The SET TRANSACTION clause sets the isolation level for the current unit of work.

Syntax

```
SET TRANSACTION ISOLATION LEVEL
  READ COMMITTED
  READ UNCOMMITTED
  REPEATABLE READ
  SERIALIZABLE
```
Description

ISOLATION LEVEL
    Specifies one of the following isolation levels:

    READ COMMITTED
        Specifies that the current DB2 isolation level is cursor stability.

    READ UNCOMMITTED
        Specifies that the current DB2 isolation level is uncommitted read.

    REPEATABLE READ
        Specifies that the current DB2 isolation level is read stability.

    SERIALIZABLE
        Specifies that the current DB2 isolation level is repeatable read.

Usage notes
    You can execute SET TRANSACTION only at the beginning of a transaction.

SQLJ assignment-clause
    The assignment clause assigns the result of an SQL operation to a Java variable.

Syntax
    <Java-ID> = { fullselect | order-by-clause | optimize-for-clause | isolation-clause | queryno-clause | fetch-first-clause | iterator-conversion-clause }

Description
    Java-ID
        Identifies an iterator that was declared previously as an instance of an iterator
        class.

    fullselect
        Generates a result table.

    iterator-conversion-clause
        See "SQLJ iterator-conversion-clause" for a description of this clause.

Usage notes
    • If the object that is identified by Java-ID is a positioned iterator, the number of
      columns in the result set must match the number of columns in the iterator. In
      addition, the data type of each column in the result set must be compatible with
      the data type of the corresponding column in the iterator. See "Java, JDBC, and
      SQL data types" for a list of compatible Java and SQL data types.

    • If the object that is identified by Java-ID is a named iterator, the name of each
      accessor method must match, except for case, the name of a column in the result
      set. In addition, the data type of the object that an accessor method returns must
      be compatible with the data type of the corresponding column in the result set.

    • You can put an assignment clause anywhere in a Java program that a Java
      assignment statement can appear. However, you cannot put an assignment
clause where a Java assignment expression can appear. For example, you cannot
specify an assignment clause in the control list of a for statement.

**SQLJ iterator-conversion-clause**

The iterator conversion clause converts a JDBC ResultSet to an iterator.

**Syntax**

\[ \text{CAST \hspace{1em} host-expression} \]

**Description**

host-expression

Identifies the JDBC ResultSet that is to be converted to an SQLJ iterator.

**Usage notes**

- If the iterator to which the JDBC ResultSet is to be converted is a positioned
  iterator, the number of columns in the ResultSet must match the number of
  columns in the iterator. In addition, the data type of each column in the
  ResultSet must be compatible with the data type of the corresponding column in
  the iterator.
- If the iterator is a named iterator, the name of each accessor method must match,
  except for case, the name of a column in the ResultSet. In addition, the data type
  of the object that an accessor method returns must be compatible with the data
  type of the corresponding column in the ResultSet.
- When an iterator that is generated through the iterator conversion clause is
  closed, the ResultSet from which the iterator is generated is also closed.

**Interfaces and classes in the sqlj.runtime package**

The sqlj.runtime package defines the run-time classes and interfaces that are used
directly or indirectly by the SQLJ programmer.

Classes such as AsciiStream are used directly by the SQLJ programmer. Interfaces
such as ResultSetIterator are implemented as part of generated class declarations.

**sqlj.runtime interfaces**

The following table summarizes the interfaces in sqlj.runtime.

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionContext</td>
<td>Manages the SQL operations that are performed during a connection to a data source.</td>
</tr>
<tr>
<td>ForUpdate</td>
<td>Implemented by iterators that are used in a positioned UPDATE or DELETE statement.</td>
</tr>
<tr>
<td>NamedIterator</td>
<td>Implemented by iterators that are declared as named iterators.</td>
</tr>
<tr>
<td>PositionedIterator</td>
<td>Implemented by iterators that are declared as positioned iterators.</td>
</tr>
<tr>
<td>ResultSetIterator</td>
<td>Implemented by all iterators to allow query results to be processed using a JDBC ResultSet.</td>
</tr>
<tr>
<td>Scrollable</td>
<td>Provides a set of methods for manipulating scrollable iterators.</td>
</tr>
</tbody>
</table>
sqlj.runtime classes

The following table summarizes the classes in sqlj.runtime.

<table>
<thead>
<tr>
<th>Class name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsciiStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as ASCII.</td>
</tr>
<tr>
<td>BinaryStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as binary.</td>
</tr>
<tr>
<td>CharacterStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as Character.</td>
</tr>
<tr>
<td>DefaultRuntime</td>
<td>Implemented by SQLJ to satisfy the expected runtime behavior of SQLJ for most JVM environments. This class is for internal use only and is not described in this documentation.</td>
</tr>
<tr>
<td>ExecutionContext</td>
<td>Implemented when an SQLJ execution context is declared, to control the execution of SQL operations.</td>
</tr>
<tr>
<td>RuntimeContext</td>
<td>Defines system-specific services that are provided by the runtime environment. This class is for internal use only and is not described in this documentation.</td>
</tr>
<tr>
<td>SQLNullException</td>
<td>Derived from the java.sql.SQLException class. An sqlj.runtime.SQLNullException is thrown when an SQL NULL value is fetched into a host identifier with a Java primitive type.</td>
</tr>
<tr>
<td>StreamWrapper</td>
<td>Wraps a java.io.InputStream instance.</td>
</tr>
<tr>
<td>UnicodeStream</td>
<td>A class for handling an input stream whose bytes should be interpreted as Unicode.</td>
</tr>
</tbody>
</table>

sqlj.runtime.ConnectionContext interface

The sqlj.runtime.ConnectionContext interface provides a set of methods that manage SQL operations that are performed during a session with a specific data source.

Translation of an SQLJ connection declaration clause causes SQLJ to create a connection context class. A connection context object maintains a JDBC Connection object on which dynamic SQL operations can be performed. A connection context object also maintains a default ExecutionContext object.

Variables

CLOSE_CONNECTION

Format:
```
public static final boolean CLOSE_CONNECTION=true;
```

A constant that can be passed to the close method. It indicates that the underlying JDBC Connection object should be closed.

KEEP_CONNECTION

Format:
```
public static final boolean KEEP_CONNECTION=false;
```

A constant that can be passed to the close method. It indicates that the underlying JDBC Connection object should not be closed.

Methods

close()

Format:
public abstract void close() throws SQLException

Performs the following functions:
- Releases all resources that are used by the given connection context object
- Closes any open ConnectedProfile objects
- Closes the underlying JDBC Connection object

close() is equivalent to close(CLOSE_CONNECTION).

close(boolean)
Format:
public abstract void close (boolean close-connection) throws SQLException

Performs the following functions:
- Releases all resources that are used by the given connection context object
- Closes any open ConnectedProfile objects
- Closes the underlying JDBC Connection object, depending on the value of the close-connection parameter

Parameters:

close-connection
Specifies whether the underlying JDBC Connection object is closed when a connection context object is closed:

CLOSE_CONNECTION
Closes the underlying JDBC Connection object.

KEEP_CONNECTION
Does not close the underlying JDBC Connection object.

cCONNECTEDProfile
Format:
public abstract ConnectedProfile getConnectedProfile(Object profileKey) throws SQLException

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

cONNECTION
Format:
public abstract Connection getConnection()

Returns the underlying JDBC Connection object for the given connection context object.

cONTEXT
Format:
public abstract ExecutionContext getExecutionContext()

Returns the default ExecutionContext object that is associated with the given connection context object.

isClosed
Format:
public abstract boolean isClosed()

Returns true if the given connection context object has been closed. Returns false if the connection context object has not been closed.
Constructors

The following constructors are defined in a concrete implementation of the
ConnectionContext interface that results from translation of the statement #sql
context Ctx;

**Ctx(String, boolean)**

Format:
```java
public Ctx(String url, boolean autocommit)
    throws SQLException
```

Parameters:
- **url** The representation of a data source, as specified in the JDBC getConnection
  method.
- **autocommit** Whether autocommit is enabled for the connection. A value of true means
  that autocommit is enabled. A value of false means that autocommit is disabled.

**Ctx(String, String, String, boolean)**

Format:
```java
public Ctx(String url, String user, String password,
            boolean autocommit)
    throws SQLException
```

Parameters:
- **url** The representation of a data source, as specified in the JDBC getConnection
  method.
- **user** The user ID under which the connection to the data source is made.
- **password** The password for the user ID under which the connection to the data
  source is made.
- **autocommit** Whether autocommit is enabled for the connection. A value of true means
  that autocommit is enabled. A value of false means that autocommit is disabled.

**Ctx(String, Properties, boolean)**

Format:
```java
public Ctx(String url, Properties info, boolean autocommit)
    throws SQLException
```

Parameters:
- **url** The representation of a data source, as specified in the JDBC getConnection
  method.
- **info** An object that contains a set of driver properties for the connection. Any of
  the IBM Data Server Driver for JDBC and SQLJ properties can be specified.
- **autocommit** Whether autocommit is enabled for the connection. A value of true means
  that autocommit is enabled. A value of false means that autocommit is disabled.
The following constructors are defined in a concrete implementation of the ConnectionContext interface that results from translation of the statement #sql context Ctx with (dataSource ="jdbc/TestDS");

Ctx()
Format:
public Ctx()
throws SQLException

Parameters:

user
The user ID under which the connection to the data source is made.

password
The password for the user ID under which the connection to the data source is made.

Ctx(Connection)
Format:
public Ctx(java.sql.Connection JDBC-connection-object)
throws SQLException

Parameters:

JDBC-connection-object
A previously created JDBC Connection object.

If the constructor call throws an SQLException, the JDBC Connection object remains open.

Ctx(ConnectionContext)
Format:
public Ctx(sqlj.runtime.ConnectionContext SQLJ-connection-context-object)
throws SQLException

Parameters:

SQLJ-connection-context-object
A previously created SQLJ ConnectionContext object.

If the constructor call throws an SQLException, the JDBC Connection object remains open.
**Ctx(ConnectionContext)**

Format:

```java
public Ctx(sqlj.runtime.ConnectionContext SQLJ-connection-context-object)
  throws SQLException
```

Parameters:

*SQLJ-connection-context-object*

A previously created SQLJ ConnectionContext object.

**Methods**

The following additional methods are generated in a concrete implementation of the ConnectionContext interface that results from translation of the statement `
#sql
context Ctx;`

**getDefaultContext**

Format:

```java
public static Ctx getDefaultContext()
```

Returns the default connection context object for the Ctx class.

**getProfileKey**

Format:

```java
public static Object getProfileKey(sqlj.runtime.profile.Loader loader, String profileName) throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

**getProfile**

Format:

```java
public static sqlj.runtime.profile.Profile getProfile(Object key)
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

**getTypeMap**

Format:

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

Returns an instance of a class that implements java.util.Map, which is the user-defined type map that is associated with the ConnectionContext. If there is no associated type map, Java null is returned.

This method is used by code that is generated by the SQLJ translator for executable clauses and iterator declaration clauses, but it can also be invoked in an SQLJ application for direct use in JDBC statements.

**setDefaultContext**

Format:

```java
public static void Ctx setDefaultContext(Ctx default-context)
```

Sets the default connection context object for the Ctx class.

**Recommendation:** Do not use this method for multithreaded applications. Instead, use explicit contexts.
sqlj.runtime.ForUpdate interface

SQLJ implements the sqlj.runtime.ForUpdate interface in SQLJ programs that contain an iterator declaration clause with implements sqlj.runtime.ForUpdate.

An SQLJ program that does positioned UPDATE or DELETE operations (UPDATE...WHERE CURRENT OF or DELETE...WHERE CURRENT OF) must include an iterator declaration clause with implements sqlj.runtime.ForUpdate.

Methods

getCursorName

Format:
public abstract String getCursorName() throws SQLException

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

sqlj.runtime.NamedIterator interface

The sqlj.runtime.NamedIterator interface is implemented when an SQLJ application executes an iterator declaration clause for a named iterator.

A named iterator includes result table column names, and the order of the columns in the iterator is not important.

An implementation of the sqlj.runtime.NamedIterator interface includes an accessor method for each column in the result table. An accessor method returns the data from its column of the result table. The name of an accessor method matches the name of the corresponding column in the named iterator.

Methods (inherited from the ResultSetIterator interface)

close

Format:
public abstract void close() throws SQLException

Releases database resources that the iterator uses.

isClosed

Format:
public abstract boolean isClosed() throws SQLException

Returns a value of true if the close method has been invoked. Returns false if the close method has not been invoked.

next

Format:
public abstract boolean next() throws SQLException

Advances the iterator to the next row. Before an instance of the next method is invoked for the first time, the iterator is positioned before the first row of the result table. next returns a value of true when a next row is available and false when all rows have been retrieved.

sqlj.runtime.PositionedIterator interface

The sqlj.runtime.PositionedIterator interface is implemented when an SQLJ application executes an iterator declaration clause for a positioned iterator.
The order of columns in a positioned iterator must be the same as the order of columns in the result table, and a positioned iterator does not include result table column names.

Methods

sqlj.runtime.PositionedIterator inherits all ResultSetIterator methods, and includes the following additional method:

endFetch
  Format:
  public abstract boolean endFetch() throws SQLException

  Returns a value of true if the iterator is not positioned on a row. Returns a value of false if the iterator is positioned on a row.

sqlj.runtime.ResultSetIterator interface

The sqlj.runtime.ResultSetIterator interface is implemented by SQLJ for all iterator declaration clauses.

An untyped iterator can be generated by declaring an instance of the sqlj.runtime.ResultSetIterator interface directly. In general, use of untyped iterators is not recommended.

Variables

ASENSITIVE
  Format:
  public static final int ASENSITIVE

  A constant that can be returned by the getSensitivity method. It indicates that the iterator is defined as ASENSITIVE.
  This value is not returned by IBM Informix.

FETCH_FORWARD
  Format:
  public static final int FETCH_FORWARD

  A constant that can be used by the following methods:
  • Set by sqlj.runtime.Scrollable.setFetchDirection and sqlj.runtime.ExecutionContext.setFetchDirection
  • Returned by sqlj.runtime.ExecutionContext.getFetchDirection

  It indicates that the iterator fetches rows in a result table in the forward direction, from first to last.

FETCH_REVERSE
  Format:
  public static final int FETCH_REVERSE

  A constant that can be used by the following methods:
  • Set by sqlj.runtime.Scrollable.setFetchDirection and sqlj.runtime.ExecutionContext.setFetchDirection
  • Returned by sqlj.runtime.ExecutionContext.getFetchDirection

  It indicates that the iterator fetches rows in a result table in the backward direction, from last to first.
This value is not returned by IBM Informix.

**FETCH_UNKNOWN**

Format:

```
public static final int FETCH_UNKNOWN
```

A constant that can be used by the following methods:

- Set by sqlj.runtime.Scrollable.setFetchDirection and sqlj.runtime.ExecutionContext.setFetchDirection
- Returned by sqlj.runtime.ExecutionContext.getFetchDirection

It indicates that the iterator fetches rows in a result table in an unknown order.

This value is not returned by IBM Informix.

**INSENSITIVE**

Format:

```
public static final int INSENSITIVE
```

A constant that can be returned by the getSensitivity method. It indicates that the iterator is defined as INSENSITIVE.

**SENSITIVE**

Format:

```
public static final int SENSITIVE
```

A constant that can be returned by the getSensitivity method. It indicates that the iterator is defined as SENSITIVE.

This value is not returned by IBM Informix.

**Methods**

**clearWarnings**

Format:

```
public abstract void clearWarnings() throws SQLException
```

After clearWarnings is called, getWarnings returns null until a new warning is reported for the iterator.

**close**

Format:

```
public abstract void close() throws SQLException
```

Closes the iterator and releases underlying database resources.

**getFetchSize**

Format:

```
synchronized public int getFetchSize() throws SQLException
```

Returns the number of rows that should be fetched by SQLJ when more rows are needed. The returned value is the value that was set by the setFetchSize method, or 0 if no value was set by setFetchSize.

**getResultSet**

Format:

```
public abstract ResultSet getResultSet() throws SQLException
```

Returns the JDBC ResultSet object that is associated with the iterator.
getRow
Format:
synchronized public int getRow() throws SQLException

Returns the current row number. The first row is number 1, the second is number 2, and so on. If the iterator is not positioned on a row, 0 is returned.

getSensitivity
Format:
synchronized public int getSensitivity() throws SQLException

Returns the sensitivity of the iterator. The sensitivity is determined by the sensitivity value that was specified or defaulted in the with clause of the iterator declaration clause.

getWarnings
Format:
public abstract SQLWarning getWarnings() throws SQLException

Returns the first warning that is reported by calls on the iterator. Subsequent iterator warnings are be chained to this SQLWarning. The warning chain is automatically cleared each time the iterator moves to a new row.

isClosed
Format:
public abstract boolean isClosed() throws SQLException

Returns a value of true if the iterator is closed. Returns false otherwise.

next
Format:
public abstract boolean next() throws SQLException

Advances the iterator to the next row. Before next is invoked for the first time, the iterator is positioned before the first row of the result table. next returns a value of true when a next row is available and false when all rows have been retrieved.

setFetchSize
Format:
synchronized public void setFetchSize(int number-of-rows) throws SQLException

Gives SQLJ a hint as to the number of rows that should be fetched when more rows are needed.
Parameters:
    number-of-rows
        The expected number of rows that SQLJ should fetch for the iterator that is associated with the given execution context.

        If number-of-rows is less than 0 or greater than the maximum number of rows that can be fetched, an SQLException is thrown.

sqlj.runtime.Scrollable interface
sqlj.runtime.Scrollable provides methods to move around in the result table and to check the position in the result table.
sqlj.runtime.Scrollable is implemented when a scrollable iterator is declared.

**Methods**

**absolute(int)**

Format:
```java
public abstract boolean absolute (int n) throws SQLException
```

Moves the iterator to a specified row.

If \( n > 0 \), positions the iterator on row \( n \) of the result table. If \( n < 0 \), and \( m \) is the number of rows in the result table, positions the iterator on row \( m + n + 1 \) of the result table.

If the absolute value of \( n \) is greater than the number of rows in the result table, positions the cursor after the last row if \( n \) is positive, or before the first row if \( n \) is negative.

absolute(0) is the same as beforeFirst(). absolute(1) is the same as first().

absolute(-1) is the same as last().

Returns true if the iterator is on a row. Otherwise, returns false.

**afterLast()**

Format:
```java
public abstract void afterLast() throws SQLException
```

Moves the iterator after the last row of the result table.

**beforeFirst()**

Format:
```java
public abstract void beforeFirst() throws SQLException
```

Moves the iterator before the first row of the result table.

**first()**

Format:
```java
public abstract boolean first() throws SQLException
```

Moves the iterator to the first row of the result table.

Returns true if the iterator is on a row. Otherwise, returns false.

**getFetchDirection()**

Format:
```java
public abstract int getFetchDirection() throws SQLException
```

Returns the fetch direction of the iterator. Possible values are:

- `sqlj.runtime.ResultSetIterator.FETCH_FORWARD`
  
  Rows are processed in a forward direction, from first to last.

- `sqlj.runtime.ResultSetIterator.FETCH_REVERSE`
  
  Rows are processed in a backward direction, from last to first.

- `sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN`
  
  The order of processing is not known.

**isAfterLast()**

Format:
```java
public abstract boolean isAfterLast() throws SQLException
```
Returns true if the iterator is positioned after the last row of the result table. Otherwise, returns false.

isBeforeFirst()
Format:
public abstract boolean isBeforeFirst() throws SQLException

Returns true if the iterator is positioned before the first row of the result table. Otherwise, returns false.

isFirst()
Format:
public abstract boolean isFirst() throws SQLException

Returns true if the iterator is positioned on the first row of the result table. Otherwise, returns false.

isLast()
Format:
public abstract boolean isLast() throws SQLException

Returns true if the iterator is positioned on the last row of the result table. Otherwise, returns false.

last()
Format:
public abstract boolean last() throws SQLException

Moves the iterator to the last row of the result table.
Returns true if the iterator is on a row. Otherwise, returns false.

previous()
Format:
public abstract boolean previous() throws SQLException

Moves the iterator to the previous row of the result table.
Returns true if the iterator is on a row. Otherwise, returns false.

relative(int)
Format:
public abstract boolean relative(int n) throws SQLException

If n>0, positions the iterator on the row that is n rows after the current row. If n<0, positions the iterator on the row that is n rows before the current row. If n=0, positions the iterator on the current row.

The cursor must be on a valid row of the result table before you can use this method. If the cursor is before the first row or after the last row, the method throws an SQLException.

Suppose that m is the number of rows in the result table and x is the current row number in the result table. If n>0 and x+n>m, the iterator is positioned after the last row. If n<0 and x+n<1, the iterator is positioned before the first row.

Returns true if the iterator is on a row. Otherwise, returns false.

setFetchDirection(int)
Format:
public abstract void setFetchDirection (int) throws SQLException

Gives the SQLJ runtime environment a hint as to the direction in which rows
of this iterator object are processed. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD  
Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE  
Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN  
The order of processing is not known.

sqlj.runtime.AsiiStream class
The sqlj.runtime.AsiiStream class is for an input stream of ASCII data with a
specified length.

The sqlj.runtime.AsiiStream class is derived from the java.io.InputStream class,
and extends the sqlj.runtime.StreamWrapper class. SQLJ interprets the bytes in an
sqlj.runtime.AsiiStream object as ASCII characters. An InputStream object with
ASCII characters needs to be passed as a sqlj.runtime.AsiiStream object.

Constructors

AsciiStream(InputStream)
Format:
public AsiiStream(java.io.InputStream input-stream)
Creates an ASCII java.io.InputStream object with an unspecified length.
Parameters:
input-stream
The InputStream object that SQLJ interprets as an AsiiStream object.

AsciiStream(InputStream, int)
Format:
public AsiiStream(java.io.InputStream input-stream, int length)
Creates an ASCII java.io.InputStream object with a specified length.
Parameters:
input-stream
The InputStream object that SQLJ interprets as an AsiiStream object.
length
The length of the InputStream object that SQLJ interprets as an AsiiStream
object.

sqlj.runtime.BinaryStream class
The sqlj.runtime.BinaryStream class is for an input stream of binary data with a
specified length.

The sqlj.runtime.BinaryStream class is derived from the java.io.InputStream class,
and extends the sqlj.runtime.StreamWrapper class. SQLJ interprets the bytes in an
sqlj.runtime.BinaryStream object are interpreted as Binary characters. An
InputStream object with Binary characters needs to be passed as a
sqlj.runtime.BinaryStream object.
Constructors

BinaryStream(InputStream)
Format:
public BinaryStream(java.io.InputStream input-stream)
Creates an Binary java.io.InputStream object with an unspecified length.
Parameters:

input-stream
The InputStream object that SQLJ interprets as a BinaryStream object.

BinaryStream(InputStream, int)
Format:
public BinaryStream(java.io.InputStream input-stream, int length)
Creates an Binary java.io.InputStream object with a specified length.
Parameters:

input-stream
The InputStream object that SQLJ interprets as a BinaryStream object.

length
The length of the InputStream object that SQLJ interprets as a
BinaryStream object.

sqlj.runtime.CharacterStream class
The sqlj.runtime.CharacterStream class is for an input stream of character data with
a specified length.

The sqlj.runtime.CharacterStream class is derived from the java.io.Reader class, and
extends the java.io.FilterReader class. SQLJ interprets the bytes in an
sqlj.runtime.CharacterStream object are interpreted as Unicode data. A Reader
object with Unicode data needs to be passed as a sqlj.runtime.CharacterStream
object.

Constructors

CharacterStream(InputStream)
Format:
public CharacterStream(java.io.Reader input-stream)
Creates a character java.io.Reader object with an unspecified length.
Parameters:

input-stream
The Reader object that SQLJ interprets as a CharacterStream object.

CharacterStream(InputStream, int)
Format:
public CharacterStream(java.io.Reader input-stream, int length)
Creates a character java.io.Reader object with a specified length.
Parameters:

input-stream
The Reader object that SQLJ interprets as a CharacterStream object.
length
The length of the Reader object that SQLJ interprets as a CharacterStream object.

Methods

getReader
Format:
public Reader getReader()

Returns the underlying Reader object that is wrapped by the CharacterStream object.

getLength
Format:
public void getLength()

Returns the length in characters of the wrapped Reader object, as specified by the constructor or in the last call to setLength.

setLength
Format:
public void setLength (int length)

Sets the number of characters that are read from the Reader object when the object is passed as an input argument to an SQL operation.

Parameters:

length
The number of characters that are read from the Reader object.

sqlj.runtime.ExecutionContext class

The sqlj.runtime.ExecutionContext class is defined for execution contexts. An execution context is used to control the execution of SQL statements.

Variables

ADD_BATCH_COUNT
Format:
public static final int ADD_BATCH_COUNT

A constant that can be returned by the getUpdateCount method. It indicates that the previous statement was not executed but was added to the existing statement batch.

AUTO_BATCH
Format:
public static final int AUTO_BATCH

A constant that can be passed to the setBatchLimit method. It indicates that implicit batch execution should be performed, and that SQLJ should determine the batch size.

EXEC_BATCH_COUNT
Format:
public static final int EXEC_BATCH_COUNT
A constant that can be returned from the getUpdateCount method. It indicates that a statement batch was just executed.

**EXCEPTION_COUNT**

Format:
```
public static final int EXCEPTION_COUNT
```

A constant that can be returned from the getUpdateCount method. It indicates that an exception was thrown before the previous execution completed, or that no operation has been performed on the execution context object.

**NEW_BATCH_COUNT**

Format:
```
public static final int NEW_BATCH_COUNT
```

A constant that can be returned from the getUpdateCount method. It indicates that the previous statement was not executed, but was added to a new statement batch.

**QUERY_COUNT**

Format:
```
public static final int QUERY_COUNT
```

A constant that can be passed to the setBatchLimit method. It indicates that the previous execution produced a result set.

**UNLIMITED_BATCH**

Format:
```
public static final int UNLIMITED_BATCH
```

A constant that can be returned from the getUpdateCount method. It indicates that statements should continue to be added to a statement batch, regardless of the batch size.

Constructors:

**ExecutionContext**

Format:
```
public ExecutionContext()
```

Creates an ExecutionContext instance.

**Methods**

**cancel**

Format:
```
public void cancel() throws SQLException
```

Cancels an SQL operation that is currently being executed by a thread that uses the execution context object. If there is a pending statement batch on the execution context object, the statement batch is canceled and cleared.

The cancel method throws an SQLException if the statement cannot be canceled.

**execute**

Format:
```
public boolean execute() throws SQLException
```

Chapter 13. JDBC and SQLJ reference information 379
This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

**executeBatch**

Format:
```java
class synchronized public int[] executeBatch() throws SQLException
```

Executes the pending statement batch and returns an array of update counts. If no pending statement batch exists, null is returned. When this method is called, the statement batch is cleared, even if the call results in an exception.

Each element in the returned array can be one of the following values:
- **-2** This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.
- **-3** This value indicates that the SQL statement failed.

*Other integer*

This value is the number of rows that were updated by the statement.

The executeBatch method throws an SQLException if a database error occurs while the statement batch executes.

**executeQuery**

Format:
```java
class public ResultSet executeQuery ( ) throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

**executeUpdate**

Format:
```java
class public int executeUpdate() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

**getBatchLimit**

Format:
```java
synchronized public int getBatchLimit()
```

Returns the number of statements that are added to a batch before the batch is implicitly executed.

The returned value is one of the following values:

**UNLIMITED_BATCH**

This value indicates that the batch size is unlimited.

**AUTO_BATCH**

This value indicates that the batch size is finite but unknown.

*Other integer*

This value indicates the current batch limit.

**getBatchUpdateCounts**

Format:
```java
public synchronized int[] getBatchUpdateCounts()
```

Returns an array that contains the number of rows that were updated by each statement that successfully executed in a batch. The order of elements in the array is as follows:

- **-2** This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.
- **-3** This value indicates that the SQL statement failed.

*Other integer*

This value is the number of rows that were updated by the statement.

The getBatchUpdateCounts method throws an SQLException if a database error occurs while the statement batch executes.
array corresponds to the order in which statements were inserted into the batch. Returns null if no statements in the batch completed successfully.

Each element in the returned array can be one of the following values:

-2 This value indicates that the SQL statement executed successfully, but the number of rows that were updated could not be determined.

-3 This value indicates that the SQL statement failed.

*Other integer*

This value is the number of rows that were updated by the statement.

**getFetchDirection**

Format:

```java
synchronized public int getFetchDirection() throws SQLException
```

Returns the current fetch direction for scrollable iterator objects that were generated from the given execution context. If a fetch direction was not set for the execution context, sqlj.runtime.ResultSetIterator.FETCH_FORWARD is returned.

**getFetchSize**

Format:

```java
synchronized public int getFetchSize() throws SQLException
```

Returns the number of rows that should be fetched by SQLJ when more rows are needed. This value applies only to iterator objects that were generated from the given execution context. The returned value is the value that was set by the setFetchSize method, or 0 if no value was set by setFetchSize.

**getMaxFieldSize**

Format:

```java
public synchronized int getMaxFieldSize()
```

Returns the maximum number of bytes that are returned for any string (character, graphic, or varying-length binary) column in queries that use the given execution context. If this limit is exceeded, SQLJ discards the remaining bytes. A value of 0 means that the maximum number of bytes is unlimited.

**getMaxRows**

Format:

```java
public synchronized int getMaxRows()
```

Returns the maximum number of rows that are returned for any query that uses the given execution context. If this limit is exceeded, SQLJ discards the remaining rows. A value of 0 means that the maximum number of rows is unlimited.

**getNextResultSet()**

Format:

```java
public ResultSet getNextResultSet() throws SQLException
```

After a stored procedure call, returns a result set from the stored procedure. A null value is returned if any of the following conditions are true:

- There are no more result sets to be returned.
- The stored procedure call did not produce any result sets.
- A stored procedure call has not been executed under the execution context.
When you invoke getNextResultSet(), SQLJ closes the currently-open result set and advances to the next result set.

If an error occurs during a call to getNextResultSet, resources for the current JDBC ResultSet object are released, and an SQLException is thrown. Subsequent calls to getNextResultSet return null.

getNextResultSet(int)

Formats:
```
public ResultSet getNextResultSet(int current)
```

After a stored procedure call, returns a result set from the stored procedure.
A null value is returned if any of the following conditions are true:
- There are no more result sets to be returned.
- The stored procedure call did not produce any result sets.
- A stored procedure call has not been executed under the execution context.

If an error occurs during a call to getNextResultSet, resources for the current JDBC ResultSet object are released, and an SQLException is thrown. Subsequent calls to getNextResultSet return null.

Parameters:

```
current
```
Indicates what SQLJ does with the currently open result set before it advances to the next result set:

```
java.sql.Statement.CLOSE_CURRENT_RESULT
    Specifies that the current ResultSet object is closed when the next ResultSet object is returned.

java.sql.Statement.KEEP_CURRENT_RESULT
    Specifies that the current ResultSet object stays open when the next ResultSet object is returned.

java.sql.Statement.CLOSE_ALL_RESULTS
    Specifies that all open ResultSet objects are closed when the next ResultSet object is returned.
```

getQueryTimeout

Format:
```
public synchronized int getQueryTimeout()
```

Returns the maximum number of seconds that SQL operations that use the given execution context object can execute. If an SQL operation exceeds the limit, an SQLException is thrown. The returned value is the value that was set by the setQueryTimeout method, or 0 if no value was set by setQueryTimeout. 0 means that execution time is unlimited.

getUpdateCount

Format:
```
public abstract int getUpdateCount() throws SQLException
```

Returns:

```
ExecutionContext.ADD_BATCH_COUNT
    If the statement was added to an existing batch.

ExecutionContext.NEW_BATCH_COUNT
    If the statement was the first statement in a new batch.
```
**ExecutionContext.EXCEPTION_COUNT**
If the previous statement generated an SQLException, or no previous statement was executed.

**ExecutionContext.EXEC_BATCH_COUNT**
If the statement was part of a batch, and the batch was executed.

**ExecutionContext.QUERY_COUNT**
If the previous statement created an iterator object or JDBC ResultSet.

**Other integer**
If the statement was executed rather than added to a batch. This value is the number of rows that were updated by the statement.

**getWarnings**
Format:
```
public synchronized SQLWarning getWarnings()
```

Returns the first warning that was reported by the last SQL operation that was executed using the given execution context. Subsequent warnings are chained to the first warning. If no warnings occurred, null is returned.

getWarnings is used to retrieve positive SQLCODEs.

**isBatching**
Format:
```
public synchronized boolean isBatching()
```

Returns true if batching is enabled for the execution context. Returns false if batching is disabled.

**registerStatement**
Format:
```
public RTStatement registerStatement(ConnectionContext connCtx, Object profileKey, int stmtNdx)
throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

**releaseStatement**
Format:
```
public void releaseStatement() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

**setBatching**
Format:
```
public synchronized void setBatching(boolean batching)
```

Parameters:

**batching**
Indicates whether batchable statements that are registered with the given execution context can be added to a statement batch:

**true**
Statements can be added to a statement batch.
false
Statements are executed individually.

setBatching affects only statements that occur in the program after setBatching is called. It does not affect previous statements or an existing statement batch.

setBatchLimit
Format:
public synchronized void setBatchLimit(int batch-size)

Sets the maximum number of statements that are added to a batch before the batch is implicitly executed.

Parameters:

batch-size
One of the following values:

ExecutionContext.UNLIMITED_BATCH
Indicates that implicit execution occurs only when SQLJ encounters a statement that is batchable but incompatible, or not batchable. Setting this value is the same as not invoking setBatchLimit.

ExecutionContext.AUTO_BATCH
Indicates that implicit execution occurs when the number of statements in the batch reaches a number that is set by SQLJ.

Positive integer
The number of statements that are added to the batch before SQLJ executes the batch implicitly. The batch might be executed before this many statements have been added if SQLJ encounters a statement that is batchable but incompatible, or not batchable.

setBatchLimit affects only statements that occur in the program after setBatchLimit is called. It does not affect an existing statement batch.

setFetchDirection
Format:
public synchronized void setFetchDirection(int direction) throws SQLException

Gives SQLJ a hint as to the current fetch direction for scrollable iterator objects that were generated from the given execution context.

Parameters:

direction
One of the following values:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD
Rows are fetched in a forward direction. This is the default.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE
Rows are fetched in a backward direction.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
The order of fetching is unknown.

Any other input value results in an SQLException.

setFetchSize
Format:
synchronized public void setFetchSize(int number-of-rows) throws SQLException
Gives SQLJ a hint as to the number of rows that should be fetched when more rows are needed.

Parameters:

\textit{number-of-rows}

The expected number of rows that SQLJ should fetch for the iterator that is associated with the given execution context.

If \textit{number-of-rows} is less than 0 or greater than the maximum number of rows that can be fetched, an SQLException is thrown.

\textbf{setMaxFieldSize}

\textbf{Format:}

\begin{verbatim}
public void setMaxFieldSize(int max-bytes)
\end{verbatim}

Specifies the maximum number of bytes that are returned for any string (character, graphic, or varying-length binary) column in queries that use the given execution context. If this limit is exceeded, SQLJ discards the remaining bytes.

Parameters:

\textit{max-bytes}

The maximum number of bytes that SQLJ should return from a BINARY, VARBINARY, CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC column. A value of 0 means that the number of bytes is unlimited. 0 is the default.

\textbf{setMaxRows}

\textbf{Format:}

\begin{verbatim}
public synchronized void setMaxRows(int max-rows)
\end{verbatim}

Specifies the maximum number of rows that are returned for any query that uses the given execution context. If this limit is exceeded, SQLJ discards the remaining rows.

Parameters:

\textit{max-rows}

The maximum number of rows that SQLJ should return for a query that uses the given execution context. A value of 0 means that the number of rows is unlimited. 0 is the default.

\textbf{setQueryTimeout}

\textbf{Format:}

\begin{verbatim}
public synchronized void setQueryTimeout(int timeout-value)
\end{verbatim}

Specifies the maximum number of seconds that SQL operations that use the given execution context object can execute. If an SQL operation exceeds the limit, an SQLException is thrown.

Parameters:

\textit{timeout-value}

The maximum number of seconds that SQL operations that use the given execution context object can execute. 0 means that execution time is unlimited. 0 is the default.
sqlj.runtime.SQLNullException class

The sqlj.runtime.SQLNullException class is derived from the java.sql.SQLException class.

An sqlj.runtime.SQLNullException is thrown when an SQL NULL value is fetched into a host identifier with a Java primitive type. The SQLSTATE value for an instance of SQLNullException is '22002'.

sqlj.runtime.StreamWrapper class

The sqlj.runtime.StreamWrapper class wraps a java.io.InputStream instance and extends the java.io.InputStream class.


Constructors

StreamWrapper(InputStream)
Format:
protected StreamWrapper(InputStream input-stream)

Creates an sqlj.runtime.StreamWrapper object with an unspecified length.
Parameters:
input-stream
The InputStream object that the sqlj.runtime.StreamWrapper object wraps.

StreamWrapper(InputStream, int)
Format:
protected StreamWrapper(java.io.InputStream input-stream, int length)

Creates an sqlj.runtime.StreamWrapper object with a specified length.
Parameters:
input-stream
The InputStream object that the sqlj.runtime.StreamWrapper object wraps.
length
The length of the InputStream object in bytes.

Methods

getInputStream
Format:
public InputStream getInputStream()

Returns the underlying InputStream object that is wrapped by the StreamWrapper object.

getLength
Format:
public void getLength()
Returns the length in bytes of the wrapped InputStream object, as specified by the constructor or in the last call to setLength.

**setLength**

Format:
```
public void setLength (int length)
```

Sets the number of bytes that are read from the wrapped InputStream object when the object is passed as an input argument to an SQL operation.

Parameters:
- `length`:
The number of bytes that are read from the wrapped InputStream object.

**sqlj.runtime.UnicodeStream class**

The sqlj.runtime.UnicodeStream class is for an input stream of Unicode data with a specified length.

The sqlj.runtime.UnicodeStream class is derived from the java.io.InputStream class, and extends the sqlj.runtime.StreamWrapper class. SQLJ interprets the bytes in a sqlj.runtime.UnicodeStream object as Unicode characters. An InputStream object with Unicode characters needs to be passed as a sqlj.runtime.UnicodeStream object.

**Constructors**

**UnicodeStream(InputStream)**

Format:
```
public UnicodeStream(java.io.InputStream input-stream)
```

Creates a Unicode java.io.InputStream object with an unspecified length.

Parameters:
- `input-stream`:
The InputStream object that SQLJ interprets as an UnicodeStream object.

**UnicodeStream(InputStream, int)**

Format:
```
public UnicodeStream(java.io.InputStream input-stream, int length)
```

Creates a Unicode java.io.InputStream object with a specified length.

Parameters:
- `input-stream`:
The InputStream object that SQLJ interprets as an UnicodeStream object.
- `length`:
The length of the InputStream object that SQLJ interprets as an UnicodeStream object.

---

**IBM Data Server Driver for JDBC and SQLJ extensions to JDBC**

The IBM Data Server Driver for JDBC and SQLJ provides a set of extensions to the support that is provided by the JDBC specification.
To use IBM Data Server Driver for JDBC and SQLJ-only methods in classes that have corresponding, standard classes, cast an instance of the related, standard JDBC class to an instance of the IBM Data Server Driver for JDBC and SQLJ-only class. For example:

```java
class ds = new com.ibm.db2.jcc.DB2SimpleDataSource();
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
```

Table 104 summarizes the IBM Data Server Driver for JDBC and SQLJ-only interfaces.

### Table 104. Summary of IBM Data Server Driver for JDBC and SQLJ-only interfaces provided by the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>Interface name</th>
<th>Applicable data sources</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2Connection</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.Connection interface.</td>
</tr>
<tr>
<td>DB2DatabaseMetaData</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.DatabaseMetaData interface.</td>
</tr>
<tr>
<td>DB2Diagnosable</td>
<td>1, 2, 3</td>
<td>Provides a mechanism for getting DB2 diagnostics from a DB2 SQLException.</td>
</tr>
<tr>
<td>DB2ParameterMetaData</td>
<td>2</td>
<td>Extends the java.sql.ParameterMetaData interface.</td>
</tr>
<tr>
<td>DB2PreparedStatement</td>
<td>1, 2, 3</td>
<td>Extends the com.ibm.db2.jcc.DB2Statement and java.sql.PreparedStatement interfaces.</td>
</tr>
<tr>
<td>DB2ResultSet</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.ResultSet interface.</td>
</tr>
<tr>
<td>DB2RowID</td>
<td>1, 2</td>
<td>Used for declaring Java objects for use with the ROWID data type.</td>
</tr>
<tr>
<td>DB2Statement</td>
<td>1, 2, 3</td>
<td>Extends the java.sql.Statement interface.</td>
</tr>
<tr>
<td>DB2SystemMonitor</td>
<td>1, 2</td>
<td>Used for collecting system monitoring data for a connection.</td>
</tr>
<tr>
<td>DB2TraceManagerMXBean</td>
<td>1, 2</td>
<td>Provides the MBean interface for the remote trace controller.</td>
</tr>
<tr>
<td>DB2Xml</td>
<td>1, 2</td>
<td>Used for updating data in XML columns and retrieving data from XML columns.</td>
</tr>
<tr>
<td>DB2BatchUpdateException</td>
<td>1, 2, 3</td>
<td>Used for retrieving error information about batch execution of statements that return automatically generated keys.</td>
</tr>
</tbody>
</table>

**Note:** The interface applies to connections to the following data sources:
1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix

Table 105 summarizes the IBM Data Server Driver for JDBC and SQLJ-only classes.

### Table 105. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>Class name</th>
<th>Applicable data sources</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2Administrator (DB2 Database for Linux, UNIX, and Windows only)</td>
<td>2 on page 389</td>
<td>Instances of the DB2Administrator class are used to retrieve DB2CataloguedDatabase objects.</td>
</tr>
</tbody>
</table>
Table 105. Summary of IBM Data Server Driver for JDBC and SQLJ-only classes provided by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>Class name</th>
<th>Applicable data sources</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2BaseDataSource</td>
<td>1 2 3</td>
<td>The abstract data source parent class for all IBM Data Server Driver for JDBC and SQLJ-specific implementations of javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, and javax.sql.XADataSource.</td>
</tr>
<tr>
<td>DB2CataloguedDatabase</td>
<td>2</td>
<td>Contains methods that retrieve information about a local DB2 Database for Linux, UNIX, and Windows database.</td>
</tr>
<tr>
<td>DB2ClientRerouteServerList</td>
<td>1 2</td>
<td>Implements the java.io.Serializable and javax.naming.Referenceable interfaces.</td>
</tr>
<tr>
<td>DB2ConnectionPoolDataSource</td>
<td>1 2 3</td>
<td>A factory for PooledConnection objects.</td>
</tr>
<tr>
<td>DB2ExceptionFormatter</td>
<td>1 2 3</td>
<td>Contains methods for printing diagnostic information to a stream.</td>
</tr>
<tr>
<td>DB2JCCPlugin</td>
<td>2</td>
<td>The abstract class for implementation of JDBC security plug-ins.</td>
</tr>
<tr>
<td>DB2PooledConnection</td>
<td>1 2 3</td>
<td>Provides methods that an application server can use to switch users on a preexisting trusted connection.</td>
</tr>
<tr>
<td>DB2PoolMonitor</td>
<td>1 2</td>
<td>Provides methods for monitoring the global transport objects pool for the connection concentrator and Sysplex workload balancing.</td>
</tr>
<tr>
<td>DB2SimpleDataSource</td>
<td>1 2 3</td>
<td>Extends the DataBaseDataSource class. Does not support connection pooling or distributed transactions.</td>
</tr>
<tr>
<td>DB2Sqlca</td>
<td>1 2 3</td>
<td>An encapsulation of the DB2 SQLCA.</td>
</tr>
<tr>
<td>DB2TraceManager</td>
<td>1 2 3</td>
<td>Controls the global log writer.</td>
</tr>
<tr>
<td>DB2Types</td>
<td>1 on page 388</td>
<td>Defines data type constants.</td>
</tr>
<tr>
<td>DB2XDataSource</td>
<td>1 2 3</td>
<td>A factory for XADataSource objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).</td>
</tr>
</tbody>
</table>

Note: The class applies to connections to the following data sources:
1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix

**DBBatchUpdateException interface**

The com.ibm.db2.jcc.DBBatchUpdateException interface is used for retrieving error information about batch execution of statements that return automatically generated keys.

**DBBatchUpdateException methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**getDBGeneratedKeys**

Format:
public java.sql.ResultSet[] getDBGeneratedKeys()
        throws java.sql.SQLException

Retrieves automatically generated keys that were created when INSERT
statements were executed in a batch. Each ResultSet object that is returned
contains the automatically generated keys for a single statement in the batch.
ResultSet objects that are null correspond to failed statements.

DB2Administrator class
Instances of the com.ibm.db2.jcc.DB2Administrator class are used to retrieve
DB2CataloguedDatabase objects. DB2Administrator applies to DB2 Database for
Linux, UNIX, and Windows databases only.

DB2Administrator methods
getInstance
Format:
        public static DB2Administrator getInstance()

Returns an instance of the DB2Administrator class.

getCataloguedDatabases
Format:
        public DB2CataloguedDatabase[] getCataloguedDatabases()
        throws java.sql.SQLException

Retrieves an array that contains a DB2CataloguedDatabase object for each local
database in the local database directory.

If a local DB2 system is available, and the catalog contains no databases, an
array with length zero is returned. If no local DB2 system is available, null is
returned. If the local system is not a DB2 Database for Linux, UNIX, and
Windows system, an SQLException is thrown.

DB2BaseDataSource class
The com.ibm.db2.jcc.DB2BaseDataSource class is the abstract data source parent
class for all IBM Data Server Driver for JDBC and SQLJ-specific implementations
of javax.sql.DataSource, javax.sql.ConnectionPoolDataSource, and
javax.sql.XADataSource.

DB2BaseDataSource implements the java.sql.Wrapper interface.

DB2BaseDataSource properties
The following properties are defined only for the IBM Data Server Driver for JDBC
and SQLJ.

You can set all properties on a DataSource or in the url parameter in a
DriverManager.getConnection call.

All properties except the following properties have a setXXX method to set the
value of the property and a getXXX method to retrieve the value:
• minTransportObjects
• maxTransportObjectIdleTime
• maxTransportObjectWaitTime
• dumpPool
• dumpPoolStatisticsOnSchedule
- `dumpPoolStatisticsOnScheduleFile`

A `setXXX` method has this form:

```java
void setProperty-name(data-type property-value)
```

A `getXXX` method has this form:

```java
data-type getProperty-name()
```

`Property-name` is the unqualified property name. For properties that are not specific to IBM Informix, the first character of the property name is capitalized. For properties that are used only by IBM Informix, all characters of the property name are capitalized.

The following table lists the IBM Data Server Driver for JDBC and SQLJ properties and their data types.

**Table 106. DB2BaseDataSource properties and their data types**

<table>
<thead>
<tr>
<th>Property name</th>
<th>Applicable data sources</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.accountingInterval</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.affinityFailbackInterval</td>
<td>1 on page 396, 3 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.allowNextOnExhaustedResultSet</td>
<td>1 on page 396, 3 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.allowNullResultSetForExecuteQuery</td>
<td>1 on page 396, 3 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.atomicMultiRowInsert</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.blockingReadConnectionTimeout</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.charOutputSize</td>
<td>1 on page 396</td>
<td>short</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientDebugInfo (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientProgramId</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientProgramName (IBM Data Server Driver for JDBC and SQLJ type 4 connectivity)</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternateServerName</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteAlternatePortNumber</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIContext</td>
<td>1 on page 396</td>
<td>javax.naming.Context</td>
</tr>
</tbody>
</table>
### Table 106. DB2BaseDataSource properties and their data types (continued)

<table>
<thead>
<tr>
<th>Property name</th>
<th>Applicable data sources</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIName</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientUser (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.clientWorkstation (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS only)</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.connectionCloseWithInFlightTransaction</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.connectNode</td>
<td>2 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentDegree</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentExplainSnapshot</td>
<td>2 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentLockTimeout</td>
<td>2 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentMaintainedTableTypesForOptimization</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentPackageSet</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentQueryOptimization</td>
<td>2 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentRefreshAge</td>
<td>1 on page 396</td>
<td>long</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentSchema</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.cursorSensitivity</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.databaseName</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dateFormat</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.decimalRoundingMode</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.decimalSeparator</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.decimalStringFormat</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.defaultIsolationLevel</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.deferPrepares</td>
<td>1 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>Property name</td>
<td>Applicable data sources</td>
<td>Data type</td>
</tr>
<tr>
<td>------------------------------------------------------------------------------</td>
<td>-------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.description</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.downgradeHoldCursorsUnderXa</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.driverType</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dumpPool</td>
<td>3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnSchedule</td>
<td>3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.dumpPoolStatisticsOnScheduleFile</td>
<td>3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableClientAffinitiesList</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableNamedParameterMarkers</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableConnectionConcentrator</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableMultiRowInsertSupport</td>
<td>1 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableRowsetSupport</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableSeamlessFailover</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.enableSysplexWLB</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.encryptionAlgorithm</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.fetchSize</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource-floatingPointStringFormat</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeInputStreams</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeLobData</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.gssCredential</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>Object</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.interruptProcessingMode</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.jdbcCollection</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.keepDynamic</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.kerberosServerPrincipal</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.loginTimeout</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>Property name</td>
<td>Applicable data sources</td>
<td>Data type</td>
</tr>
<tr>
<td>------------------------------------------------------------------------------</td>
<td>-------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.logWriter</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>PrintWriter</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.maxRetriesForClientReroute</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.maxRowsetSize</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.maxTransportObjectIdleTime</td>
<td>3 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.maxTransportObjectWaitTime</td>
<td>3 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.maxTransportObjects</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.minTransportObjects</td>
<td>3 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.optimizationProfile</td>
<td>2 on page 396, 3 on page 396, 2 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.optimizationProfileToFlush</td>
<td>2 on page 396, 3 on page 396, 2 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.password</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.pdqProperties</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.pkList</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.planName</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.plugin</td>
<td>2 on page 396, 3 on page 396, 2 on page 396</td>
<td>Object</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.pluginName</td>
<td>2 on page 396, 3 on page 396, 2 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.portNumber</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.progressiveStreaming</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc.DB2BaseDataSource.queryDataSize</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc=DB2BaseDataSource.queryCloseImplicit</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc=DB2BaseDataSource.readOnly</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc=DB2BaseDataSource.reportLongTypes</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>short</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc=DB2BaseDataSource.resultSetHoldability</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc=DB2BaseDataSource.resultSetHoldabilityForCatalogQueries</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc=DB2BaseDataSource.retrieveMessagesFromServerOrGetMessage</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jdbc=DB2BaseDataSource.retryIntervalForClientReroute</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>Property name</td>
<td>Applicable data sources</td>
<td>Data type</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.retryWithAlternativeSecurityMechanism</td>
<td>2 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.returnAlias</td>
<td>1 on page 396, 2 on page 396</td>
<td>short</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sendCharInputsUTF8</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sendDataAsIs</td>
<td>1 on page 396, 2 on page 396, 3 on page 396, 4 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.serverName</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sqljEnableClassLoaderSpecificProfiles</td>
<td>1 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.ssid</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslConnection</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStoreLocation</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sslTrustStorePassword</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.streamBufferSize</td>
<td>1 on page 396, 2 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.stripTrailingZerosForDecimalNumbers</td>
<td>1 on page 396, 2 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.supportsAsynchronousXARollback</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.sysSchema</td>
<td>1 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timeFormat</td>
<td>1 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timestampFormat</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.timestampPrecisionReporting</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceDirectory</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceFile</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceFileAppend</td>
<td>1 on page 396, 2 on page 396, 3 on page 396</td>
<td>boolean</td>
</tr>
<tr>
<td>Property name</td>
<td>Applicable data sources</td>
<td>Data type</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.traceLevel</td>
<td>1, 2, 3</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.useCachedCursor</td>
<td>1</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.useDBC4ColumnNameAndLabelSemantics</td>
<td>1, 2</td>
<td>int</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.user</td>
<td>1, 2, 3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.useRowsetCursor</td>
<td>1</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.useTransactionRedirect</td>
<td>2</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.xaNetworkOptimization</td>
<td>1, 2, 3</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBANSIWARN</td>
<td>3</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBDATE</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBPATH</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBSPACETEMP</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBTEMP</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DBUPSPACE</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.DELIMIDENT</td>
<td>3</td>
<td>boolean</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.IFX_DIRECTIVES</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.IFX_EXTENDED_DIRECTIVES</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.IFX_XASTDCOMPLIANCE_XAEND</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXOPCACHE</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.INFORMIXSTACKSIZE</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.NODEFDAC</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.OPTCOMPIND</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.OPTOFC</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.PDQPRIORITY</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.PSORT_DBTEMP</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.PSORT_NPROCS</td>
<td>3</td>
<td>String</td>
</tr>
<tr>
<td>com.ibm.db2.jcc.DB2BaseDataSource.STMT_CACHE</td>
<td>3</td>
<td>String</td>
</tr>
</tbody>
</table>

Note: The property applies to connections to the following data sources:
1. DB2 for z/OS
2. DB2 Database for Linux, UNIX, and Windows
3. IBM Informix

**DB2BaseDataSource fields**

The following constants are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**public final static int INTERRUPT_PROCESSING_MODE_DISABLED = 0**

A constant for the interruptProcessingMode property. This value indicates that interrupt processing is disabled.

**public final static int INTERRUPT_PROCESSING_MODE_STATEMENT_CANCEL = 1**

A constant for the interruptProcessingMode property. This value indicates that the IBM Data Server Driver for JDBC and SQLJ cancels the currently executing statement when an application executes Statement.cancel, if the data server supports interrupt processing.
public final static int INTERRUPT_PROCESSING_MODE_CLOSE_SOCKET = 2
A constant for the interruptProcessingMode property. This value indicates that
the IBM Data Server Driver for JDBC and SQLJ drops the underlying socket
and closes the connection when an application executes Statement.cancel.

public final static int NO = 2
The NO value for properties.

public final static int NOT_SET = 0
The default value for properties.

public final static int YES = 1
The YES value for properties.

**DB2BaseDataSource methods**

In addition to the getXXX and setXXX methods for the DB2BaseDataSource
properties, the following methods are defined only for the IBM Data Server Driver
for JDBC and SQLJ.

**getReference**

Format:

```java
public javax.naming.Reference getReference()
throws javax.naming.NamingException
```

Retrieves the Reference of a DataSource object. For an explanation of a
Reference, see the description of javax.naming.Referenceable in the JNDI
documentation at:

http://java.sun.com/products/jndi/docs.html

**DB2CallableStatement interface**

The com.ibm.db2.jdbc.DB2CallableStatement interface extends the
java.sql.CallableStatement and the com.ibm.db2.jdbc.DB2PreparedStatement
interfaces.

**DB2CallableStatement methods**

The following methods are defined only for the IBM Data Server Driver for JDBC
and SQLJ.

**getJccArrayAtName**

Format:

```java
public java.sql.Array getJccArrayAtName(String parameterMarkerName)
throws java.sql.SQLException
```

Retrieves an ARRAY value that is designated by a named parameter marker as
a java.sql.Array value.

This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).

Parameters:

- `parameterMarkerName`
  The name of the parameter marker for which a value is retrieved.
public java.math.BigDecimal getJccBigDecimalAtName(String parameterMarkerName)
  throws java.sql.SQLException
public java.math.BigDecimal getJccBigDecimalAtName(String parameterMarkerName, int scale)
  throws java.sql.SQLException

Retrieves a DECIMAL value that is designated by a named parameter marker as a java.math.BigDecimal value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:
  
  parameterMarkerName
    The name of the parameter marker for which a value is retrieved.

  scale
    The scale of the value that is retrieved.

gcJccBlobAtName
  Formats:
public java.sql.Blob getJccBlobAtName(String parameterMarkerName)
  throws java.sql.SQLException

Retrieves a BLOB value that is designated by a named parameter marker as a java.sql.Blob value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:
  
  parameterMarkerName
    The name of the parameter marker for which a value is retrieved.

gcJccBooleanAtName
  Format:
public boolean getJccBooleanAtName(String parameterMarkerName)
  throws java.sql.SQLException

Retrieves a BIT or BOOLEAN value that is designated by a named parameter marker as a boolean value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:
  
  parameterMarkerName
    The name of the parameter marker for which a value is retrieved.

gcJccByteAtName
  Format:
public byte getJccByteAtName(String parameterMarkerName)
  throws java.sql.SQLException

Retrieves a TINYINT value that is designated by a named parameter marker as a byte value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:
**parameterMarkerName**
The name of the parameter marker for which a value is retrieved.

**getJccBytesAtName**
Format:
```java
public byte[] getJccBytesAtName(String parameterMarkerName)
```
```
throws java.sql.SQLException
```

Retrieves a BINARY or VARBINARY value that is designated by a named parameter marker as an array of byte values.

This method can be called only if the enableNamedParameterMarkers property is set to `DB2BaseDataSource.YES` (1).

Parameters:
- **parameterMarkerName**
  The name of the parameter marker for which a value is retrieved.

**getJccClobAtName**
Format:
```java
public java.sql.Blob getJccClobAtName(String parameterMarkerName)
```
```
throws java.sql.SQLException
```

Retrieves a CLOB value that is designated by a named parameter marker as a `java.sql.Clob` value.

This method can be called only if the enableNamedParameterMarkers property is set to `DB2BaseDataSource.YES` (1).

Parameters:
- **parameterMarkerName**
  The name of the parameter marker for which a value is retrieved.

**getJccDateAtName**
Formats:
```java
public java.sql.Date getJccDateAtName(String parameterMarkerName)
```
```
throws java.sql.SQLException
```

Retrieves a DATE value that is designated by a named parameter marker as a `java.sql.Date` value.

This method can be called only if the enableNamedParameterMarkers property is set to `DB2BaseDataSource.YES` (1).

Parameters:
- **parameterMarkerName**
  The name of the parameter marker for which a value is retrieved.
- **cal**
  The `java.util.Calendar` object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the date.

**getJccDoubleAtName**
Format:
```java
public double getJccDoubleAtName(String parameterMarkerName)
```
```
throws java.sql.SQLException
```

Chapter 13. JDBC and SQLJ reference information
Retrieves a DOUBLE value that is designated by a named parameter marker as a double value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

\[ parameterMarkerName \]

The name of the parameter marker for which a value is retrieved.

**getJccFloatAtName**

Format:

```
public double getJccFloatAtName(String parameterMarkerName)
  throws java.sql.SQLException
```

Retrieves a FLOAT value that is designated by a named parameter marker as a double value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

\[ parameterMarkerName \]

The name of the parameter marker for which a value is retrieved.

**getJccIntAtName**

Format:

```
public int getJccIntAtName(String parameterMarkerName)
  throws java.sql.SQLException
```

Retrieves a INTEGER value that is designated by a named parameter marker as an int value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

\[ parameterMarkerName \]

The name of the parameter marker for which a value is retrieved.

**getJccLongAtName**

Format:

```
public long getJccLongAtName(String parameterMarkerName)
  throws java.sql.SQLException
```

Retrieves a BIGINT value that is designated by a named parameter marker as a long value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

\[ parameterMarkerName \]

The name of the parameter marker for which a value is retrieved.

**getJccObjectAtName**

Formats:
public java.sql.Object getJccObjectAtName(String parameterMarkerName)
    throws java.sql.SQLException
public java.sql.Object getJccObjectAtName(String parameterMarkerName, Map map)
    throws java.sql.SQLException

Retrieves a value that is designated by a named parameter marker as a
java.sql.Object value.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:

   parameterMarkerName
   The name of the parameter marker for which a value is retrieved.

   map
   The mapping from SQL type names to Java classes.

getJccRowIdAtName
Format:
public java.sql.RowId getJccRowIdAtName(String parameterMarkerName)
    throws java.sql.SQLException

Retrieves a ROWID value that is designated by a named parameter marker as
a java.sql.RowId value.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
This method requires the IBM Data Server Driver for JDBC and SQLJ Version
4.8 or later.
Parameters:

   parameterMarkerName
   The name of the parameter marker for which a value is retrieved.

getJccShortAtName
Format:
public short getJccShortAtName(String parameterMarkerName)
    throws java.sql.SQLException

Retrieves a SMALLINT value that is designated by a named parameter marker
as a short value.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:

   parameterMarkerName
   The name of the parameter marker for which a value is retrieved.

getJccSQLXMLAtName
Format:
public java.sql.SQLXML getJccSQLXMLAtName(String parameterMarkerName)
    throws java.sql.SQLException

Retrieves a SQLXML value that is designated by a named parameter marker as
a java.sql.SQLXML value.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

This method requires the IBM Data Server Driver for JDBC and SQLJ Version 4.8 or later.

Parameters:

getJccStringAtName

Format:

public java.lang.String getJccStringAtName(String parameterMarkerName)
  throws java.sql.SQLException

Retrieves a CHAR, VARCHAR, or LONGVARCHAR value that is designated by a named parameter marker as a java.lang.String value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

  The name of the parameter marker for which a value is retrieved.

getJccTimeAtName

Formats:

public java.sql.Time getJccTimeAtName(String parameterMarkerName)
  throws java.sql.SQLException

public java.sql.Time getJccTimeAtName(String parameterMarkerName, java.util.Calendar cal)
  throws java.sql.SQLException

Retrieves a TIME value that is designated by a named parameter marker as a java.sql.Time value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

  The name of the parameter marker for which a value is retrieved.

cal  The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the time.

getJccTimestampAtName

Formats:

public java.sql.Timestamp getJccTimestampAtName(String parameterMarkerName)
  throws java.sql.SQLException

public java.sql.Timestamp getJccTimestampAtName(String parameterMarkerName, java.util.Calendar cal)
  throws java.sql.SQLException

Retrieves a TIMESTAMP value that is designated by a named parameter marker as a java.sql.Timestamp value.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:
**parameterMarkerName**  
The name of the parameter marker for which a value is retrieved.

**cal**  
The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the timestamp.

**registerJccOutParameterAtName**  
Formats:

```java
public void registerJccOutParameterAtName(String parameterMarkerName,
                                           int sqlType)
    throws java.sql.SQLException

public void registerJccOutParameterAtName(String parameterMarkerName,
                                           int sqlType,
                                           int scale)
    throws java.sql.SQLException

public void registerJccOutParameterAtName(String parameterMarkerName,
                                           int sqlType,
                                           String typeName)
    throws java.sql.SQLException
```

Registers an OUT parameter that is identified by `parameterMarkerName` as the JDBC type `sqlType`.

This method can be called only if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.YES` (1).

Parameters:

**parameterMarkerName**  
The name of the parameter marker for the parameter that is to be registered.

**sqlType**  
The JDBC type code, as defined in java.sql.Types, of the parameter that is to be registered.

**scale**  
The scale of the parameter that is to be registered. This parameter applies only to this case:
- If `sqlType` is java.sql.Types.DECIMAL or java.sql.Types.NUMERIC, `scale` is the number of digits to the right of the decimal point.

**typeName**  
If `jdbcType` is java.sql.Types.DISTINCT or java.sql.Types.REF, the fully-qualified name of the SQL user-defined type of the parameter that is to be registered.

**setJccXXXAtName** methods  
These methods are inherited from DB2PreparedStatement.

### DB2CataloguedDatabase class

The com.ibm.db2.jcc.DB2CataloguedDatabase class contains methods that retrieve information about a local DB2 Database for Linux, UNIX, and Windows database.

No database connection is needed for calling DB2CataloguedDatabase methods.

**DB2CataloguedDatabase methods**

#### getServerName

Format:

```java
public String getServerName()
```
Retrieves the name of the server on which the database resides.

getPortNumber
Format:
public int getPortNumber()

Retrieves the port number that is associated with the DB2 instance.

getDatabaseName
Format:
public String getDatabaseName()

Retrieves the database name.

getDatabaseAlias
Format:
public String getDatabaseAlias()

Retrieves the database alias.

DB2ClientRerouteServerList class
The com.ibm.db2.jcc.DB2ClientRerouteServerList class implements the java.io.Serializable and javax.naming.Referenceable interfaces.

DB2ClientRerouteServerList methods
getAlternatePortNumber
Format:
public int[] getAlternatePortNumber()

Retrieves the port numbers that are associated with the alternate servers.

getAlternateServerName
Format:
public String[] getAlternateServerName()

Retrieves an array that contains the names of the alternate servers. These values are IP addresses or DNS server names.

getPrimaryPortNumber
Format:
public int getPrimaryPortNumber()

Retrieves the port number that is associated with the primary server.

getPrimaryServerName
Format:
public String[] getPrimaryServerName()

Retrieves the name of the primary server. This value is an IP address or a DNS server name.

setAlternatePortNumber
Format:
public void setAlternatePortNumber(int[] alternatePortNumberList)

Sets the port numbers that are associated with the alternate servers.
**setAlternateServerName**
Format:
```
public void setAlternateServerName(String[] alternateServer)
```
Sets the alternate server names for servers. These values are IP addresses or DNS server names.

**setPrimaryPortNumber**
Format:
```
public void setPrimaryPortNumber(int primaryPortNumber)
```
Sets the port number that is associated with the primary server.

**setPrimaryServerName**
Format:
```
public void setPrimaryServerName(String primaryServer)
```
Sets the primary server name for a server. This value is an IP address or a DNS server name.

### DB2Connection interface

The com.ibm.db2.jcc.DB2Connection interface extends the java.sql.Connection interface.

DB2Connection implements the java.sql.Wrapper interface.

### DB2Connection methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**alternateWasUsedOnConnect**
Format:
```
public boolean alternateWasUsedOnConnect()
```
Throws `java.sql.SQLException`

Returns true if the driver used alternate server information to obtain the connection. The alternate server information is available in the transient `clientRerouteServerList` information on the DB2BaseDataSource, which the database server updates as primary and alternate servers change.

**changeDB2Password**
Format:
```
public abstract void changeDB2Password(String oldPassword,
                                       String newPassword)
```
Throws `java.sql.SQLException`

Changes the password for accessing the data source, for the user of the Connection object.

Parameter descriptions:

- **oldPassword**
  The original password for the Connection.

- **newPassword**
  The new password for the Connection.
createArrayOf
Format:

Array createArrayOf(String typeName,
          Object[] elements)
        throws SQLException;

Creates a java.sql.Array object.
Parameter descriptions:

typeName
   The SQL data type of the elements of the array map to. typeName can be a
   built-in data type or a distinct type.

elements
   The elements that populate the Array object.

deregisterDB2XmlObject
Formats:

public void deregisterDB2XmlObject(String sqlIdSchema,
          String sqlIdName)
        throws SQLException

Removes a previously registered XML schema from the data source.
Parameter descriptions:

sqlIdSchema
   The SQL schema name for the XML schema. sqlIdSchema is a String value
   with a maximum length of 128 bytes. The value of sqlIdSchema must
   conform to the naming rules for any SQL schema name. The name cannot
   begin with the string 'SYS'. If the value of sqlIdSchema is null, the database
   system uses the value in the CURRENT SCHEMA special register.

sqlIdName
   The SQL name for the XML schema. sqlIdName is a String value with a
   maximum length of 128 bytes. The value of sqlIdName must conform to the
   rules for an SQL identifier. If the value of sqlIdSchema is null, the value of
   sqlIdName can be null, In that case, the database system generates the value
   for sqlIdName.

getDB2ClientProgramId
Format:

public String getDB2ClientProgramId()
        throws java.sql.SQLException

Returns the user-defined program identifier for the client. The program
identifier can be used to identify the application at the data source.

getDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and
Windows data servers.

getDB2ClientAccountingInformation
Format:

public String getDB2ClientAccountingInformation()
        throws SQLException

Returns accounting information for the current client.

Important: getDB2ClientAccountingInformation is deprecated in the JDBC 4.0
implementation of the IBM Data Server Driver for JDBC and SQLJ. Use
java.sql.Connection.getClientInfo instead.
getDB2ClientApplicationInformation
Format:
```java
public String getDB2ClientApplicationInformation()
throws java.sql.SQLException
```

Returns application information for the current client.

**Important:** getDB2ClientApplicationInformation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

getDB2ClientUser
Format:
```java
public String getDB2ClientUser()
throws java.sql.SQLException
```

Returns the current client user name for the connection. This name is not the user value for the JDBC connection.

**Important:** getDB2ClientUser is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

getDB2ClientWorkstation
Format:
```java
public String getDB2ClientWorkstation()
throws java.sql.SQLException
```

Returns current client workstation name for the current client.

**Important:** getDB2ClientWorkstation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

getDB2Correlator
Format:
```java
String getDB2Correlator()
throws java.sql.SQLException
```

Returns the value of the crrtkn (correlation token) instance variable that DRDA sends with the ACCRDB command. The correlation token uniquely identifies a logical connection to a server.

getDB2CurrentPackagePath
Format:
```java
public String getDB2CurrentPackagePath()
throws java.sql.SQLException
```

Returns the list of DB2 package collections that are searched for JDBC and SQLJ packages.

The getDB2CurrentPackagePath method applies only to connections to DB2 database systems.

getDB2CurrentPackageSet
Format:
```java
public String getDB2CurrentPackageSet()
throws java.sql.SQLException
```

Returns the collection ID for the connection.
The getDB2CurrentPackageSet method applies only to connections to DB2 database systems.

**getDB2ProgressiveStreaming**
Format:
```java
public int getDB2ProgressiveStreaming()
    throws java.sql.SQLException
```

Returns the current progressive streaming setting for the connection.

The returned value depends on whether the data source supports progressive streaming, how the progressiveStreaming property is set, and whether DB2Connection.setProgressiveStreaming was called:
- If the data source does not support progressive streaming, 2 (NO) is always returned, regardless of the progressiveStreaming property setting.
- If the data source supports progressive streaming, and DB2Connection.setProgressiveStreaming was called, the returned value is the value that DB2Connection.setProgressiveStreaming set.
- If the data source supports progressive streaming, and DB2Connection.setProgressiveStreaming was not called, the returned value is 2 (NO) if progressiveStreaming was set to DB2BaseDataSource.NO. If progressiveStreaming was set to DB2BaseDataSource.YES or was not set, the returned value is 1 (YES).

**getDB2SecurityMechanism**
Format:
```java
public int getDB2SecurityMechanism()
    throws java.sql.SQLException
```

Returns the security mechanism that is in effect for the connection:
- 3 Clear text password security
- 4 User ID-only security
- 7 Encrypted password security
- 9 Encrypted user ID and password security
- 11 Kerberos security
- 12 Encrypted user ID and data security
- 13 Encrypted user ID, password, and data security
- 15 Plugin security
- 16 Encrypted user ID-only security

**getDB2SystemMonitor**
Format:
```java
public abstract DB2SystemMonitor getDB2SystemMonitor()
    throws java.sql.SQLException
```

Returns the system monitor object for the connection. Each IBM Data Server Driver for JDBC and SQLJ connection can have a single system monitor.
Returns the current progressive streaming setting for the connection.

The returned value depends on whether the data source supports progressive streaming, how the progressiveStreaming property is set, and whether DB2Connection.setProgressiveStreaming was called:

- If the data source does not support progressive streaming, 2 (NO) is always returned, regardless of the progressiveStreaming property setting.
- If the data source supports progressive streaming, and DB2Connection.setProgressiveStreaming was called, the returned value is the value that DB2Connection.setProgressiveStreaming set.
- If the data source supports progressive streaming, and DB2Connection.setProgressiveStreaming was not called, the returned value is 2 (NO) if progressiveStreaming was set to DB2BaseDataSource.NO. If progressiveStreaming was set to DB2BaseDataSource.YES or was not set, the returned value is 1 (YES).

getDBStatementConcentrator
Format:
public int getDBStatementConcentrator()
throws java.sql.SQLException

Returns the statement concentrator use setting for the connection. The statement concentrator use setting is set by the setDBStatementConcentrator method or by the statementConcentrator property.

getJccLogWriter
Format:
public PrintWriter getJccLogWriter()
throws java.sql.SQLException

Returns the current trace destination for the IBM Data Server Driver for JDBC and SQLJ trace.

getJccSpecialRegisterProperties
Format:
public java.util.Properties getJccSpecialRegisterProperties()
throws java.sql.SQLException

Returns a java.util.Properties object, in which the keys are the special registers that are supported at the target data source, and the key values are the current values of those special registers.

This method does not apply to connections to IBM Informix data sources.

getSavePointUniqueOption
Format:
public boolean getSavePointUniqueOption()
throws java.sql.SQLException

Returns true if setSavePointUniqueOption was most recently called with a value of true. Returns false otherwise.

installDB2JavaStoredProcedure
Format:
public void DB2Connection.installDB2JavaStoredProcedure(
java.io.InputStream jarFile,
int jarFileLength,
String jarId)
throws java.sql.SQLException
Invokes the sqlj.install_jar stored procedure on a DB2 Database for Linux, UNIX, and Windows server to create a new definition of a JAR file in the catalog for that server.

Parameter descriptions:

*jarFile*
   The contents of the JAR file that is to be defined to the server.

*jarFileLength*
   The length of the JAR file that is to be defined to the server.

*jarId*
   The name of the JAR in the database, in the form `schema.JAR-id` or `JAR-id`.
   This is the name that you use when you refer to the JAR in SQL statements. If you omit `schema`, the database system uses the SQL authorization ID that is in the CURRENT SCHEMA special register. The owner of the JAR is the authorization ID in the CURRENT SQLID special register.

This method does not apply to connections to IBM Informix data sources.

**isDB2Alive**
Format:
```
public boolean DB2Connection.isDB2Alive()
```

*throws java.sql.SQLException*

Returns true if the socket for a connection to the data source is still active.

**Important:** isDB2Alive is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use Connection.isDBValid instead.

**isDBValid**
Format:
```
public boolean DB2Connection.isDBValid(boolean throwException, int timeout)
```

*throws java.sql.SQLException*

Returns true if the connection has not been closed and is still valid. Returns false otherwise.

Parameter descriptions:

*throwException*
   Specifies whether isDBValid throws an SQLException if the connection is not valid. Possible values are:
   - true: isDBValid throws an SQLException if the connection is not valid.
   - false: isDBValid throws an SQLException only if the value of `timeout` is less than 0.

*timeout*
   The time in seconds to wait for a database operation that the driver submits to complete. The driver submits that database operation to the data source to validate the connection. If the timeout period expires before the database operation completes, isDBValid returns false. A value of 0 indicates that there is no timeout period for the database operation.

This method does not apply to connections to IBM Informix data sources.

**prepareDB2OptimisticLockingQuery**
Format:
public java.sql.PreparedStatement
DB2Connection.prepareDB2OptimisticLockingQuery(String sql,
int returnOptimisticLockingColumns)
throws SQLException

Creates a PreparedStatement object that can request optimistic locking
information.

Parameter descriptions:
sql The SQL statement that is to be prepared.

returnOptimisticLockingColumns
Specifies whether optimistic locking columns are returned. Possible values
are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2Statement.RETURN_OPTLOCK_COLUMN_NONE (0)</td>
<td>Do not return optimistic locking columns.</td>
</tr>
<tr>
<td>DB2Statement.RETURN_OPTLOCK_COLUMN_ALWAYS (1)</td>
<td>Add row change columns to the result set even if they do not uniquely represent a single row. This setting is equivalent to the database prepare attribute WITH ROW CHANGE COLUMNS POSSIBLY DISTINCT.</td>
</tr>
<tr>
<td>DB2Statement.RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES (2)</td>
<td>Add row change columns to the result set only if they uniquely represent a single row. This setting is equivalent to the database prepare attribute WITH ROW CHANGE COLUMNS ALWAYS DISTINCT.</td>
</tr>
</tbody>
</table>

reconfigureDB2Connection

Format:
public void reconfigureDB2Connection(java.util.Properties properties)
throws SQLException

Reconfigures a connection with new settings. The connection does not need to
be returned to a connection pool before it is reconfigured. This method can be
called while a transaction is in progress, and can be used for trusted or
untrusted connections.

Trusted connections are supported for:
• IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  – DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  – DB2 for z/OS Version 9.1 or later
  – IBM Informix Version 11.70 or later
• IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for
  z/OS Version 9.1 or later

Parameter descriptions:

properties
New properties for the connection. These properties override any
properties that are already defined on the DB2Connection instance.

registerDB2XmlSchema

Formats:
public void registerDB2XmlSchema(String[] sqlIdSchema,
String[] sqlIdName,
String[] xmlSchemaLocations,
InputStream[] xmlSchemaDocuments,
int[] xmlSchemaDocumentsLengths,
public void registerDB2XmlSchema(String[] sqlIdSchema,
        String[] sqlIdName,
        String[] xmlSchemaLocations,
        String[] xmlSchemaDocuments,
        String xmlSchemaDocumentsProperties,
        String xmlSchemaProperties,
        boolean isUsedForShredding)
        throws SQLException

Registers an XML schema with one or more XML schema documents. If multiple XML schema documents are processed with one call to registerDB2XmlSchema, those documents are processed as part of a single transaction.

The first form of registerDB2XmlSchema is for XML schema documents that are read from an input stream. The second form of registerDB2XmlSchema is for XML schema documents that are read from strings.

Parameter descriptions:

sqlIdSchema
   The SQL schema name for the XML schema. Only the first element of the sqlIdSchema array is used. sqlIdSchema is a String value with a maximum length of 128 bytes. The value of sqlIdSchema must conform to the naming rules for any SQL schema name. The name cannot begin with the string ‘SYS’. If the value of sqlIdSchema is null, the database system uses the value in the CURRENT SCHEMA special register.

sqlIdName
   The SQL name for the XML schema. Only the first element of the sqlIdName array is used. sqlIdName is a String value with a maximum length of 128 bytes. The value of sqlIdName must conform to the rules for an SQL identifier. If the value of sqlIdSchema is null, the value of sqlIdName can be null, In that case, the database system generates the value for sqlIdName.

xmlSchemaLocations
   XML schema locations for the primary XML schema documents of the schemas that are being registered. XML schema location values are normally in URI format. Each xmlSchemaLocations value is a String value with a maximum length of 1000 bytes. The value is used only to match the information that is specified in the XML schema document that references this document. The database system does no validation of the format, and no attempt is made to resolve the URI.

xmlSchemaDocuments
   The content of the primary XML schema documents. Each xmlSchemaDocuments value is a String or InputStream value with a maximum length of 30 MB. The values must not be null.

xmlSchemaDocumentsLengths
   The lengths of the XML schema documents in the xmlSchemaDocuments parameter, if the first form of registerDB2XmlSchema is used. Each xmlSchemaDocumentsLengths value is an int value.
xmlSchemaDocumentsProperties
Contains properties of the primary XML schema documents, such as properties that are used by an external XML schema versioning system. The database system does no validation of the contents of these values. They are stored in the XSR table for retrieval and used in other tools and XML schema repository implementations. Each xmlSchemaDocumentsProperties value is a String or InputStream value with a maximum length of 5 MB. A value is null if there are no properties to be passed.

xmlSchemaDocumentsPropertiesLengths
The lengths of the XML schema properties in the xmlSchemaDocumentsProperties parameter, if the first form of registerDB2XmlSchema is used. Each xmlSchemaDocumentsPropertiesLengths value is an int value.

xmlSchemaProperties
Contains properties of the entire XML schema, such as properties that are used by an external XML schema versioning system. The database system does no validation of the contents of this value. They are stored in the XSR table for retrieval and used in other tools and XML schema repository implementations. The xmlSchemaProperties value is a String or InputStream value with a maximum length of 5 MB. The value is null if there are no properties to be passed.

xmlSchemaPropertiesLengths
The length of the XML schema property in the xmlSchemaProperties parameter, if the first form of registerDB2XmlSchema is used. The xmlSchemaPropertiesLengths value is an int value.

isUsedForShredding
Indicates whether there are annotations in the schema that are to be used for XML decomposition. isUsedForShredding is a boolean value.

This method does not apply to connections to IBM Informix data sources.

setDBProgressiveStreaming
Format:
public void setDBProgressiveStreaming(int newSetting)
throws java.sql.SQLException

Sets the progressive streaming setting for all ResultSet objects that are created on the connection.

Parameter descriptions:

newSetting
The new progresssive streaming setting. Possible values are:

DB2BaseDataSource.YES (1)
Enable progressive streaming. If the data source does not support progressive streaming, this setting has no effect.

DB2BaseDataSource.NO (2)
Disable progressive streaming.

updateDB2XmlSchema
Format:
public void updateDB2XmlSchema(String[] targetSqlIdSchema,
String[] targetSqlIdName,
String[] sourceSqlIdSchema,
String[] sourceSqlIdName,
String[] xmlSchemaLocations,
boolean dropSourceSchema)
throws SQLException

Updates the contents of an XML schema with the contents of another XML
schema in the XML schema repository, and optionally drops the source
schema. The schema documents in the target XML schema are replaced with
the schema documents from the source XML schema. Before
updateDB2XmlSchema can be called, registration of the source and target XML
schemas must be completed.

The SQL ALTERIN privilege is required for updating the target XML schema.
The SQL DROPIN privilege is required for dropping the source XML schema.

Parameter descriptions:

targetSqlIdSchema
  The SQL schema name for a registered XML schema that is to be updated.
targetSqlIdSchema is a String value with a maximum length of 128 bytes.

targetSqlIdName
  The name of the registered XML schema that is to be updated.
targetSqlIdName is a String value with a maximum length of 128 bytes.

sourceSqlIdSchema
  The SQL schema name for a registered XML schema that is used to update
  the target XML schema. sourceSqlIdSchema is a String value with a
  maximum length of 128 bytes.

sourceSqlIdName
  The name of the registered XML schema that is used to update the target
  XML schema. sourceSqlIdName is a String value with a maximum length of
  128 bytes.

dropSourceSchema
  Indicates whether the source XML schema is to be dropped after the target
  XML schema is updated. dropSourceSchema is a boolean value. false is the
  default.

This method does not apply to connections to IBM Informix data sources.

removeDB2JavaStoredProcedure
Format:
public void DB2Connection.removeDB2JavaStoredProcedure(
    String jarId)
throws java.sql.SQLException

Invokes the sqlj.remove_jar stored procedure on a DB2 Database for Linux,
UNIX, and Windows server to delete the definition of a JAR file from the
catalog for that server.

Parameter descriptions:

tjarId
  The name of the JAR in the database, in the form schema.JAR-id or JAR-id.
  This is the name that you use when you refer to the JAR in SQL
  statements. If you omit schema, the database system uses the SQL
  authorization ID that is in the CURRENT SCHEMA special register.

This method does not apply to connections to IBM Informix data sources.
replaceDB2JavaStoredProcedure
Format:
public void DB2Connection.replaceDB2JavaStoredProcedure(
   java.io.InputStream jarFile,
   int jarFileLength,
   String jarId)
   throws java.sql.SQLException

Invokes the sqlj.replace_jar stored procedure on a DB2 Database for Linux,
UNIX, and Windows server to replace the definition of a JAR file in the catalog
for that server.

Parameter descriptions:
jarFile
   The contents of the JAR file that is to be replaced on the server.
jarFileLength
   The length of the JAR file that is to be replace on the server.
jarId
   The name of the JAR in the database, in the form schema.JAR-id or JAR-id.
This is the name that you use when you refer to the JAR in SQL
statements. If you omit schema, the database system uses the SQL
authorization ID that is in the CURRENT SCHEMA special register. The
owner of the JAR is the authorization ID in the CURRENT SQLID special
register.

This method does not apply to connections to IBM Informix data sources.

reuseDB2Connection (trusted connection reuse)
Formats:
public void reuseDB2Connection(byte[] cookie,
   String user,
   String password,
   String usernameRegistry,
   byte[] userSecToken,
   String originalUser,
   java.util.Properties properties)
   throws java.sql.SQLException
public void reuseDB2Connection(byte[] cookie,
   org.ietf.GSSCredential gssCredential,
   String usernameRegistry,
   byte[] userSecToken,
   String originalUser,
   java.util.Properties properties)
   throws java.sql.SQLException

Trusted connections are supported for:
• IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
   – DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
   – DB2 for z/OS Version 9.1 or later
   – IBM Informix Version 11.70 or later
• IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for
  z/OS Version 9.1 or later

The second of these forms of reuseDB2Connection does not apply to IBM Data
Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

These forms of reuseDB2Connection are used by a trusted application server to
reuse a preexisting trusted connection on behalf of a new user. Properties that
can be reset are passed, including the new user ID. The database server resets the associated physical connection. If reuseDB2Connection executes successfully, the connection becomes available for immediate use, with different properties, by the new user.

Parameter descriptions:

cookie
A unique cookie that the JDBC driver generates for the Connection instance. The cookie is known only to the application server and the underlying JDBC driver that established the initial trusted connection. The application server passes the cookie that was created by the driver when the pooled connection instance was created. The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection to ensure that the request originated from the application server that established the trusted physical connection. If the cookies match, the connection becomes available for immediate use, with different properties, by the new user.

user
The client ID that the database system uses to establish the database authorization ID. If the user was not authenticated by the application server, the application server needs to pass a client ID that represents an unauthenticated user.

password
The password for user.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

userNameRegistry
A name that identifies a mapping service that maps a workstation user ID to a z/OS RACF ID. An example of a mapping service is the Integrated Security Services Enterprise Identity Mapping (EIM). The mapping service is defined by a plugin. Valid values for userNameRegistry are defined by the plugin providers. If userNameRegistry is null, no mapping of user is done.

userSecToken
The client's security tokens. This value is traced as part of DB2 for z/OS accounting data. The content of userSecToken is described by the application server and is referred to by the database system as an application server security token.

originalUser
The original user ID that was used by the application server.

properties
Properties for the reused connection.

reuseDB2Connection (untrusted reuse with reauthentication)
Formats:

public void reuseDB2Connection(String user,
        String password,
        java.util.Properties properties)
throws java.sql.SQLException

public void reuseDB2Connection(
        org.ietf.jgss.GSSCredential gssCredential,
        java.util.Properties properties)
throws java.sql.SQLException
The first of these forms of reuseDB2Connection is not supported for IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

The second of these forms of reuseDB2Connection does not apply to IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

In a heterogeneous pooling environment, these forms of reuseDB2Connection reuse an existing Connection instance after reauthentication.

Parameter description:

user
The authorization ID that is used to establish the connection.

password
The password for the authorization ID that is used to establish the connection.

gssCredential
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

properties
Properties for the reused connection. These properties override any properties that are already defined on the DB2Connection instance.

reuseDB2Connection (untrusted or trusted reuse without reauthentication)

Formats:

public void reuseDB2Connection(java.util.Properties properties)
throws java.sql.SQLException

Reuses an existing Connection instance without reauthentication. This method is intended for reuse of a Connection instance when the properties do not change.

Trusted connections are supported for:

• IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  – DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  – DB2 for z/OS Version 9.1 or later
  – IBM Informix Version 11.70 or later
• IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

This method is for dirty reuse of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as user, password, databaseName, serverName, portNumber, planName, and pkList remain unchanged.

Parameter description:

properties
Properties for the reused connection. These properties override any properties that are already defined on the DB2Connection instance.

setDB2ClientAccountingInformation
Format:
public void setDB2ClientAccountingInformation(String info)
    throws java.sql.SQLException

Specifies accounting information for the connection. This information is for
client accounting purposes. This value can change during a connection.

Parameter description:

info
    User-specified accounting information. The maximum length depends on
    the server. For a DB2 Database for Linux, UNIX, and Windows server, the
    maximum length is 255 bytes. For a DB2 for z/OS server, the maximum
    length is 22 bytes. A Java empty string (""") is valid for this parameter
    value, but a Java null value is not valid.

Important: setDB2ClientAccountingInformation is deprecated in the JDBC 4.0
implementation of the IBM Data Server Driver for JDBC and SQLJ. Use
java.sql.Connection.setClientInfo instead.

setDB2ClientApplicationInformation
Format:
public String setDB2ClientApplicationInformation(String info)
    throws java.sql.SQLException

Specifies application information for the current client.

Important: setDB2ClientApplicationInformation is deprecated in the JDBC 4.0
implementation of the IBM Data Server Driver for JDBC and SQLJ. Use
java.sql.Connection.setClientInfo instead.

Parameter description:

info
    User-specified application information. The maximum length depends on
    the server. For a DB2 Database for Linux, UNIX, and Windows server, the
    maximum length is 255 bytes. For a DB2 for z/OS server, the maximum
    length is 32 bytes. A Java empty string (""") is valid for this parameter
    value, but a Java null value is not valid.

setDB2ClientDebugInfo
Formats:
public void setDB2ClientDebugInfo(String debugInfo)
    throws java.sql.SQLException
public void setDB2ClientDebugInfo(String mgrInfo,
        String traceInfo)
    throws java.sql.SQLException

Sets a value for the CLIENT DEBUGINFO connection attribute, to notify the
database system that stored procedures and user-defined functions that are
using the connection are running in debug mode. CLIENT DEBUGINFO is
used by the DB2 Unified Debugger. Use the first form to set the entire CLIENT
DEBUGINFO string. Use the second form to modify only the session manager
and trace information in the CLIENT DEBUGINFO string.

Setting the CLIENT DEBUGINFO attribute to a string of length greater than
zero requires one of the following privileges:
  • The DEBUGSESSION privilege
  • SYSADM authority

Parameter description:
**debugInfo**
A string of up to 254 bytes, in the following form:

```
Mip:port,Iip,Ppid,Ttid,Cid,L lvl
```

The parts of the string are:

- **Mip:port**
  Session manager IP address and port number
- **Iip**
  Client IP address
- **Ppid**
  Client process ID
- **Ttid**
  Client thread ID (optional)
- **Cid**
  Data connection generated ID
- **Llvl**
  Debug library diagnostic trace level

For example:

```
M9.72.133.89:8355,I9.72.133.89,P4552,T123,C1,L0
```

See the description of SET CLIENT DEBUGINFO for a detailed description of this string.

**mgrInfo**
A string of the following form, which specifies the IP address and port number for the Unified Debugger session manager.

```
Mip:port
```

For example:

```
M9.72.133.89:8355
```

See the description of SET CLIENT DEBUGINFO for a detailed description of this string.

**trcInfo**
A string of the following form, which specifies the debug library diagnostics trace level.

```
L lvl
```

For example:

```
L0
```

See the description of SET CLIENT DEBUGINFO for a detailed description of this string.

**setDB2ClientProgramId**

Format:

```
public abstract void setDB2ClientProgramId(String program-ID)
throws java.sql.SQLException
```

Sets a user-defined program identifier for the connection, on DB2 for z/OS servers. That program identifier is an 80-byte string that is used to identify the caller.

setDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows or IBM Informix data servers.
The DB2 for z/OS server places the string in IFCID 316 trace records along with other statistics, so that you can identify which program is associated with a particular SQL statement.

**setDB2ClientUser**

**Format:**

```java
public void setDB2ClientUser(String user)
    throws java.sql.SQLException
```

Specifies the current client user name for the connection. This name is for client accounting purposes, and is not the user value for the JDBC connection. Unlike the user for the JDBC connection, the current client user name can change during a connection.

**Parameter description:**

- **user**
  - The user ID for the current client. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 16 bytes. A Java empty string ("") is valid for this parameter value, but a Java null value is not valid.

**Important:** setDB2ClientUser is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.setClientInfo instead.

**setDB2ClientWorkstation**

**Format:**

```java
public void setDB2ClientWorkstation(String name)
    throws java.sql.SQLException
```

Specifies the current client workstation name for the connection. This name is for client accounting purposes. The current client workstation name can change during a connection.

**Parameter description:**

- **name**
  - The workstation name for the current client. The maximum length depends on the server. For a DB2 Database for Linux, UNIX, and Windows server, the maximum length is 255 bytes. For a DB2 for z/OS server, the maximum length is 18 bytes. A Java empty string ("") is valid for this parameter value, but a Java null value is not valid.

**Important:** setDB2ClientWorkstation is deprecated in the JDBC 4.0 implementation of the IBM Data Server Driver for JDBC and SQLJ. Use java.sql.Connection.getClientInfo instead.

**setDB2CurrentPackagePath**

**Format:**

```java
public void setDB2CurrentPackagePath(String packagePath)
    throws java.sql.SQLException
```

Specifies a list of collection IDs that the database system searches for JDBC and SQLJ packages.

The setDB2CurrentPackagePath method applies only to connections to DB2 database systems.

**Parameter description:**
packagePath
   A comma-separated list of collection IDs.

setDB2CurrentPackageSet
   Format:
   public void setDB2CurrentPackageSet(String packageSet)
   throws java.sql.SQLException

   Specifies the collection ID for the connection. When you set this value, you
   also set the collection ID of the IBM Data Server Driver for JDBC and SQLJ
   instance that is used for the connection.

   The setDB2CurrentPackageSet method applies only to connections to DB2
   database systems.

   Parameter description:

   packageSet
      The collection ID for the connection. The maximum length for the
      packageSet value is 18 bytes. You can invoke this method as an alternative
      to executing the SQL SET CURRENT PACKAGESET statement in your
      program.

setDB2ProgressiveStreaming
   Format:
   public void setDB2ProgressiveStreaming(int newSetting)
   throws java.sql.SQLException

   Sets the progressive streaming setting for all ResultSet objects that are created
   on the connection.

   Parameter descriptions:

   newSetting
      The new progressive streaming setting. Possible values are:

      DB2BaseDataSource.YES (1)
         Enable progressive streaming. If the data source does not support
         progressive streaming, this setting has no effect.

      DB2BaseDataSource.NO (2)
         Disable progressive streaming.

setJccLogWriter
   Formats:
   public void setJccLogWriter(PrintWriter logWriter)
   throws java.sql.SQLException

   public void setJccLogWriter(PrintWriter logWriter, int traceLevel)
   throws java.sql.SQLException

   Enables or disables the IBM Data Server Driver for JDBC and SQLJ trace, or
   changes the trace destination during an active connection.

   Parameter descriptions:

   logWriter
      An object of type java.io.PrintWriter to which the IBM Data Server Driver
      for JDBC and SQLJ writes trace output. To turn off the trace, set the value
      of logWriter to null.
traceLevel

Specifies the types of traces to collect. See the description of the traceLevel property in "Properties for the IBM Data Server Driver for JDBC and SQLJ" for valid values.

setSavePointUniqueOption

Format:

```java
public void setSavePointUniqueOption(boolean flag)
throws java.sql.SQLException
```

Specifies whether an application can reuse a savepoint name within a unit of recovery. Possible values are:

- **true** A Connection.setSavepoint(savepoint-name) method cannot specify the same value for savepoint-name more than once within the same unit of recovery.

- **false** A Connection.setSavepoint(savepoint-name) method can specify the same value for savepoint-name more than once within the same unit of recovery.

When false is specified, if the Connection.setSavepoint(savepoint-name) method is executed, and a savepoint with the name savepoint-name already exists within the unit of recovery, the database manager destroys the existing savepoint, and creates a new savepoint with the name savepoint-name.

Reuse of a savepoint is not the same as executing Connection.releaseSavepoint(savepoint-name). Connection.releaseSavepoint(savepoint-name) releases savepoint-name, and any savepoints that were subsequently set.

**DB2ConnectionPoolDataSource class**

DB2ConnectionPoolDataSource is a factory for PooledConnection objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).

The com.ibm.db2.jcc.DB2ConnectionPoolDataSource class extends the com.ibm.db2.jcc.DB2BaseDataSource class, and implements the javax.sql.ConnectionPoolDataSource, java.io.Serializable, and javax.naming.Referenceable interfaces.

**DB2ConnectionPoolDataSource properties**

These properties are defined only for the IBM Data Server Driver for JDBC and SQLJ. "Properties for the IBM Data Server Driver for JDBC and SQLJ" for explanations of these properties.

These properties have a setXXX method to set the value of the property and a getXXX method to retrieve the value. A setXXX method has this form:

```java
void setProperty-name(data-type property-value)
```

A getXXX method has this form:

```java
data-type getProperty-name()
```

Property-name is the unqualified property name, with the first character capitalized.
The following table lists the IBM Data Server Driver for JDBC and SQLJ properties and their data types.

### Table 108. DB2ConnectionPoolDataSource properties and their data types

<table>
<thead>
<tr>
<th>Property name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.ibm.db2.jcc.DB2ConnectionPoolDataSource.maxStatements</td>
<td>int</td>
</tr>
</tbody>
</table>

**DB2ConnectionPoolDataSource methods**

**getDB2PooledConnection**

Formats:

```java
class DB2PooledConnection {  
    public DB2PooledConnection getDB2PooledConnection(String user,  
            String password,  
            java.util.Properties properties)  
    throws java.sql.SQLException  
    public DB2PooledConnection getDB2PooledConnection(  
            org.ietf.jgss.GSSCredential gssCredential,  
            java.util.Properties properties)  
    throws java.sql.SQLException  
}
```

Establishes the initial untrusted connection in a heterogeneous pooling environment.

The first form `getDB2PooledConnection` provides a user ID and password. The second form of `getDB2PooledConnection` is for connections that use Kerberos security.

Parameter descriptions:

- **user**  
  The authorization ID that is used to establish the connection.

- **password**  
  The password for the authorization ID that is used to establish the connection.

- **gssCredential**  
  If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

- **properties**  
  Properties for the connection.

**getDB2TrustedPooledConnection**

Formats:

```java
class DB2TrustedPooledConnection {  
    public Object[] getDB2TrustedPooledConnection(String user,  
            String password,  
            java.util.Properties properties)  
    throws java.sql.SQLException  
    public Object[] getDB2TrustedPooledConnection(  
            java.util.Properties properties)  
    throws java.sql.SQLException  
    public Object[] getDB2TrustedPooledConnection(  
            org.ietf.jgss.GSSCredential gssCredential,  
            java.util.Properties properties)  
    throws java.sql.SQLException  
}
```

An application server using a system authorization ID uses this method to establish a trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
- DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
- DB2 for z/OS Version 9.1 or later
- IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

The following elements are returned in Object[]:
- The first element is a trusted DB2PooledConnection instance.
- The second element is a unique cookie for the generated pooled connection instance.

The first form getDB2TrustedPooledConnection provides a user ID and password, while the second form of getDB2TrustedPooledConnection uses the user ID and password of the DB2ConnectionPoolDataSource object. The third form of getDB2TrustedPooledConnection is for connections that use Kerberos security.

Parameter descriptions:

**user**
- The DB2 authorization ID that is used to establish the trusted connection to the database server.

**password**
- The password for the authorization ID that is used to establish the trusted connection.

**gssCredential**
- If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

**properties**
- Properties for the connection.

### DB2DatabaseMetaData interface

The com.ibm.db2.jcc.DB2DatabaseMetaData interface extends the java.sql.DatabaseMetaData interface.

### DB2DatabaseMetaData methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

#### isIDSDatabaseAnsiCompliant

Format:
```java
public boolean isIDSDatabaseAnsiCompliant();
```

Returns true if the current active IBM Informix (IDS) database is ANSI-compliant. Returns false otherwise.

An ANSI-compliant database is a database that was created with the WITH LOG MODE ANSI option.

This method applies to connections to IDS data sources only. An SQLException is thrown if the data source is not an IDS data source.

#### isIDSDatabaseLogging

Format:
```java
public boolean isIDSDatabaseLogging();
```
Returns true if the current active IDS database supports logging. Returns false otherwise.

An IDS database that supports logging is a database that was created with the WITH LOG MODE ANSI option, the WITH BUFFERED LOG, or the WITH LOG option.

This method applies to connections to IDS data sources only. An SQLException is thrown if the data source is not an IDS data source.

**isResetRequiredForDB2eWLM**

Format:
```
public boolean isResetRequiredForDB2eWLM();
```

Returns true if the target database server requires clean reuse to support eWLM. Returns false otherwise.

**supportsDB2ProgressiveStreaming**

Format:
```
public boolean supportsDB2ProgressiveStreaming();
```

Returns true if the target data source supports progressive streaming. Returns false otherwise.

---

**DB2Diagnosable interface**

The com.ibm.db2.jcc.DB2Diagnosable interface provides a mechanism for getting DB2 diagnostics from an SQLException.

**DB2Diagnosable methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**getSqlca**

Format:
```
public DB2Sqlca getSqlca();
```

Returns a DB2Sqlca object from a java.sql.Exception that is produced under a IBM Data Server Driver for JDBC and SQLJ.

**getThrowable**

Format:
```
public Throwable getThrowable();
```

Returns a java.lang.Throwable object from a java.sql.Exception that is produced under a IBM Data Server Driver for JDBC and SQLJ.

**printTrace**

Format:
```
static public void printTrace(java.io.PrintWriter printWriter,
                String header);
```

Prints diagnostic information after a java.sql.Exception is thrown under a IBM Data Server Driver for JDBC and SQLJ.

Parameter descriptions:

**printWriter**

The destination for the diagnostic information.
header
User-defined information that is printed at the beginning of the output.

DB2ExceptionFormatter class
The com.ibm.db2.jcc.DB2ExceptionFormatter class contains methods for printing
diagnostic information to a stream.

DB2ExceptionFormatter methods
The following methods are defined only for the IBM Data Server Driver for JDBC
and SQLJ.

printTrace
Formats:
static public void printTrace(java.sql.SQLException sqlException,
java.io.PrintWriter printWriter, String header)
static public void printTrace(DB2Sqlca sqlca,
java.io.PrintWriter printWriter, String header)
static public void printTrace(java.lang.Throwable throwable,
java.io.PrintWriter printWriter, String header)

Prints diagnostic information after an exception is thrown.
Parameter descriptions:
sqlException | sqlca | throwable
The exception that was thrown during a previous JDBC or Java operation.
printWriter
The destination for the diagnostic information.
header
User-defined information that is printed at the beginning of the output.

DB2FileReference class
The com.ibm.db2.jcc.DB2FileReference class is an abstract class that defines
methods that support insertion of data into tables from file reference variables.
This class applies only to IBM Data Server Driver for JDBC and SQLJ type 2
connectivity to DB2 for z/OS Version 9 or later.

DB2FileReference fields
The following constants define types codes only for the IBM Data Server Driver for
JDBC and SQLJ.
public static final short MAX_FILE_NAME_LENGTH = 255
The maximum length of the file name for a file reference variable.

DB2FileReference methods
The following methods are defined only for the IBM Data Server Driver for JDBC
and SQLJ.

getDriverType
Format:
public int getDriverType()
Returns the server data type of the file reference variable. This type is one of the values in com.ibm.db2.jcc.DB2Types.

**getFileSize**

Format:
```
public String getFileSize()
```

Returns the encoding of the data in the file for a DB2FileReference object.

**getFileName**

Format:
```
public String getFileName()
```

Returns the file name for a DB2FileReference object.

**getFileSizeCsid**

Format:
```
public int getFileSizeCsid()
```

Returns the CCSID of the data in the file for a DB2FileReference object.

**setFileName**

Format:
```
public String setFileName(String fileName)
throws java.sql.SQLException
```

Sets the file name in a DB2FileReference object.

Parameter descriptions:

- **fileName**
  The name of the input file for the file reference variable. The name must specify an existing HFS file.

**DB2JCCPlugin class**

The com.ibm.db2.jcc.DB2JCCPlugin class is an abstract class that defines methods that can be implemented to provide DB2 Database for Linux, UNIX, and Windows plug-in support. This class applies only to DB2 Database for Linux, UNIX, and Windows.

**DB2JCCPlugin methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**getTicket**

Format:
```
public abstract byte[] getTicket(String user, String password, byte[] returnedToken)
throws org.ietf.jgss.GSSException
```

Retrieves a Kerberos ticket for a user.

Parameter descriptions:

- **user**
  The user ID for which the Kerberos ticket is to be retrieved.
password
   The password for user.

returnedToken

**DB2ParameterMetaData interface**

The `com.ibm.db2.jcc.DB2ParameterMetaData` interface extends the `java.sql.ParameterMetaData` interface.

**DB2ParameterMetaData methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getParameterMarkerNames
   Format:
   ```java
   public String[] getParameterMarkerNames()
   throws java.sql.SQLException
   ```

   Returns a list of the parameter marker names that are used in an SQL statement.

   This method returns null if the `enableNamedParameterMarkers` property is set to `DB2BaseDataSource.NOT_SET` or `DB2BaseDataSource.NO`, or if there are no named parameter markers in the SQL statement.

getProcedureParameterName
   Format:
   ```java
   public String getProcedureParameterName(int param)
   throws java.sql.SQLException
   ```

   Returns the name in the CREATE PROCEDURE statement of a parameter in an SQL CALL statement. If the parameter has no name in the CREATE PROCEDURE statement, the ordinal position of the parameter in the CREATE PROCEDURE statement is returned.

   **Parameter descriptions:**

   **param**
   The ordinal position of the parameter in the CALL statement.

   This method applies to connections to DB2 Database for Linux, UNIX, and Windows 9.7 or later data servers only.

**DB2PooledConnection class**

The `com.ibm.db2.jdbc.DB2PooledConnection` class provides methods that an application server can use to switch users on a preexisting trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later
DB2PooledConnection methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

getConnection (untrusted or trusted reuse without reauthentication)
Format:
public DB2Connection getConnection()
throws java.sql.SQLException

This method is for dirty reuse of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as user, password, databaseName, serverName, portNumber, planName, and pkList remain unchanged.

getDB2Connection (trusted reuse)
Formats:
public DB2Connection getDB2Connection(byte[] cookie, String user, String password, String userRegistry, byte[] userSecToken, String originalUser, java.util.Properties properties)
throws java.sql.SQLException

Switches the user that is associated with a trusted connection without authentication.

The second form of getDB2Connection is supported only for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

Parameter descriptions:

cookie
A unique cookie that the JDBC driver generates for the Connection instance. The cookie is known only to the application server and the underlying JDBC driver that established the initial trusted connection. The application server passes the cookie that was created by the driver when the pooled connection instance was created. The JDBC driver checks that the supplied cookie matches the cookie of the underlying trusted physical connection to ensure that the request originated from the application server that established the trusted physical connection. If the cookies match, the connection can become available, with different properties, for immediate use by a new user.

user
The client identity that is used by the data source to establish the authorization ID for the database server. If the user was not authenticated
by the application server, the application server must pass a user identity that represents an unauthenticated user.

**password**
The password for user.

**gssCredential**
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

**userNameRegistry**
A name that identifies a mapping service that maps a workstation user ID to a z/OS RACF ID. An example of a mapping service is the Integrated Security Services Enterprise Identity Mapping (EIM). The mapping service is defined by a plugin. Valid values for userNameRegistry are defined by the plugin providers. If userNameRegistry is null, the connection does not use a mapping service.

**userSecToken**
The client's security tokens. This value is traced as part of DB2 for z/OS accounting data. The content of userSecToken is described by the application server and is referred to by the data source as an application server security token.

**originalUser**
The client identity that sends the original request to the application server. originalUser is included in DB2 for z/OS accounting data as the original user ID that was used by the application server.

**properties**
Properties for the reused connection. These properties override any properties that are already defined on the DB2PooledConnection instance.

**getDB2Connection (untrusted reuse with reauthentication)**
Formats:
```java
public DB2Connection getDB2Connection(
    String user,
    String password,
    java.util.Properties properties)
    throws java.sql.SQLException

public DB2Connection getDB2Connection(org.ietf.jgss.GSSCredential gssCredential,
    java.util.Properties properties)
    throws java.sql.SQLException
```

Switches the user that is associated with a untrusted connection, with authentication.

The first form getDB2Connection provides a user ID and password. The second form of getDB2Connection is for connections that use Kerberos security.

Parameter descriptions:

**user**
The user ID that is used by the data source to establish the authorization ID for the database server.

**password**
The password for user.

**properties**
Properties for the reused connection. These properties override any properties that are already defined on the DB2PooledConnection instance.
getDB2Connection (untrusted or trusted reuse without reauthentication)

Formats:

```java
public java.sql.Connection getDB2Connection(
    java.util.Properties properties
) throws java.sql.SQLException
```

Reuses an untrusted connection, without reauthentication.

This method is for dirty reuse of a connection. This means that the connection state is not reset when the object is reused from the pool. Special register settings and property settings remain in effect unless they are overridden by passed properties. Global temporary tables are not deleted. Properties that are not specified are not re-initialized. All JDBC standard transient properties, such as the isolation level, autocommit mode, and read-only mode are reset to their JDBC defaults. Certain properties, such as user, password, databaseName, serverName, portNumber, planName, and pkList remain unchanged.

Parameter descriptions:

**properties**

Properties for the reused connection. These properties override any properties that are already defined on the DB2PooledConnection instance.

DB2PoolMonitor class

The com.ibm.db2.jcc.DB2PoolMonitor class provides methods for monitoring the global transport objects pool that is used for the connection concentrator and Sysplex workload balancing.

DB2PoolMonitor fields

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

```java
public static final int TRANSPORT_OBJECT = 1
```

This value is a parameter for the DB2PoolMonitor.getPoolMonitor method.

DB2PoolMonitor methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**agedOutObjectCount**

Format:

```java
public abstract int agedOutObjectCount()
```

Retrieves the number of objects that exceeded the idle time that was specified by db2.jcc.maxTransportObjectIdleTime and were deleted from the pool.

**createdObjectCount**

Format:

```java
public abstract int createdObjectCount()
```

Retrieves the number of objects that the IBM Data Server Driver for JDBC and SQLJ created since the pool was created.

**getMonitorVersion**

Format:

```java
public int getMonitorVersion()
```
Retrieves the version of the DB2PoolMonitor class that is shipped with the IBM Data Server Driver for JDBC and SQLJ.

**getPoolMonitor**

Format:
```
public static DB2PoolMonitor getPoolMonitor(int monitorType)
```

Retrieves an instance of the DB2PoolMonitor class.

Parameter descriptions:

**monitorType**
The monitor type. This value must be
  `DB2PoolMonitor.TRANSPORT_OBJECT`.

**heavyWeightReusedObjectCount**

Format:
```
public abstract int heavyWeightReusedObjectCount()
```

Retrieves the number of objects that were reused from the pool.

**lightWeightReusedObjectCount**

Format:
```
public abstract int lightWeightReusedObjectCount()
```

Retrieves the number of objects that were reused but were not in the pool. This can happen if a Connection object releases a transport object at a transaction boundary. If the Connection object needs a transport object later, and the original transport object has not been used by any other Connection object, the Connection object can use that transport object.

**longestBlockedRequestTime**

Format:
```
public abstract long longestBlockedRequestTime()
```

Retrieves the longest amount of time that a request was blocked, in milliseconds.

**numberOfConnectionReleaseRefused**

Format:
```
public abstract int numberOfConnectionReleaseRefused()
```

Retrieves the number of times that the release of a connection was refused.

**numberOfRequestsBlocked**

Format:
```
public abstract int numberOfRequestsBlocked()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached its maximum capacity. A blocked request might be successful if an object is returned to the pool before the `db2.jcc.maxTransportObjectWaitTime` is exceeded and an exception is thrown.

**numberOfRequestsBlockedDataSourceMax**

Format:
```
public abstract int numberOfRequestsBlockedDataSourceMax()
```
Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the pool reached the maximum for the DataSource object.

**numberOfRequestsBlockedPoolMax**

Format:
```
public abstract int numberOfRequestsBlockedPoolMax()
```

Retrieves the number of requests that the IBM Data Server Driver for JDBC and SQLJ made to the pool that the pool blocked because the maximum number for the pool was reached.

**removedObjectCount**

Format:
```
public abstract int removedObjectCount()
```

Retrieves the number of objects that have been deleted from the pool since the pool was created.

**shortestBlockedRequestTime**

Format:
```
public abstract long shortestBlockedRequestTime()
```

Retrieves the shortest amount of time that a request was blocked, in milliseconds.

**successfulRequestsFromPool**

Format:
```
public abstract int successfulRequestsFromPool()
```

Retrieves the number of successful requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created. A successful request means that the pool returned an object.

**totalPoolObjects**

Format:
```
public abstract int totalPoolObjects()
```

Retrieves the number of objects that are currently in the pool.

**totalRequestsToPool**

Format:
```
public abstract int totalRequestsToPool()
```

Retrieves the total number of requests that the IBM Data Server Driver for JDBC and SQLJ has made to the pool since the pool was created.

**totalTimeBlocked**

Format:
```
public abstract long totalTimeBlocked()
```

Retrieves the total time in milliseconds for requests that were blocked by the pool. This time can be much larger than the elapsed execution time of the application if the application uses multiple threads.

**DB2PreparedStatement interface**

DB2PreparedStatement methods

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

executeDB2QueryBatch
Format:
public void executeDB2QueryBatch()
throws java.sql.SQLException

Executes a statement batch that contains queries with parameters.
This method is not supported for connections to IBM Informix data sources.

getDBGeneratedKeys
Format:
public java.sql.ResultSet[] getDBGeneratedKeys()
throws java.sql.SQLException

Retrieves automatically generated keys that were created when INSERT statements were executed in a batch. Each ResultSet object that is returned contains the automatically generated keys for a single statement in the batch.
getDBGeneratedKeys returns an array of length 0 under the following conditions:
• getDBGeneratedKeys is called out of sequence. For example, if getDBGeneratedKeys is called before executeBatch, an array of length 0 is returned.
• The PreparedStatement that is executed in a batch was not created using one of the following methods:
  Connection.prepareStatement(String sql, int[] autoGeneratedKeys)
  Connection.prepareStatement(String sql, String[] autoGeneratedColumnNames)
  Connection.prepareStatement(String sql, Statement.RETURN_GENERATED_KEYS)

If getDBGeneratedKeys is called against a PreparedStatement that was created using one of the previously listed methods, and the PreparedStatement is not in a batch, a single ResultSet is returned.

getAddressCost
Format:
public int getEstimateCost()
throws java.sql.SQLException

Returns the estimated cost of an SQL statement from the data server after the data server dynamically prepares the statement successfully. This value is the same as the fourth element in the sqlerrd array of the SQLCA.
If the deferPrepares property is set to true, calling getEstimateCost causes the data server to execute a dynamic prepare operation.
If the SQL statement cannot be prepared, or the data server does not return estimated cost information at prepare time, getEstimateCost returns -1.

getAddressRowCount
Format:
public int getEstimateRowCount()
throws java.sql.SQLException

Developing Java Applications
Returns the estimated row count for an SQL statement from the data server after the data server dynamically prepares the statement successfully. This value is the same as the third element in the sqlerrd array of the SQLCA.

If the deferPrepares property is set to true, calling getEstimateRowCount causes the data server to execute a dynamic prepare operation.

If the SQL statement cannot be prepared, or the data server does not return estimated row count information at prepare time, getEstimateRowCount returns -1.

**setJccArrayAtName**

Format:

```
public void setJccArrayAtName(String parameterMarkerName,
    java.sql.Array x)
  throws java.sql.SQLException
```

Assigns a java.sql.Array value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- **parameterMarkerName**
  The name of the parameter marker to which a value is assigned.

- **x**
  The java.sql.Array value that is assigned to the named parameter marker.

**setJccAsciiStreamAtName**

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```
public void setJccAsciiStreamAtName(String parameterMarkerName,
    java.io.InputStream x, int length)
  throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```
public void setJccAsciiStreamAtName(String parameterMarkerName,
    java.io.InputStream x)
  throws java.sql.SQLException
```

```
public void setJccAsciiStreamAtName(String parameterMarkerName,
    java.io.InputStream x, long length)
  throws java.sql.SQLException
```

Assigns an ASCII value in a java.io.InputStream to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- **parameterMarkerName**
  The name of the parameter marker to which a value is assigned.

- **x**
  The ASCII java.io.InputStream value that is assigned to the parameter marker.

- **length**
  The length in bytes of the java.io.InputStream value that is assigned to the named parameter marker.
**setJccBigDecimalAtName**

Format:

```java
public void setJccBigDecimalAtName(String parameterMarkerName,
        java.math.BigDecimal x)
    throws java.sql.SQLException
```

Assigns a java.math.BigDecimal value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- `parameterMarkerName` The name of the parameter marker to which a value is assigned.
- `x` The java.math.BigDecimal value that is assigned to the named parameter marker.

**setJccBinaryStreamAtName**

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```java
public void setJccBinaryStreamAtName(String parameterMarkerName,
        java.io.InputStream x, int length)
    throws java.sql.SQLException
```

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```java
public void setJccBinaryStreamAtName(String parameterMarkerName,
        java.io.InputStream x)
    throws java.sql.SQLException
```

```java
public void setJccBinaryStreamAtName(String parameterMarkerName,
        java.io.InputStream x, long length)
    throws java.sql.SQLException
```

Assigns a binary value in a java.io.InputStream to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- `parameterMarkerName` The name of the parameter marker to which a value is assigned.
- `x` The binary java.io.InputStream value that is assigned to the parameter marker.
- `length` The number of bytes of the java.io.InputStream value that are assigned to the named parameter marker.

**setJccBlobAtName**

Formats:

Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and later:

```java
public void setJccBlobAtName(String parameterMarkerName,
        java.sql.Blob x)
    throws java.sql.SQLException
```

Assigns a Blob value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- `parameterMarkerName` The name of the parameter marker to which a value is assigned.
- `x` The java.sql.Blob value that is assigned to the named parameter marker.
Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and later:

```java
public void setJccBlobAtName(String parameterMarkerName,
        java.io.InputStream x)
        throws java.sql.SQLException
public void setJccBlobAtName(String parameterMarkerName,
        java.io.InputStream x, long length)
        throws java.sql.SQLException
```

Assigns a BLOB value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- `parameterMarkerName` The name of the parameter marker to which a value is assigned.
- `x` The java.sql.Blob value or java.io.InputStream value that is assigned to the parameter marker.
- `length` The number of bytes of the java.io.InputStream value that are assigned to the named parameter marker.

**setJccBooleanAtName**

Format:

```java
public void setJccBooleanAtName(String parameterMarkerName,
        boolean x)
        throws java.sql.SQLException
```

Assigns a boolean value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- `parameterMarkerName` The name of the parameter marker to which a value is assigned.
- `x` The boolean value that is assigned to the named parameter marker.

**setJccByteAtName**

Format:

```java
public void setJccByteAtName(String parameterMarkerName,
        byte x)
        throws java.sql.SQLException
```

Assigns a byte value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

- `parameterMarkerName` The name of the parameter marker to which a value is assigned.
- `x` The byte value that is assigned to the named parameter marker.

**setJccBytesAtName**

Format:
public void setJccBytesAtName(String parameterMarkerName,
   byte[] x)
   throws java.sql.SQLException

Assigns an array of byte values to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:

   parameterMarkerName
       The name of the parameter marker to which a value is assigned.

   x     The byte array that is assigned to the named parameter marker.

**setJccCharacterStreamAtName**

Formats:
Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and
later:
public void setJccCharacterStreamAtName(String parameterMarkerName,
   java.io.Reader x, int length)
   throws java.sql.SQLException

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and
later:
public void setJccCharacterStreamAtName(String parameterMarkerName,
   java.io.Reader x)
   throws java.sql.SQLException
public void setJccCharacterStreamAtName(String parameterMarkerName,
   java.io.Reader x, long length)
   throws java.sql.SQLException

Assigns a Unicode value in a java.io.Reader to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:

   parameterMarkerName
       The name of the parameter marker to which a value is assigned.

   x     The Unicode java.io.Reader value that is assigned to the named parameter
          marker.

   length
       The number of characters of the java.io.InputStream value that are assigned
to the named parameter marker.

**setJccClobAtName**

Formats:
Supported by the IBM Data Server Driver for JDBC and SQLJ version 3.57 and
later:
public void setJccClobAtName(String parameterMarkerName,
   java.sql.Clob x)
   throws java.sql.SQLException

Supported by the IBM Data Server Driver for JDBC and SQLJ version 4.7 and
later:
public void setJccClobAtName(String parameterMarkerName, java.io.Reader x) throws java.sql.SQLException
public void setJccClobAtName(String parameterMarkerName, java.io.Reader x, long length) throws java.sql.SQLException

Assigns a CLOB value to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Clob value or java.io.Reader value that is assigned to the
named parameter marker.

length
The number of bytes of the java.io.InputStream value that are assigned to
the named parameter marker.

setJccDateAtName
Formats:

public void setJccDateAtName(String parameterMarkerName, java.sql.Date x) throws java.sql.SQLException
public void setJccDateAtName(String parameterMarkerName, java.sql.Date x, java.util.Calendar cal) throws java.sql.SQLException

Assigns a java.sql.Date value to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The java.sql.Date value that is assigned to the named parameter marker.

cal The java.util.Calendar object that the IBM Data Server Driver for JDBC and
SQLJ uses to construct the date.

setJccDoubleAtName
Format:

public void setJccDoubleAtName(String parameterMarkerName, double x) throws java.sql.SQLException

Assigns a value of type double to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
The name of the parameter marker to which a value is assigned.

x The value of type double that is assigned to the parameter marker.
setJccFloatAtName
Format:
public void setJccFloatAtName(String parameterMarkerName,
    float x)
    throws java.sql.SQLException

Assigns a value of type float to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:

    parameterMarkerName
    The name of the parameter marker to which a value is assigned.

    x    The value of type float that is assigned to the parameter marker.

setJccIntAtName
Format:
public void setJccIntAtName(String parameterMarkerName,
    int x)
    throws java.sql.SQLException

Assigns a value of type int to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:

    parameterMarkerName
    The name of the parameter marker to which a value is assigned.

    x    The value of type int that is assigned to the parameter marker.

setJccLongAtName
Format:
public void setJccLongAtName(String parameterMarkerName,
    long x)
    throws java.sql.SQLException

Assigns a value of type long to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property
is set to DB2BaseDataSource.YES (1).
Parameters:

    parameterMarkerName
    The name of the parameter marker to which a value is assigned.

    x    The value of type long that is assigned to the parameter marker.

setJccNullAtName
Format:
public void setJccNullAtName(String parameterMarkerName,
    int jdbcType)
    throws java.sql.SQLException
public void setJccNullAtName(String parameterMarkerName,
    int jdbcType,
    String typeName)
    throws java.sql.SQLException

Assigns the SQL NULL value to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

jdbcType

The JDBC type code of the NULL value that is assigned to the parameter marker, as defined in java.sql.Types.

typeName

If jdbcType is java.sql.Types.DISTINCT or java.sql.Types.REF, the fully-qualified name of the SQL user-defined type of the NULL value that is assigned to the parameter marker.

setJccObjectAtName

Formats:

public void setJccObjectAtName(String parameterMarkerName, java.sql.Object x)
throws java.sql.SQLException
public void setJccObjectAtName(String parameterMarkerName, java.sql.Object x, int targetJdbcType)
throws java.sql.SQLException
public void setJccObjectAtName(String parameterMarkerName, java.sql.Object x, int targetJdbcType, int scale)
throws java.sql.SQLException

Assigns a value with type java.lang.Object to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName

The name of the parameter marker to which a value is assigned.

x

The value with type Object that is assigned to the parameter marker.

targetJdbcType

The data type, as defined in java.sql.Types, that is assigned to the input value when it is sent to the data source.

scale

The scale of the value that is assigned to the parameter marker. This parameter applies only to these cases:

- If targetJdbcType is java.sql.Types.DECIMAL or java.sql.Types.NUMERIC, scale is the number of digits to the right of the decimal point.
- If x has type java.io.InputStream or java.io.Reader, scale is the this is the length of the data in the Stream or Reader object.

setJccShortAtName

Format:

public void setJccShortAtName(String parameterMarkerName, short x)
throws java.sql.SQLException

Assigns a value of type short to a named parameter marker.
This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

`parameterMarkerName`
- The name of the parameter marker to which a value is assigned.

`x`
- The value of type short that is assigned to the parameter marker.

**setJccSQLXMLAtName**

Format:
```java
public void setJccSQLXMLAtName(String parameterMarkerName,
                                 java.sql.SQLXML x)
    throws java.sql.SQLException
```

Assigns a value of type java.sql.SQLXML to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

This method is supported only for connections to DB2 Database for Linux, UNIX, and Windows Version 9.1 or later or DB2 for z/OS Version 9 or later.

Parameters:

`parameterMarkerName`
- The name of the parameter marker to which a value is assigned.

`x`
- The value of type java.sql.SQLXML that is assigned to the parameter marker.

**setJccStringAtName**

Format:
```java
public void setJccStringAtName(String parameterMarkerName,
                                String x)
    throws java.sql.SQLException
```

Assigns a value of type String to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

`parameterMarkerName`
- The name of the parameter marker to which a value is assigned.

`x`
- The value of type String that is assigned to the parameter marker.

**setJccTimeAtName**

Formats:
```java
public void setJccTimeAtName(String parameterMarkerName,
                               java.sql.Time x)
    throws java.sql.SQLException
public void setJccTimeAtName(String parameterMarkerName,
                               java.sql.Time x,
                               java.util.Calendar cal)
    throws java.sql.SQLException
```

Assigns a java.sql.Time value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:
parameterMarkerName
  The name of the parameter marker to which a value is assigned.

x
  The java.sql.Time value that is assigned to the parameter marker.

cal
  The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the time.

setJccTimestampAtName
  Formats:
  public void setJccTimestampAtName(String parameterMarkerName,
         java.sql.Timestamp x)
         throws java.sql.SQLException
  public void setJccTimestampAtName(String parameterMarkerName,
         java.sql.Timestamp x,
         java.util.Calendar cal)
         throws java.sql.SQLException

Assigns a java.sql.Timestamp value to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
  The name of the parameter marker to which a value is assigned.

x
  The java.sql.Timestamp value that is assigned to the parameter marker.

cal
  The java.util.Calendar object that the IBM Data Server Driver for JDBC and SQLJ uses to construct the timestamp.

setJccUnicodeStreamAtName
  Format:
  public void setJccUnicodeStreamAtName(String parameterMarkerName,
         java.io.InputStream x, int length)
         throws java.sql.SQLException

Assigns a Unicode value in a java.io.InputStream to a named parameter marker.

This method can be called only if the enableNamedParameterMarkers property is set to DB2BaseDataSource.YES (1).

Parameters:

parameterMarkerName
  The name of the parameter marker to which a value is assigned.

x
  The Unicode java.io.InputStream value that is assigned to the parameter marker.

length
  The number of bytes of the java.io.InputStream value that are assigned to the parameter marker.

DB2ResultSet interface

The com.ibm.db2.jcc.DB2ResultSet interface is used to create objects from which IBM Data Server Driver for JDBC and SQLJ-only query information can be obtained.

DB2ResultSet implements the java.sql.Wrapper interface.
**DB2ResultSet methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**getDB2RowChangeToken**

Format:

```java
public long DB2ResultSet.getDB2RowChangeToken()
throws java.sql.SQLException
```

Returns the row change token for the current row, if it is available. Returns 0 if optimistic locking columns were not requested or are not available.

This method applies only to connections to DB2 Database for Linux, UNIX, and Windows.

**getDB2RID**

Format:

```java
public Object DB2ResultSet.getDB2RID()
throws java.sql.SQLException
```

Returns the RID for the current row, if it is available. The RID is available if optimistic locking columns were requested and are available. Returns null if optimistic locking columns were not requested or are not available.

This method applies only to connections to DB2 Database for Linux, UNIX, and Windows.

**getDB2RIDType**

Format:

```java
public int DB2ResultSet.getDB2RIDType()
throws java.sql.SQLException
```

Returns the data type of the RID column in a DB2ResultSet. The returned value maps to a `java.sql.Types` constant. If the DB2ResultSet does not contain a RID column, `java.sql.Types.NULL` is returned.

This method applies only to connections to DB2 Database for Linux, UNIX, and Windows.

**DB2ResultSetMetaData interface**

The `com.ibm.db2.jcc.DB2ResultSetMetaData` interface provides methods that provide information about a ResultSet object.

Before a `com.ibm.db2.jcc.DB2ResultSetMetaData` method can be used, a `java.sql.ResultSetMetaData` object that is returned from a `java.sql.ResultSet.getMetaData` call needs to be cast to `com.ibm.db2.jcc.DB2ResultSetMetaData`.

**DB2ResultSetMetaData methods:**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**getDB2OptimisticLockingColumns**

Format:

```java
public int getDB2OptimisticLockingColumns()
throws java.sql.SQLException
```
Returns a value that indicates whether optimistic locking columns are available. Possible values are:

0    Optimistic locking columns are not available.
1    Optimistic locking columns are available, but the change token might not have the granularity to prevent false negatives.
2    Optimistic locking columns are available, and the change token has the granularity to prevent false negatives.

**isDB2ColumnNameDerived**

Format:

```java
public boolean isDB2ColumnNameDerived (int column)
    throws java.sql.SQLException
```

Returns true if the name of a ResultSet column is in the SQL SELECT list that generated the ResultSet.

For example, suppose that a ResultSet is generated from the SQL statement:

```
SELECT EMPNAME, SUM(SALARY) FROM EMP.
```

Column name EMPNAME is derived from the SQL SELECT list, but the name of the column in the ResultSet that corresponds to SUM(SALARY) is not derived from the SELECT list.

Parameter descriptions:

column
    The name of a column in the ResultSet.

**DB2RowID interface**

The com.ibm.db2.jcc.DB2RowID interface is used for declaring Java objects for use with the SQL ROWID data type.

The com.ibm.db2.jcc.DB2RowID interface does not apply to connection to IBM Informix.

**DB2RowID methods**

The following method is defined only for the IBM Data Server Driver for JDBC and SQLJ.

**getBytes**

Format:

```java
public byte[] getBytes()
```

Converts a com.ibm.jcc.DB2RowID object to bytes.

**DB2SimpleDataSource class**

The com.ibm.db2.jcc.DB2SimpleDataSource class extends the DB2BaseDataSource class.

A DB2BaseDataSource object does not support connection pooling or distributed transactions. It contains all of the properties and methods that the DB2BaseDataSource class contains. In addition, DB2SimpleDataSource contains the following IBM Data Server Driver for JDBC and SQLJ-only properties.

DB2SimpleDataSource implements the java.sql.Wrapper interface.
**DB2SimpleDataSource properties**

The following property is defined only for the IBM Data Server Driver for JDBC and SQLJ. See "Properties for the IBM Data Server Driver for JDBC and SQLJ" for an explanation of this property.

String com.ibm.db2.jcc.DB2SimpleDataSource.password

**DB2SimpleDataSource methods**

The following method is defined only for the IBM Data Server Driver for JDBC and SQLJ.

**setPassword**

Format:

```
public void setPassword(String password)
```

Sets the password for the DB2SimpleDataSource object. There is no corresponding getPassword method. Therefore, the password cannot be encrypted because there is no way to retrieve the password so that you can decrypt it.

**DB2Sqlca class**

The com.ibm.db2.jcc.DB2Sqlca class is an encapsulation of the SQLCA.

**DB2Sqlca methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**getMessage**

Format:

```
public abstract String getMessage()
```

Returns error message text.

**getSqlCode**

Format:

```
public abstract int getSqlCode()
```

Returns an SQL error code value.

**getSqlErrd**

Format:

```
public abstract int[] getSqlErrd()
```

Returns an array, each element of which contains an SQLCA SQLERRD.

**getSqlErrmc**

Format:

```
public abstract String getSqlErrmc()
```

Returns a string that contains the SQLCA SQLERRMC values, delimited with spaces.

**getSqlErrmcTokens**

Format:

```
public abstract String[] getSqlErrmcTokens()
```
getSqlErrp
Format:
public abstract String getSqlErrp()

Returns the SQLCA SQLERRP value.

getSqlState
Format:
public abstract String getSqlState()

Returns the SQLCA SQLSTATE value.

getSqlWarn
Format:
public abstract char[] getSqlWarn()

Returns an array, each element of which contains an SQLCA SQLWARN value.

**DB2Statement interface**
The com.ibm.db2.jcc.DB2Statement interface extends the java.sql.Statement interface.

DB2Statement implements the java.sql.Wrapper interface.

**DB2Statement fields**
The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

```java
public static final int RETURN_OPTLOCK_COLUMN_NONE = 0
public static final int RETURN_OPTLOCK_COLUMN_ALWAYS = 1
public static final int RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES = 2
```
These values are arguments for the DB2Statement.executeDB2OptimisticLockingQuery method.

**DB2Statement methods**
The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**executeDB2OptimisticLockingQuery**
Format:
```java
public java.sql.ResultSet DB2Statement.executeDB2OptimisticLockingQuery(
    String sql,
    int returnOptLockingColumn)
throws java.sql.SQLException
```
Executes an SQL query statement, and returns a ResultSet that contains optimistic locking information, if it is requested.

Parameter descriptions:

```java
sql
An SQL SELECT statement that returns a single ResultSet.
```
**returnOptimisticLockingColumns**

Specifies whether optimistic locking columns are returned. Possible values are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2Statement.RETURN_OPTLOCK_COLUMN_NONE (0)</td>
<td>Do not return optimistic locking columns.</td>
</tr>
<tr>
<td>DB2Statement.RETURN_OPTLOCK_COLUMN_ALWAYS (1)</td>
<td>Add row change columns to the result set even if they do not uniquely represent a single row. This setting is equivalent to the database prepare attribute WITH ROW CHANGE COLUMNS POSSIBLY DISTINCT.</td>
</tr>
<tr>
<td>DB2Statement.RETURN_OPTLOCK_COLUMN_NO_FALSE_NEGATIVES (2)</td>
<td>Add row change columns to the result set only if they uniquely represent a single row. This setting is equivalent to the database prepare attribute WITH ROW CHANGE COLUMNS ALWAYS DISTINCT.</td>
</tr>
</tbody>
</table>

**getDB2ClientProgramId**

Format:
```java
public String getDB2ClientProgramId()
throws java.sql.SQLException
```

Returns the user-defined client program identifier for the connection, which is stored on the data source.

getDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

**setDB2ClientProgramId**

Format:
```java
public abstract void setDB2ClientProgramId(String program-ID)
throws java.sql.SQLException
```

Sets a user-defined program identifier for the connection on a data server. That program identifier is an 80-byte string that is used to identify the caller.

setDB2ClientProgramId does not apply to DB2 Database for Linux, UNIX, and Windows data servers.

The DB2 for z/OS server places the string in IFCID 316 trace records along with other statistics, so that you can identify which program is associated with a particular SQL statement.

**getIDSBigSerial**

Format:
```java
public int getIDSBigSerial()
throws java.sql.SQLException
```

Retrieves an automatically generated key from a BIGSERIAL column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for getIDSBigSerial to execute successfully:

- The INSERT statement is the last SQL statement that is executed before this method is called.
- The table into which the row is inserted contains a BIGSERIAL column.
• The form of the JDBC Connection.prepareStatement method or Statement.executeUpdate method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IBM Informix (IDS) databases.

**getIDSSerial**

Format:
```java
public int getIDSSerial()
    throws java.sql.SQLException
```

Retrieves an automatically generated key from a SERIAL column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for getIDSSerial to execute successfully:

• The INSERT statement is the last SQL statement that is executed before this method is called.
• The table into which the row is inserted contains a SERIAL column.
• The form of the JDBC Connection.prepareStatement method or Statement.executeUpdate method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IBM Informix (IDS) databases.

**getIDSSerial8**

Format:
```java
public long getIDSSerial8()
    throws java.sql.SQLException
```

Retrieves an automatically generated key from a SERIAL8 column after the automatically generated key was inserted by a previously executed INSERT statement.

The following conditions must be true for getIDSSerial8 to execute successfully:

• The INSERT statement is the last SQL statement that is executed before this method is called.
• The table into which the row is inserted contains a SERIAL8 column.
• The form of the JDBC Connection.prepareStatement method or Statement.executeUpdate method that prepares or executes the INSERT statement does not have parameters that request automatically generated keys.

This method applies only to connections to IDS data sources.

**getIDSSQLStatementOffSet**

Format:
```java
public int getIDSSQLStatementOffSet()
    throws java.sql.SQLException
```

After an SQL statement executes on an IDS data source, if the statement has a syntax error, getIDSSQLStatementOffSet returns the offset into the statement text of the syntax error.

getIDSSQLStatementOffSet returns:

• 0, if the statement does not have a syntax error.
-1, if the data source is not IDS.

This method applies only to connections to IDS data sources.

**DB2SystemMonitor interface**

The com.ibm.db2.jcc.DB2SystemMonitor interface is used for collecting system monitoring data for a connection. Each connection can have one DB2SystemMonitor instance.

**DB2SystemMonitor fields**

The following fields are defined only for the IBM Data Server Driver for JDBC and SQLJ.

```java
public final static int RESET_TIMES
public final static int ACCUMULATE_TIMES
```

These values are arguments for the DB2SystemMonitor.start method. RESET_TIMES sets time counters to zero before monitoring starts. ACCUMULATE_TIMES does not set time counters to zero.

**DB2SystemMonitor methods**

The following methods are defined only for the IBM Data Server Driver for JDBC and SQLJ.

**enable**

Format:

```java
public void enable(boolean on)
throws java.sql.SQLException
```

Enables the system monitor that is associated with a connection. This method cannot be called during monitoring. All times are reset when enable is invoked.

**getApplicationTimeMillis**

Format:

```java
public long getApplicationTimeMillis()
throws java.sql.SQLException
```

Returns the sum of the application, JDBC driver, network I/O, and database server elapsed times. The time is in milliseconds.

A monitored elapsed time interval is the difference, in milliseconds, between these points in the JDBC driver processing:

**Interval beginning**

When start is called.

**Interval end**

When stop is called.

getApplicationTimeMillis returns 0 if system monitoring is disabled. Calling this method without first calling the stop method results in an SQLException.

**getCoreDriverTimeMicros**

Format:

```java
public long getCoreDriverTimeMicros()
throws java.sql.SQLException
```

-1, if the data source is not IDS.

This method applies only to connections to IDS data sources.
Returns the sum of elapsed monitored API times that were collected while system monitoring was enabled. The time is in microseconds.

A monitored API is a JDBC driver method for which processing time is collected. In general, elapsed times are monitored only for APIs that might result in network I/O or database server interaction. For example, PreparedStatement.setXXX methods and ResultSet.getXXX methods are not monitored.

Monitored API elapsed time includes the total time that is spent in the driver for a method call. This time includes any network I/O time and database server elapsed time.

A monitored API elapsed time interval is the difference, in microseconds, between these points in the JDBC driver processing:

**Interval beginning**
- When a monitored API is called by the application.

**Interval end**
- Immediately before the monitored API returns control to the application.

getCoreDriverTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an SQLException.

**getNetworkIOTimeMicros**
Format:

```java
public long getNetworkIOTimeMicros()
    throws java.sql.SQLException
```

Returns the sum of elapsed network I/O times that were collected while system monitoring was enabled. The time is in microseconds.

Elapsed network I/O time includes the time to write and read DRDA data from network I/O streams. A network I/O elapsed time interval is the time interval to perform the following operations in the JDBC driver:

- Issue a TCP/IP command to send a DRDA message to the database server. This time interval is the difference, in microseconds, between points immediately before and after a write and flush to the network I/O stream is performed.
- Issue a TCP/IP command to receive DRDA reply messages from the database server. This time interval is the difference, in microseconds, between points immediately before and after a read on the network I/O stream is performed.

Network I/O time intervals are captured for all send and receive operations, including the sending of messages for commits and rollbacks.

The time spent waiting for network I/O might be impacted by delays in CPU dispatching at the database server for low-priority SQL requests.

getNetworkIOTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method, or calling this method when the underlying JVM does not support reporting times in microseconds results in an SQLException.

**getServerTimeMicros**
Format:
public long getServerTimeMicros()  
  throws java.sql.SQLException

Returns the sum of all reported database server elapsed times that were collected while system monitoring was enabled. The time is in microseconds.

The database server reports elapsed times under these conditions:

- The database server supports returning elapsed time data to the client.
  DB2 Database for Linux, UNIX, and Windows Version 9.5 and later and DB2 for z/OS support this function.
- The database server performs operations that can be monitored. For example, database server elapsed time is not returned for commits or rollbacks.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 Database for Linux, UNIX, and Windows, and IBM Data Server Driver for JDBC and SQLJ type 4 connectivity: The database server elapsed time is defined as the elapsed time to parse the request data stream, process the command, and generate the reply data stream at the database server. Network time to receive or send the data stream is not included. The database server elapsed time interval is the difference, in microseconds, between these points in the database server processing:

**Interval beginning**
When the operating system dispatches the database server to process a TCP/IP message that is received from the JDBC driver.

**Interval end**
When the database server is ready to issue the TCP/IP command to return the reply message to the client.

For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity to DB2 for z/OS: The database server elapsed time interval is the difference, in microseconds, between these points in the JDBC driver native processing:

**Interval beginning**
The z/OS Store Clock (STCK) value when a JDBC driver native method calls the RRS attachment facility to process an SQL request.

**Interval end**
The z/OS Store Clock (STCK) value when control returns to the JDBC driver native method following an RRS attachment facility call to process an SQL request.

getServerTimeMicros returns 0 if system monitoring is disabled. Calling this method without first calling the stop method results in a SQLException.

**start**
Format:

```java
public void start (int lapMode)  
  throws java.sql.SQLException
```

If the system monitor is enabled, start begins the collection of system monitoring data for a connection. Valid values for `lapMode` are `RESET_TIMES` or `ACCUMULATE_TIMES`.

Calling this method with system monitoring disabled does nothing. Calling this method more than once without an intervening stop call results in an SQLException.
stop
Format:
public void stop()
    throws java.sql.SQLException

If the system monitor is enabled, stop ends the collection of system monitoring data for a connection. After monitoring is stopped, monitored times can be obtained with the getXXX methods of DB2SystemMonitor.

Calling this method with system monitoring disabled does nothing. Calling this method without first calling start, or calling this method more than once without an intervening start call results in an SQLException.

DB2TraceManager class
The com.ibm.db2.jcc.DB2TraceManager class controls the global log writer.

The global log writer is driver-wide, and applies to all connections. The global log writer overrides any other JDBC log writers. In addition to starting the global log writer, the DB2TraceManager class provides the ability to suspend and resume tracing of any type of log writer. That is, the suspend and resume methods of the DB2TraceManager class apply to all current and future DriverManager log writers, DataSource log writers, or IBM Data Server Driver for JDBC and SQLJ-only connection-level log writers.

DB2TraceManager methods
getTraceManager
Format:
static public DB2TraceManager getTraceManager()
    throws java.sql.SQLException

Gets an instance of the global log writer.

setLogWriter
Formats:
public abstract void setLogWriter(String traceDirectory,
    String baseTraceFileName, int traceLevel)
    throws java.sql.SQLException
public abstract void setLogWriter(String traceFile,
    boolean fileAppend, int traceLevel)
    throws java.sql.SQLException
public abstract void setLogWriter(java.io.PrintWriter logWriter,
    int traceLevel)
    throws java.sql.SQLException

Enables a global trace. After setLogWriter is called, all calls for DataSource or Connection traces are discarded until DB2TraceManager.unsetLogWriter is called.

When setLogWriter is called, all future Connection or DataSource traces are redirected to a trace file or PrintWriter, depending on the form of setLogWriter that you use. If the global trace is suspended when setLogWriter is called, the specified settings take effect when the trace is resumed.

Parameter descriptions:

traceDirectory
    Specifies a directory into which global trace information is written. This setting overrides the settings of the traceDirectory and logWriter properties for a DataSource or DriverManager connection.
When the form of setLogWriter with the traceDirectory parameter is used, the JDBC driver sets the traceFileAppend property to false when setLogWriter is called, which means that the existing log files are overwritten. Each JDBC driver connection is traced to a different file in the specified directory. The naming convention for the files in that directory depends on whether a non-null value is specified for baseTraceFileName:

- If a null value is specified for baseTraceFileName, a connection is traced to a file named traceFile_global_n.
  - n is the nth JDBC driver connection.
- If a non-null value is specified for baseTraceFileName, a connection is traced to a file named baseTraceFileName_global_n.
  - baseTraceFileName is the value of the baseTraceFileName parameter.
  - n is the nth JDBC driver connection.

**baseTraceFileName**

Specifies the stem for the names of the files into which global trace information is written. The combination of baseTraceFileName and traceDirectory determines the full path name for the global trace log files.

**traceFileName**

Specifies the file into which global trace information is written. This setting overrides the settings of the traceFile and logWriter properties for a DataSource or DriverManager connection.

When the form of setLogWriter with the traceFileName parameter is used, only one log file is written.

traceFileName can include a directory path.

**logWriter**

Specifies a character output stream to which all global log records are written.

This value overrides the logWriter property on a DataSource or DriverManager connection.

**traceLevel**

Specifies what to trace.

You can specify one or more of the following traces with the traceLevel parameter:

- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity for DB2 Database for Linux, UNIX, and Windows only) (X'800')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')
- com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR (X'20000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS () (X'40000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:
• Use bitwise OR (\(|\)) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:
  TRACE_DRDA_FLOWS | TRACE_CONNECTION_CALLS
• Use a bitwise complement (tilde (~)) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:
  ~TRACE_DRDA_FLOWS

fileAppend
  Specifies whether to append to or overwrite the file that is specified by the traceFile parameter. true means that the existing file is not overwritten.

unsetLogWriter
  Format:
  public abstract void unsetLogWriter()
  throws java.sql.SQLException
  Disables the global log writer override for future connections.

suspendTrace
  Format:
  public void suspendTrace()
  throws java.sql.SQLException
  Suspends all global, Connection-level, or DataSource-level traces for current and future connections. suspendTrace can be called when the global log writer is enabled or disabled.

resumeTrace
  Format:
  public void resumeTrace()
  throws java.sql.SQLException
  Resumes all global, Connection-level, or DataSource-level traces for current and future connections. resumeTrace can be called when the global log writer is enabled or disabled. If the global log writer is disabled, resumeTrace resumes Connection-level or DataSource-level traces. If the global log writer is enabled, resumeTrace resumes the global trace.

getLogWriter
  Format:
  public abstract java.io.PrintWriter getLogWriter()
  throws java.sql.SQLException
  Returns the PrintWriter for the global log writer, if it is set. Otherwise, getLogWriter returns null.

getTraceFile
  Format:
  public abstract String getTraceFile()
  throws java.sql.SQLException
  Returns the name of the destination file for the global log writer, if it is set. Otherwise, getTraceFile returns null.
getTraceDirectory
Format:
public abstract String getTraceDirectory()
throws java.sql.SQLException

Returns the name of the destination directory for global log writer files, if it is set. Otherwise, getTraceDirectory returns null.

getTraceLevel
Format:
public abstract int getTraceLevel()
throws java.sql.SQLException

Returns the trace level for the global trace, if it is set. Otherwise, getTraceLevel returns -1 (TRACE_ALL).

getTraceFileAppend
Format:
public abstract boolean getTraceFileAppend()
throws java.sql.SQLException

Returns true if the global trace records are appended to the trace file. Otherwise, getTraceFileAppend returns false.

DB2TraceManagerMXBean interface
The com.ibm.db2.jcc.mx.DB2TraceManagerMXBean interface is the means by which an application makes DB2TraceManager available as an MXBean for the remote trace controller.

DB2TraceManagerMXBean methods
setTraceFile
Format:
public void setTraceFile(String traceFile,
    boolean fileAppend, int traceLevel)
throws java.sql.SQLException

Specifies the name of the file into which the remote trace manager writes trace information, and the type of information that is to be traced.

Parameter descriptions:

traceFileName
Specifies the file into which global trace information is written. This setting overrides the settings of the traceFile and logWriter properties for a DataSource or DriverManager connection.

When the form of setLogWriter with the traceFileName parameter is used, only one log file is written.

traceFileName can include a directory path.

fileAppend
Specifies whether to append to or overwrite the file that is specified by the traceFile parameter. true means that the existing file is not overwritten.

traceLevel
Specifies what to trace.

You can specify one or more of the following traces with the traceLevel parameter:
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS (IBM Data Server
  Driver for JDBC and SQLJ type 2 connectivity for DB2 Database for
  Linux, UNIX, and Windows only) (X'800')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_META_CALLS (X'2000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR (X'20000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS () (X'40000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:
• Use bitwise OR (|) operators with two or more trace values. For
  example, to trace DRDA flows and connection calls, specify this value
  for traceLevel:
    TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS
• Use a bitwise complement (tilde (~)) operator with a trace value to
  specify all except a certain trace. For example, to trace everything except
  DRDA flows, specify this value for traceLevel:
    ~TRACE_DRDA_FLOWS

getTraceFile
Format:
public void getTraceFile()
  throws java.sql.SQLException

Returns the name of the destination file for the remote trace controller, if it is
set. Otherwise, getTraceFile returns null.

setTraceDirectory
Format:
public void setTraceDirectory(String traceDirectory,
  String baseTraceFileName,
  int traceLevel) throws java.sql.SQLException

Specifies the name of the directory into which the remote trace controller
writes trace information, and the type of information that is to be traced.

Parameter descriptions:
traceDirectory
  Specifies a directory into which trace information is written. This setting
  overrides the settings of the traceDirectory and logWriter properties for a
  DataSource or DriverManager connection.

  Each JDBC driver connection is traced to a different file in the specified
directory. The naming convention for the files in that directory depends on
whether a non-null value is specified for baseTraceFileName:
• If a null value is specified for baseTraceFileName, a connection is traced to a file named traceFile_global_n. 
  n is the n<sup>th</sup> JDBC driver connection.

• If a non-null value is specified for baseTraceFileName, a connection is traced to a file named baseTraceFileName_global_n. 
  baseTraceFileName is the value of the baseTraceFileName parameter. 
  n is the n<sup>th</sup> JDBC driver connection.

**baseTraceFileName**

Specifies the stem for the names of the files into which global trace information is written. The combination of baseTraceFileName and traceDirectory determines the full path name for the global trace log files.

**traceLevel**

Specifies what to trace.

You can specify one or more of the following traces with the traceLevel parameter:

• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_NONE (X'00')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTION_CALLS (X'01')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_STATEMENT_CALLS (X'02')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_CALLS (X'04')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRIVER_CONFIGURATION (X'10')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_CONNECTS (X'20')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS (X'40')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_RESULT_SET_META_DATA (X'80')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_PARAMETER_META_DATA (X'100')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DIAGNOSTICS (X'200')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SQLJ (X'400')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_XA_CALLS (IBM Data Server Driver for JDBC and SQLJ type 2 connectivity for DB2 Database for Linux, UNIX, and Windows only) (X'800')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_Meta_CALLS (X'2000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DATASOURCE_CALLS (X'4000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_LARGE_OBJECT_CALLS (X'8000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR (X'20000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_TRACEPOINTS () (X'40000')
• com.ibm.db2.jcc.DB2BaseDataSource.TRACE_ALL (X'FFFFFFFF')

To specify more than one trace, use one of these techniques:

• Use bitwise OR (|) operators with two or more trace values. For example, to trace DRDA flows and connection calls, specify this value for traceLevel:

  TRACE_DRDA_FLOWS|TRACE_CONNECTION_CALLS

• Use a bitwise complement (tilde (~)) operator with a trace value to specify all except a certain trace. For example, to trace everything except DRDA flows, specify this value for traceLevel:

  ~TRACE_DRDA_FLOWS

**getTraceFileAppend**

Format:

```java
public abstract boolean getTraceFileAppend()
throws java.sql.SQLException
```

Returns true if trace records that are generated by the trace controller are appended to the trace file. Otherwise, getTraceFileAppend returns false.
getTraceDirectory
Format:
public void getTraceDirectory()
    throws java.sql.SQLException

Returns the name of the destination directory for trace records that are
generated by the trace controller, if it is set. Otherwise, getTraceDirectory
returns null.

getTraceLevel
Format:
public void getTraceLevel()
    throws java.sql.SQLException

Returns the trace level for the trace records that are generated by the trace
controller, if it is set. Otherwise, getTraceLevel returns -1 (TRACE_ALL).

unsetLogWriter
Format:
public abstract void unsetLogWriter()
    throws java.sql.SQLException

Disables the global log writer override for future connections.

suspendTrace
Format:
public void suspendTrace()
    throws java.sql.SQLException

Suspends all global, Connection-level, or DataSource-level traces for current
and future connections. suspendTrace can be called when the global log writer
is enabled or disabled.

resumeTrace
Format:
public void resumeTrace()
    throws java.sql.SQLException

Resumes all global, Connection-level, or DataSource-level traces for current and
future connections. resumeTrace can be called when the global log writer is
enabled or disabled. If the global log writer is disabled, resumeTrace resumes
Connection-level or DataSource-level traces. If the global log writer is enabled,
resumeTrace resumes the global trace.

DB2Types class
The com.ibm.db2.jcc.DB2Types class provides fields that define IBM Data Server
Driver for JDBC and SQLJ-only data types.

DB2Types fields
The following constants define types codes only for the IBM Data Server Driver for
JDBC and SQLJ.
• public final static int BLOB_FILE = -100002
• public final static int CLOB_FILE = -100003
• public final static int CURSOR = -100008
• public final static int DECFLOAT = -100001
public final static int XML_AS_BLOB_FILE = -100004
public final static int XML_AS_CLOB_FILE = -100005

DB2XADatasource class

DB2XADatasource is a factory for XADatasource objects. An object that implements this interface is registered with a naming service that is based on the Java Naming and Directory Interface (JNDI).

The com.ibm.db2.jcc.DB2XADatasource class extends the com.ibm.db2.jcc.DB2BaseDatasource class, and implements the javax.sql.XADatasource, java.io.Serializable, and javax.naming.Referenceable interfaces.

DB2XADatasource methods

getDB2TrustedXAConnection

An application server using a system authorization ID uses this method to establish a trusted connection.

Trusted connections are supported for:

- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

The following elements are returned in Object[]:

- The first element is a DB2TrustedXAConnection instance.
- The second element is a unique cookie for the generated XA connection instance.

The first form getDB2TrustedXAConnection provides a user ID and password. The second form of getDB2TrustedXAConnection uses the user ID and password of the DB2XADatasource object. The third form of getDB2TrustedXAConnection is for connections that use Kerberos security.

Parameter descriptions:

user
The authorization ID that is used to establish the trusted connection.

password
The password for the authorization ID that is used to establish the trusted connection.
**gssCredential**
If the data source uses Kerberos security, specifies a delegated credential that is passed from another principal.

**properties**
Properties for the connection.

**getDB2TrustedPooledConnection**
Format:
```java
public Object[] getDB2TrustedPooledConnection(java.util.Properties properties) throws java.sql.SQLException
```

An application server using a system authorization ID uses this method to establish a trusted connection, using the user ID and password for the DB2XADatasource object.

Trusted connections are supported for:
- IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to:
  - DB2 Database for Linux, UNIX, and Windows Version 9.5 or later
  - DB2 for z/OS Version 9.1 or later
  - IBM Informix Version 11.70 or later
- IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS Version 9.1 or later

The following elements are returned in Object[]:
- The first element is a trusted DB2TrustedPooledConnection instance.
- The second element is a unique cookie for the generated pooled connection instance.

Parameter descriptions:

**properties**
Properties for the connection.

**getDB2XAConnection**
Formats:
```java
public DB2XAConnection getDB2XAConnection(String user, String password, java.util.Properties properties) throws java.sql.SQLException
public DB2XAConnection getDB2XAConnection(org.ietf.jgss.GSSCredential gssCredential, java.util.Properties properties) throws java.sql.SQLException
```

Establishes the initial untrusted connection in a heterogeneous pooling environment.

The first form getDB2PooledConnection provides a user ID and password. The second form of getDB2XAConnection is for connections that use Kerberos security.

Parameter descriptions:

**user**
The authorization ID that is used to establish the connection.

**password**
The password for the authorization ID that is used to establish the connection.
gssCredential
If the data source uses Kerberos security, specifies a delegated credential
that is passed from another principal.

properties
Properties for the connection.

DB2Xml interface
The com.ibm.db2.jcc.DB2Xml interface is used for declaring Java objects for use
with the DB2 XML data type.

DB2Xml methods
The following method is defined only for the IBM Data Server Driver for JDBC
and SQLJ.

closeDB2Xml
Format:
```
public void closeDB2Xml()
throws SQLException
```

Releases the resources that are associated with a com.ibm.jcc.DB2Xml object.

getDB2AsciiStream
Format:
```
public java.io.InputStream getDB2AsciiStream()
throws SQLException
```

Retrieves data from a DB2Xml object, and converts the data to US-ASCII
encoding.

getDB2BinaryStream
Format:
```
public java.io.InputStream getDB2BinaryStream()
throws SQLException
```

Retrieves data from a DB2Xml object as a binary stream. The character
encoding of the bytes in the binary stream is defined in the XML 1.0
specification.

getDB2Bytes
Format:
```
public byte[] getDB2Bytes()
throws SQLException
```

Retrieves data from a DB2Xml object as a byte array. The character encoding of
the bytes is defined in the XML 1.0 specification.

getDB2CharacterStream
Format:
```
public java.io.Reader getDB2CharacterStream()
throws SQLException
```

Retrieves data from a DB2Xml object as a java.io.Reader object.

getDB2String
Format:
```
public String getDB2String()
throws SQLException
```
Retrieves data from a DB2Xml object as a String value.

**getDB2XmlAsciiStream**
Format:
```java
public InputStream getDB2XmlAsciiStream()
    throws SQLException
```
Retrieves data from a DB2Xml object, converts the data to US-ASCII encoding, and imbeds an XML declaration with an encoding specification for US-ASCII in the returned data.

**getDB2XmlBinaryStream**
Format:
```java
public java.io.InputStream getDB2XmlBinaryStream(String targetEncoding)
    throws SQLException
```
Retrieves data from a DB2Xml object as a binary stream, converts the data to `targetEncoding`, and imbeds an XML declaration with an encoding specification for `targetEncoding` in the returned data.

Parameter:

- `targetEncoding`
  A valid encoding name that is listed in the IANA Charset Registry. The encoding names that are supported by the DB2 server are listed in "Mappings of CCSIDs to encoding names for serialized XML output data".

**getDB2XmlBytes**
Format:
```java
public byte[] getDB2XmlBytes(String targetEncoding)
    throws SQLException
```
Retrieves data from a DB2Xml object as a byte array, converts the data to `targetEncoding`, and imbeds an XML declaration with an encoding specification for `targetEncoding` in the returned data.

Parameter:

- `targetEncoding`
  A valid encoding name that is listed in the IANA Charset Registry. The encoding names that are supported by the DB2 server are listed in "Mappings of CCSIDs to encoding names for serialized XML output data".

**getDB2XmlCharacterStream**
Format:
```java
public java.io.Reader getDB2XmlCharacterStream()
    throws SQLException
```
Retrieves data from a DB2Xml object as a java.io.Reader object, converts the data to ISO-10646-UCS-2 encoding, and imbeds an XML declaration with an encoding specification for ISO-10646-UCS-2 in the returned data.

**getDB2XmlString**
Format:
```java
public String getDB2XmlString()
    throws SQLException
```
Retrieves data from a DB2Xml object as a `String` object, converts the data to ISO-10646-UCS-2 encoding, and imbeds an XML declaration with an encoding specification for ISO-10646-UCS-2 in the returned data.
isDB2XmlClosed

Format:

```java
public boolean isDB2XmlClosed()
    throws SQLException;
```

Indicates whether a com.ibm.jcc.DB2Xml object has been closed.

---

**JDBC differences between the current IBM Data Server Driver for JDBC and SQLJ and earlier DB2 JDBC drivers**

Before you can upgrade your JDBC applications from older drivers to the IBM Data Server Driver for JDBC and SQLJ, you need to understand the differences between those drivers.

**Important:** The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) is deprecated. This information is provided to assist you in moving your applications to the IBM Data Server Driver for JDBC and SQLJ.

**Supported methods**

For a comparison of method support by the JDBC drivers, see "Driver support for JDBC APIs".

**Use of progressive streaming by the JDBC drivers**

For IBM Data Server Driver for JDBC and SQLJ, Version 3.50 and later, use of progressive streaming is the default for LOB retrieval, for connections to DB2 Database for Linux, UNIX, and Windows Version 9.5 and later.

Progressive streaming is supported in the IBM Data Server Driver for JDBC and SQLJ Version 3.1 and later, but for IBM Data Server Driver for JDBC and SQLJ version 3.2 and later, use of progressive streaming is the default for LOB and XML retrieval, for connections to DB2 for z/OS Version 9.1 and later.

Previous versions of the IBM Data Server Driver for JDBC and SQLJ and the DB2 JDBC Type 2 Driver did not support progressive streaming.

**Important:** With progressive streaming, when you retrieve a LOB or XML value from a ResultSet into an application variable, you can manipulate the contents of that application variable until you move the cursor or close the cursor on the ResultSet. After that, the contents of the application variable are no longer available to you. If you perform any actions on the LOB in the application variable, you receive an SQLException. For example, suppose that progressive streaming is enabled, and you execute statements like this:

```java
...
ResultSet rs = stmt.executeQuery("SELECT CLOBCOL FROM MY_TABLE");
rs.next();  // Retrieve the first row of the ResultSet
Clob clobFromRow1 = rs.getClob(1);  // Put the CLOB from the first column of
     // the first row in an application variable
String substr1Clob = clobFromRow1.getSubString(1,50);  // Retrieve the first 50 bytes of the CLOB
rs.next();  // Move the cursor to the next row.
    // clobFromRow1 is no longer available.
// String substr2Clob = clobFromRow1.getSubString(51,100);  // This statement would yield an SQLException
Clob clobFromRow2 = rs.getClob(1);
```
After you execute rs.next() to position the cursor at the second row of the ResultSet, the CLOB value in clobFromRow1 is no longer available to you. Similarly, after you execute rs.close() to close the ResultSet, the values in clobFromRow1 and clobFromRow2 are no longer available.

To avoid errors that are due to this changed behavior, you need to take one of the following actions:

- Modify your applications.
  - Applications that retrieve LOB data into application variables can manipulate the data in those application variables only until the cursors that were used to retrieve the data are moved or closed.
- Disable progressive streaming by setting the `progressiveStreaming` property to `DB2BaseDataSource.NO` (2).

**ResultSetMetaData values for IBM Data Server Driver for JDBC and SQLJ version 4.0 and later**

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` differs from the default behavior for earlier JDBC drivers.

If you need to use IBM Data Server Driver for JDBC and SQLJ version 4.0 or later, but your applications need to return the `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` values that were returned with older JDBC drivers, you can set the `useJDBC4ColumnNameAndLabelSemantics` Connection and `DataSource` property to `DB2BaseDataSource.NO` (2).

**Batch updates with automatically generated keys have different results in different driver versions**

With the IBM Data Server Driver for JDBC and SQLJ version 3.52 or later, preparing an SQL statement for retrieval of automatically generated keys is supported.

With the IBM Data Server Driver for JDBC and SQLJ version 3.50 or version 3.51, preparing an SQL statement for retrieval of automatically generated keys and using the `PreparedStatement` object for batch updates causes an `SQLException`.

Versions of the IBM Data Server Driver for JDBC and SQLJ before Version 3.50 do not throw an `SQLException` when an application calls the `addBatch` or `executeBatch` method on a `PreparedStatement` object that is prepared to return automatically generated keys. However, the `PreparedStatement` object does not return automatically generated keys.

**Initial value of the CURRENT_CLIENT_ACCTNG special register**

For a JDBC or SQLJ application that runs under the IBM Data Server Driver for JDBC and SQLJ version 2.6 or later, using type 4 connectivity, the initial value for the DB2 for z/OS CURRENT_CLIENT_ACCTNG special register is the concatenation of the DB2 for z/OS version and the value of the clientWorkStation.
property. For any other JDBC driver, version, and connectivity, the initial value is not set.

**Support for scrollable and updatable ResultSets**

The IBM Data Server Driver for JDBC and SQLJ supports scrollable and updatable ResultSets.

The DB2 JDBC Type 2 Driver supports scrollable ResultSets but not updatable ResultSets.

**Difference in URL syntax**

The syntax of the `url` parameter in the `DriverManager.getConnection` method is different for each driver. See the following topics for more information:

- "Connect to a data source using the DriverManager interface with the IBM Data Server Driver for JDBC and SQLJ"
- "Connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver"

**Difference in error codes and SQLSTATEs returned for driver errors**

The IBM Data Server Driver for JDBC and SQLJ does not use existing SQLCODEs or SQLSTATEs for internal errors, as the other drivers do. See "Error codes issued by the IBM Data Server Driver for JDBC and SQLJ" and "SQLSTATEs issued by the IBM Data Server Driver for JDBC and SQLJ".

The JDBC/JJ driver for OS/390 and z/OS return ODBC SQLSTATEs when internal errors occur.

**How much error message text is returned**

With the IBM Data Server Driver for JDBC and SQLJ, when you execute `SQLException.getMessage()`, formatted message text is not returned unless you set the `retrieveMessagesFromServerOnGetMessage` property to true.

With the DB2 JDBC Type 2 Driver, when you execute `SQLException.getMessage()`, formatted message text is returned.

**Security mechanisms**

The JDBC drivers have different security mechanisms.

For information on IBM Data Server Driver for JDBC and SQLJ security mechanisms, see "Security under the IBM Data Server Driver for JDBC and SQLJ".

For information on security mechanisms for the DB2 JDBC Type 2 Driver, see "Security under the DB2 JDBC Type 2 Driver".

**Support for read-only connections**

With the IBM Data Server Driver for JDBC and SQLJ, you can make a connection read-only through the `readOnly` property for a Connection or DataSource object.
The DB2 JDBC Type 2 Driver uses the Connection.setReadOnly value when it determines whether to make a connection read-only. However, setting Connection.setReadOnly(true) does not guarantee that the connection is read-only.

**Results returned from ResultSet.getString for a BIT DATA column**

The IBM Data Server Driver for JDBC and SQLJ returns data from a ResultSet.getString call for a CHAR FOR BIT DATA or VARCHAR FOR BIT DATA column as a lowercase hexadecimal string.

The DB2 JDBC Type 2 Driver returns the data as an uppercase hexadecimal string.

**Results returned from ResultSet.getString for a TIMESTAMP column**

By default, the IBM Data Server Driver for JDBC and SQLJ truncates trailing zeroes when it returns data for a ResultSet.getString call for a TIMESTAMP column value. You can change this behavior with the timestampPrecisionReporting property.

The DB2 JDBC Type 2 Driver does not truncate trailing zeroes when it returns data for a ResultSet.getString call for a TIMESTAMP column value.

**Result of an executeUpdate call that affects no rows**

The IBM Data Server Driver for JDBC and SQLJ generates an SQLWarning when an executeUpdate call affects no rows.

The DB2 JDBC Type 2 Driver does not generate an SQLWarning.

**Result of a getDate or getTime call for a TIMESTAMP column**

The IBM Data Server Driver for JDBC and SQLJ does not generate an SQLWarning when a getDate or getTime call is made against a TIMESTAMP column.

The DB2 JDBC Type 2 Driver generates an SQLWarning when a getDate or getTime call is made against a TIMESTAMP column.

**Date and time adjustment for input and output values that do not correspond to real dates and times**

During update or retrieval of data in SQL DATE, TIME, or TIMESTAMP columns, the IBM Data Server Driver for JDBC and SQLJ adjusts date and time values that do not correspond to real dates and times. For example, if you update a TIMESTAMP column with the value 2007-12-31 24:00:00.0, the IBM Data Server Driver for JDBC and SQLJ adjusts the value to 2008-01-01 00:00:00.0. If you update a TIMESTAMP column with the value 9999-12-31 24:00:00.0, the IBM Data Server Driver for JDBC and SQLJ throws an exception because the adjusted value, 10000-01-01 00:00:00.0, is invalid.

The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does no adjustment of date or time values that do not correspond to real dates or times. That driver passes the values to and from the database as they are. For example, if you update a TIMESTAMP column with the value 9999-12-31 24:00:00.0 under the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows, no exception is thrown. See the information on date, time, and timestamp values that can cause problems in JDBC and SQLJ applications for more information.
Default format for retrieval of DATE, TIME, or TIMESTAMP column data into strings

For the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows, the default format for retrieval of DATE, TIME, or TIMESTAMP column data into strings is JIS, unless the default format is changed by the CLI keyword DateTimeStringFormat.

For the IBM Data Server Driver for JDBC and SQLJ, the default format for retrieval of DATE, TIME, or TIMESTAMP column data into strings is ISO, unless the default format is changed by the dateFormat or timeFormat Connection or DataSource property.

When an exception is thrown for PreparedStatement.setXXXStream with a length mismatch

When you use the PreparedStatement.setBinaryStream, PreparedStatement.setCharacterStream, or PreparedStatement.setUnicodeStream method, the length parameter value must match the number of bytes in the input stream.

If the numbers of bytes do not match, the IBM Data Server Driver for JDBC and SQLJ does not throw an exception until the subsequent PreparedStatement.executeUpdate method executes. Therefore, for the IBM Data Server Driver for JDBC and SQLJ, some data might be sent to the server when the lengths to not match. That data is truncated or padded by the server. The calling application needs to issue a rollback request to undo the database updates that include the truncated or padded data.

The DB2 JDBC Type 2 Driver throws an exception after the PreparedStatement.setBinaryStream, PreparedStatement.setCharacterStream, or PreparedStatement.setUnicodeStream method executes.

Default mappings for PreparedStatement.setXXXStream

With the IBM Data Server Driver for JDBC and SQLJ, when you use the PreparedStatement.setBinaryStream, PreparedStatement.setCharacterStream, or PreparedStatement.setUnicodeStream method, and no information about the data type of the target column is available, the input data is mapped to a BLOB or CLOB data type.

For the DB2 JDBC Type 2 Driver, the input data is mapped to a VARCHAR FOR BIT DATA or VARCHAR data type.

Differences in character conversion

When character data is transferred between a client and a server, the data must be converted to a form that the receiver can process.

For the IBM Data Server Driver for JDBC and SQLJ, character data that is sent from the data source to the client is converted using Java's built-in character converters. The conversions that the IBM Data Server Driver for JDBC and SQLJ supports are limited to those that are supported by the underlying JRE implementation.
A IBM Data Server Driver for JDBC and SQLJ client using type 4 connectivity sends data to the data source as Unicode UTF-8.

For the DB2 JDBC Type 2 Driver, character conversions can be performed if the conversions are supported by the DB2 server.

If a GRAPHIC column contains single-byte characters (an error condition), the DB2 JDBC Type 2 Driver performs character conversion during data retrieval without issuing an error, but the results might be incorrect. The IBM Data Server Driver for JDBC and SQLJ issues an error.

If character data other than ASCII data is retrieved from a CLOB column with getAsciiStream, the DB2 JDBC Type 2 Driver performs character conversion correctly during data retrieval. The IBM Data Server Driver for JDBC and SQLJ does not perform character conversion correctly. getAsciiStream should be used only for retrieval of ASCII data. For other types of character data, use getCharacterStream or getString instead.

Those drivers use CCSID information from the data source if it is available. The drivers convert input parameter data to the CCSID of the data source before sending the data. If target CCSID information is not available, the drivers send the data as Unicode UTF-8.

**Implicit or explicit data type conversion for input parameters**

If you execute a PreparedStatement.setXXX method, and the resulting data type from the setXXX method does not match the data type of the table column to which the parameter value is assigned, the driver returns an error unless data type conversion occurs.

With the IBM Data Server Driver for JDBC and SQLJ, conversion to the correct SQL data type occurs implicitly if the target data type is known and if the deferPrepares and sendDataAsIs connection properties are set to false. In this case, the implicit values override any explicit values in the setXXX call. If the deferPrepares connection property or the sendDataAsIs connection property is set to true, you must use the PreparedStatement.setObject method to convert the parameter to the correct SQL data type.

For the DB2 JDBC Type 2 Driver, if the data type of a parameter does not match its default SQL data type, you must use the PreparedStatement.setObject method to convert the parameter to the correct SQL data type.

**Support for String to BINARY conversions for input parameters**

The IBM Data Server Driver for JDBC and SQLJ does not support PreparedStatement.setObject calls of the following form when x is an object of type String:

```
setObject(parameterIndex, x, java.sqlTypes.BINARY)
```

The DB2 JDBC Type 2 Driver supports calls of this type. The driver interprets the value of x as a hexadecimal string.
Result of PreparedStatement.setObject with a decimal scale mismatch

With the IBM Data Server Driver for JDBC and SQLJ, if you call PreparedStatement.setObject with a decimal input parameter, and the scale of the input parameter is greater than the scale of the target column, the driver truncates the trailing digits of the input value before assigning the value to the column.

The DB2 JDBC Type 2 Driver rounds the trailing digits of the input value before assigning the value to the column.

Valid range for ResultSet.getBigDecimal scale parameter

The deprecated form of ResultSet.getBigDecimal has a scale parameter as the second parameter. The IBM Data Server Driver for JDBC and SQLJ allows a range of 0 to 32 for the scale parameter.

The DB2 JDBC Type 2 Driver allows a range of -1 to 32.

Support for conversions from the java.lang.Character data type for input parameters

For the following form of PreparedStatement.setObject, the IBM Data Server Driver for JDBC and SQLJ supports the standard data type mappings of Java objects to JDBC data types when it converts to a JDBC data type:

```
setObject(parameterIndex, x)
```

The DB2 JDBC Type 2 Driver supports the non-standard mapping of from java.lang.Character to CHAR.

Support for ResultSet.getBinaryStream against a character column

The IBM Data Server Driver for JDBC and SQLJ supports ResultSet.getBinaryStream with an argument that represents a character column only if the column has the FOR BIT DATA attribute.

For the DB2 JDBC Type 2 Driver, if the ResultSet.getBinaryStream argument is a character column, that column does not need to have the FOR BIT DATA attribute.

Data returned from ResultSet.getString against a binary column

With the IBM Data Server Driver for JDBC and SQLJ, when you execute ResultSet.getString against a binary column, the returned data is in the form of lowercase, hexadecimal digit pairs.

With the DB2 JDBC Type 2 Driver, when you execute ResultSet.getString against a binary column, the returned data is in the form of uppercase, hexadecimal digit pairs.

Result of using setObject with a Boolean input type and a CHAR target type

With the IBM Data Server Driver for JDBC and SQLJ, when you execute PreparedStatement.setObject(parameterIndex, x, CHAR), and x is Boolean, the value "0" or "1" is inserted into the table column.
With the DB2 JDBC Type 2 Driver, the string "false" or "true" is inserted into the table column. The table column length must be at least 5.

**Result of using getBoolean to retrieve a value from a CHAR column**

With the IBM Data Server Driver for JDBC and SQLJ, when you execute ResultSet.getBoolean or CallableStatement.getBoolean to retrieve a Boolean value from a CHAR column, and the column contains the value "false" or "0", the value false is returned. If the column contains any other value, true is returned.

With the DB2 JDBC Type 2 Driver, when you execute ResultSet.getBoolean or CallableStatement.getBoolean to retrieve a Boolean value from a CHAR column, and the column contains the value "true" or "1", the value true is returned. If the column contains any other value, false is returned.

**Result of executing ResultSet.next on a closed cursor**

With the IBM Data Server Driver for JDBC and SQLJ, when all rows have been retrieved from a ResultSet, the cursor is automatically closed. When you execute ResultSet.next after the cursor is closed, an SQLException is thrown. This behavior conforms with the JDBC standard.

With the DB2 JDBC Type 2 Driver, when all rows have been retrieved from a ResultSet, the cursor is not closed. When ResultSet.next is executed after all rows have been retrieved, a value of false is returned, and no exception is thrown.

**Result of specifying null arguments in DatabaseMetaData calls**

With the IBM Data Server Driver for JDBC and SQLJ, you can specify null for an argument in a DatabaseMetaData method call only where the JDBC specification states that null is allowed. Otherwise, an exception is thrown.

With the DB2 JDBC Type 2 Driver, null means that the argument is not used to narrow the search.

**Folding of method arguments to uppercase**

The IBM Data Server Driver for JDBC and SQLJ does not fold any arguments in method calls to uppercase.

The DB2 JDBC Type 2 Driver folds the argument of a Statement.setCursorName call to uppercase. To prevent the cursor name from being folded to uppercase, precede and follow the cursor name with the characters ". For example:

```
Statement.setCursorName("\"mycursor\"");
```

**Support for timestamp escape clauses**

The IBM Data Server Driver for JDBC and SQLJ supports the standard form of an escape clause for TIME:

```
{t 'hh:mm:ss'}
```

In addition to the standard form, the DB2 JDBC Type 2 Driver supports the following form of a TIME escape clause:

```
{ts 'hh:mm:ss'}
```
Including a CALL statement in a statement batch

The IBM Data Server Driver for JDBC and SQLJ supports CALL statements in a statement batch.

The DB2 JDBC Type 2 Driver does not support CALL statements in a statement batch.

Removal of extra characters from SQL statement text

The IBM Data Server Driver for JDBC and SQLJ does not remove white-space characters, such as spaces, tabs, and new-line characters, from SQL statement text before it passes that text to the data source.

The DB2 JDBC Type 2 Driver removes white-space characters from SQL statement text before it passes that text to the data source.

Result of executing PreparedStatement.executeUpdateBatch

When a PreparedStatement.executeUpdateBatch statement is executed under the IBM Data Server Driver for JDBC and SQLJ, the driver returns an int array of update counts. Each element of the array contains the number of rows that were updated by a statement in the batch.

When a PreparedStatement.executeUpdateBatch statement is executed under the DB2 JDBC Type 2 Driver, the driver cannot determine the update counts, so it returns -3 for each update count.

Support for compound SQL

The IBM Data Server Driver for JDBC and SQLJ driver does not support compound SQL blocks.

Compound SQL allows multiple SQL statements to be grouped into a single executable block. For example:

```sql
EXEC SQL BEGIN COMPOUND ATOMIC STATIC
  UPDATE ACCOUNTS SET ABALANCE = ABALANCE + :delta
  WHERE AID = :aid;
  UPDATE TELLERS SET TBALANCE = TBALANCE + :delta
  WHERE TID = :tid;
  INSERT INTO TELLERS (TID, BID, TBALANCE) VALUES (:i, :branch_id, 0);
  COMMIT;
END COMPOUND;
```

The DB2 JDBC Type 2 Driver supports execution of compound SQL blocks with PreparedStatement.executeUpdate or Statement.executeUpdate.

Result of not setting a parameter in a batched update

The IBM Data Server Driver for JDBC and SQLJ driver throws an exception after a PreparedStatement.addBatch call if a parameter is not set.

The DB2 JDBC Type 2 Driver throws an exception after the PreparedStatement.executeUpdateBatch call if a parameter is not set for any of the statements in the batch.
Ability to call uncatalogued stored procedures

The IBM Data Server Driver for JDBC and SQLJ driver does not let you call stored procedures that are not defined in the DB2 catalog.

The DB2 JDBC Type 2 Driver lets you call stored procedures that are not defined in the DB2 catalog.

Specification of data types for stored procedure parameters

With the IBM Data Server Driver for JDBC and SQLJ driver, if the data source does not support dynamic execution of the CALL statement, you must specify CALL statement parameters exactly as they are specified in the stored procedure definition.

For example, DB2 for z/OS data sources do not support dynamic execution of CALL statements. Suppose that the first parameter of a stored procedure on a DB2 for z/OS server is defined like this in the CREATE PROCEDURE statement:

```
OUT PARM1 DECIMAL(3,0)
```

In the calling application, a statement like `cs.registerOutParameter(1, Types.DECIMAL)` is not correct. You need to use the form of the `registerOutParameter` method that specifies the scale as well as the data type: `cs.registerOutParameter (1, Types.DECIMAL, 0)`.

The DB2 JDBC Type 2 Driver does not require that the parameter data types in a calling application match the data types in the CREATE PROCEDURE statement.

Result of preparation of an SQL statement with a semicolon

When the IBM Data Server Driver for JDBC and SQLJ prepares an SQL statement, it sends the statement to the data server with no extra processing. Therefore, if you prepare an SQL statement that ends with a semicolon with the IBM Data Server Driver for JDBC and SQLJ, you receive an error.

When the DB2 JDBC Type 2 Driver prepares an SQL statement that contains a final semicolon, it strips the semicolon. Therefore, if the SQL statement has valid syntax, it executes successfully.

For example, this code fails under the IBM Data Server Driver for JDBC and SQLJ but executes successfully under the DB2 JDBC Type 2 Driver.

```
PreparedStatement pstmt1 = con.prepareStatement("SELECT c1,c2 FROM testtab;");
```

Maximum number of concurrently open Statement or PreparedStatement objects

The number of Statement or PreparedStatement objects that you can leave open concurrently is greater for the DB2 Type 2 Driver for Linux, UNIX, and Windows than for the IBM Data Server Driver for JDBC and SQLJ. With the IBM Data Server Driver for JDBC and SQLJ, you might receive error code -805 after you have opened fewer Statement or PreparedStatement objects than with the DB2 Type 2 Driver for Linux, UNIX, and Windows. The difference is due to the way that the drivers handle the underlying resources for those objects.

**Recommendation:** Close Statement or PreparedStatement objects when you have finished using them.
JDBC differences between versions of the IBM Data Server Driver for JDBC and SQLJ

Before you can upgrade your JDBC applications from older to newer versions of the IBM Data Server Driver for JDBC and SQLJ, you need to understand the differences between those drivers.

Supported methods

For a list of methods that the IBM Data Server Driver for JDBC and SQLJ supports, see "Driver support for JDBC APIs".

Use of progressive streaming by the JDBC drivers

For IBM Data Server Driver for JDBC and SQLJ, Version 3.50 and later, progressive streaming, which is also known as dynamic data format, behavior is the default for LOB retrieval, for connections to DB2 Database for Linux, UNIX, and Windows Version 9.5 and later.

Progressive streaming is supported in the IBM Data Server Driver for JDBC and SQLJ Version 3.1 and later, but for IBM Data Server Driver for JDBC and SQLJ version 3.2 and later, progressive streaming behavior is the default for LOB and XML retrieval, for connections to DB2 for z/OS Version 9.1 and later.

Previous versions of the IBM Data Server Driver for JDBC and SQLJ did not support progressive streaming.

Important: With progressive streaming, when you retrieve a LOB or XML value from a ResultSet into an application variable, you can manipulate the contents of that application variable until you move the cursor or close the cursor on the ResultSet. After that, the contents of the application variable are no longer available to you. If you perform any actions on the LOB in the application variable, you receive an SQLException. For example, suppose that progressive streaming is enabled, and you execute statements like this:

```java
... 
ResultSet rs = stmt.executeQuery("SELECT CLOBCOL FROM MY_TABLE");
rs.next(); // Retrieve the first row of the ResultSet
Clob clobFromRow1 = rs.getClob(1);
    // Put the CLOB from the first column of
    // the first row in an application variable
String substr1Clob = clobFromRow1.getSubString(1,50);
    // Retrieve the first 50 bytes of the CLOB
rs.next(); // Move the cursor to the next row.
    // clobFromRow1 is no longer available.
// String substr2Clob = clobFromRow1.getSubString(51,100);
    // This statement would yield an SQLException
Clob clobFromRow2 = rs.getClob(1);
    // Put the CLOB from the first column of
    // the second row in an application variable
rs.close(); // Close the ResultSet.
    // clobFromRow2 is also no longer available.
```

After you execute rs.next() to position the cursor at the second row of the ResultSet, the CLOB value in clobFromRow1 is no longer available to you. Similarly, after you execute rs.close() to close the ResultSet, the values in clobFromRow1 and clobFromRow2 are no longer available.
To avoid errors that are due to this changed behavior, you need to take one of the following actions:

- Modify your applications.
  
  Applications that retrieve LOB data into application variables can manipulate the data in those application variables only until the cursors that were used to retrieve the data are moved or closed.
- Disable progressive streaming by setting the `progressiveStreaming` property to `DB2BaseDataSource.NO` (2).

**ResultSetMetaData values for IBM Data Server Driver for JDBC and SQLJ version 4.0 and later**

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` differs from the default behavior for earlier JDBC drivers.

If you need to use IBM Data Server Driver for JDBC and SQLJ version 4.0 or later, but your applications need to return the `ResultSetMetaData.getColumnName` and `ResultSetMetaData.getColumnLabel` values that were returned with older JDBC drivers, you can set the `useJDBC4ColumnNameAndLabelSemantics` Connection and `DataSource` property to `DB2BaseDataSource.NO` (2).

**Batch updates with automatically generated keys have different results in different driver versions**

With the IBM Data Server Driver for JDBC and SQLJ version 3.52 or later, preparing an SQL statement for retrieval of automatically generated keys is supported.

With the IBM Data Server Driver for JDBC and SQLJ version 3.50 or version 3.51, preparing an SQL statement for retrieval of automatically generated keys and using the `PreparedStatement` object for batch updates causes an `SQLException`.

Versions of the IBM Data Server Driver for JDBC and SQLJ before Version 3.50 do not throw an `SQLException` when an application calls the `addBatch` or `executeBatch` method on a `PreparedStatement` object that is prepared to return automatically generated keys. However, the `PreparedStatement` object does not return automatically generated keys.

**Batch updates of data on DB2 for z/OS servers have different results in different driver versions**

After you successfully invoke an `executeBatch` statement, the IBM Data Server Driver for JDBC and SQLJ returns an array. The purpose of the array is to indicate the number of rows that are affected by each SQL statement that is executed in the batch.

If the following conditions are true, the IBM Data Server Driver for JDBC and SQLJ returns `Statement.SUCCESS_NO_INFO` (-2) in the array elements:

- The application is connected to a subsystem that is in DB2 for z/OS Version 8 new-function mode, or later.
- The application is using Version 3.1 or later of the IBM Data Server Driver for JDBC and SQLJ.
• The IBM Data Server Driver for JDBC and SQLJ uses multi-row INSERT operations to execute batch updates.

This occurs because with multi-row INSERT, the database server executes the entire batch as a single operation, so it does not return results for individual SQL statements.

If you are using an earlier version of the IBM Data Server Driver for JDBC and SQLJ, or you are connected to a data source other than DB2 for z/OS Version 8 or later, the array elements contain the number of rows that are affected by each SQL statement.

**Batch updates and deletes of data on DB2 for z/OS servers have different size limitations in different driver versions**

Before IBM Data Server Driver for JDBC and SQLJ version 3.59 or 4.9, a DisconnectException with error code -4499 was thrown for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS if the size of an update or delete batch was greater than 32KB. Starting with version 3.59 or 4.9, this restriction no longer exists, and the exception is no longer thrown.

**Initial value of the CURRENT CLIENT_ACCTNG special register**

For a JDBC or SQLJ application that runs under the IBM Data Server Driver for JDBC and SQLJ version 2.6 or later, using type 4 connectivity, the initial value for the DB2 for z/OS CURRENT CLIENT_ACCTNG special register is the concatenation of the DB2 for z/OS version and the value of the clientWorkStation property. For any other JDBC driver, version, and connectivity, the initial value is not set.

**Properties that control the use of multi-row FETCH**

Before version 3.7 and version 3.51 of the IBM Data Server Driver for JDBC and SQLJ, multi-row FETCH support was enabled and disabled through the useRowsetCursor property, and was available only for scrollable cursors, and for IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS. Starting with version 3.7 and 3.51:

• For IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, the IBM Data Server Driver for JDBC and SQLJ uses only the enableRowsetSupport property to determine whether to use multi-row FETCH for scrollable or forward-only cursors.

• For IBM Data Server Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS or DB2 Database for Linux, UNIX, and Windows, or IBM Data Server Driver for JDBC and SQLJ type 2 connectivity on DB2 Database for Linux, UNIX, and Windows, the IBM Data Server Driver for JDBC and SQLJ uses the enableRowsetSupport property to determine whether to use multi-row FETCH for scrollable cursors, if enableRowsetSupport is set. If enableRowsetSupport is not set, the driver uses the useRowsetCursor property to determine whether to use multi-row FETCH.

**JDBC 1 positioned updates and deletes and multi-row FETCH**

Before version 3.7 and version 3.51 of the IBM Data Server Driver for JDBC and SQLJ, multi-row FETCH from DB2 for z/OS tables was controlled by the useRowsetCursor property. If an application contained JDBC 1 positioned update
or delete operations, and multi-row FETCH support was enabled, the IBM Data Server Driver for JDBC and SQLJ permitted the update or delete operations, but unexpected updates or deletes might occur.

Starting with version 3.7 and 3.51 of the IBM Data Server Driver for JDBC and SQLJ, the enableRowSetSupport property enables or disables multi-row FETCH from DB2 for z/OS tables or DB2 Database for Linux, UNIX, and Windows tables. The enableRowSetSupport property overrides the useRowsetCursor property. If multi-row FETCH is enabled through the enableRowSetSupport property, and an application contains a JDBC 1 positioned update or delete operation, the IBM Data Server Driver for JDBC and SQLJ throws an SQLException.

**Valid forms of prepareStatement for retrieval of automatically generated keys from a DB2 for z/OS view**

Starting with version 3.57 or version 4.7 of the IBM Data Server Driver for JDBC and SQLJ, if you are inserting data into a view on a DB2 for z/OS data server, and you want to retrieve automatically generated keys, you need to use one of the following methods to prepare the SQL statement that inserts rows into the view:

- `Connection.prepareStatement(sql-statement, String [] columnNames);`
- `Connection.prepareStatement(sql-statement, int [] columnIndexes);`
- `Statement.executeUpdate(sql-statement, String [] columnNames);`
- `Statement.executeUpdate(sql-statement, int [] columnIndexes);`

**Examples of ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel values**

For the IBM Data Server Driver for JDBC and SQLJ version 4.0 and later, the default behavior of ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel differs from the default behavior for earlier JDBC drivers. You can use the useJDBC4ColumnNameAndLabelSemantics property to change this behavior.

The following examples show the values that are returned for IBM Data Server Driver for JDBC and SQLJ Version 4.0, and for previous JDBC drivers, when the useJDBC4ColumnNameAndLabelSemantics property is not set.

All queries use a table that is defined like this:

```sql
CREATE TABLE MYTABLE(INTCOL INT)
```

**Example:** The following query contains an AS CLAUSE, which defines a label for a column in the result set:

```sql
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

The following table lists the ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel values that are returned for the query:
### Table 110. ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel before and after IBM Data Server Driver for JDBC and SQLJ Version 4.0 for a query with an AS CLAUSE

<table>
<thead>
<tr>
<th>Target data source</th>
<th>Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
<th>Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getColumnLabel value</td>
<td>getColumnLabel value</td>
</tr>
<tr>
<td>DB2 Database for Linux, UNIX, and Windows</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
<tr>
<td>IBM Informix</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 8 or later, and DB2 UDB for iSeries V5R3 and later</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 7, and DB2 UDB for iSeries V5R2</td>
<td>MYLABEL</td>
<td>MYLABEL</td>
</tr>
</tbody>
</table>

**Example:** The following query contains no AS clause:

```
SELECT MYCOL FROM MYTABLE
```

The ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel methods on the query return MYCOL, regardless of the target data source.

**Example:** On a DB2 for z/OS or DB2 for i data source, a LABEL ON statement is used to define a label for a column:

```
LABEL ON COLUMN MYTABLE.MYCOL IS 'LABELONCOL'
```

The following query contains an AS CLAUSE, which defines a label for a column in the ResultSet:

```
SELECT MYCOL AS MYLABEL FROM MYTABLE
```

The following table lists the ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel values that are returned for the query.

### Table 111. ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel before and after IBM Data Server Driver for JDBC and SQLJ Version 4.0 for a table column with a LABEL ON statement in a query with an AS CLAUSE

<table>
<thead>
<tr>
<th>Target data source</th>
<th>Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
<th>Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0 and later</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getColumnLabel value</td>
<td>getColumnLabel value</td>
</tr>
<tr>
<td>DB2 for z/OS Version 8 or later, and DB2 UDB for iSeries V5R3 and later</td>
<td>MYLABEL</td>
<td>LABELONCOL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 7, and DB2 UDB for iSeries V5R2</td>
<td>MYLABEL</td>
<td>LABELONCOL</td>
</tr>
</tbody>
</table>

**Example:** On a DB2 for z/OS or DB2 for i data source, a LABEL ON statement is used to define a label for a column:
LABEL ON COLUMN MYTABLE.MYCOL IS ‘LABELONCOL’

The following query contains no AS CLAUSE:
SELECT MYCOL FROM MYTABLE

The following table lists the ResultSetMetaData.getColumnName and ResultSetMetaData.getColumnLabel values that are returned for the query.

<table>
<thead>
<tr>
<th>Target data source</th>
<th>Behavior before IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
<th>Behavior for IBM Data Server Driver for JDBC and SQLJ Version 4.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>getColumnName value</td>
<td>getColumnLabel value</td>
</tr>
<tr>
<td>DB2 for z/OS Version 8 or later, and DB2 UDB for i5/OS V5R3 and later</td>
<td>MYCOL</td>
<td>LABELONCOL</td>
</tr>
<tr>
<td>DB2 for z/OS Version 7, and DB2 UDB for i5/OS V5R2</td>
<td>MYCOL</td>
<td>LABELONCOL</td>
</tr>
</tbody>
</table>

### SQLJ differences between the IBM Data Server Driver for JDBC and SQLJ and other DB2 JDBC drivers

The IBM Data Server Driver for JDBC and SQLJ differs in a number of ways from older JDBC drivers. When you move to the IBM Data Server Driver for JDBC and SQLJ, you need to modify your SQLJ programs to account for those differences.

**Important:** The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) is deprecated. This information is provided to assist you in moving your applications to the IBM Data Server Driver for JDBC and SQLJ.

SQLJ support in the IBM Data Server Driver for JDBC and SQLJ differs from SQLJ support in the other DB2 JDBC drivers in the following areas:

#### db2sqliconfigure errors and the -collection parameter

The db2sqljconfigure utility that is part of the IBM Data Server Driver for JDBC and SQLJ has a -collection parameter. The db2profc utility that is part of the DB2 JDBC Type 2 Driver does not have a -collection parameter. If the db2sqljconfigure utility performs a bind operation on a DB2 for z/OS server, and the -collection parameter contains any lowercase characters, db2sqljconfigure returns a -4499 error because collection IDs cannot contain lowercase characters in DB2 for z/OS. This situation cannot occur with db2profc.

#### Differences in serialized profiles

The DB2 JDBC Type 2 Driver and the IBM Data Server Driver for JDBC and SQLJ produce different binary code when you execute their SQLJ translator and the SQLJ customizer utilities. Therefore, SQLJ applications that you translated and customized using the DB2 JDBC Type 2 Driver sqlj and db2profc utilities do not run under the IBM Data Server Driver for JDBC and SQLJ. Before you can run
those SQLJ applications under the IBM Data Server Driver for JDBC and SQLJ, you must retranslate and recustomize the applications using the IBM Data Server Driver for JDBC and SQLJ sqlj and db2sqljcustomize utilities. You must do so even if you have not modified the applications.

**SQL VALUES support**

The DB2 JDBC Type 2 Driver supports the SQL VALUES statement in an SQLJ statement clause, but the IBM Data Server Driver for JDBC and SQLJ does not. Therefore, you need to modify your SQLJ applications that include VALUES statements.

**Example:** Suppose that an SQLJ program contains the following statement:

```sql
#sql [ctxt] hv = {VALUES (MY_ROUTINE(1))};
```

For the IBM Data Server Driver for JDBC and SQLJ, you need to change that statement to something like this:

```sql
#sql [ctxt] {SELECT MY_ROUTINE(1) INTO :hv FROM SYSIBM.SYSDUMMY1};
```

**Difference in connection techniques**

The connection techniques that are available, and the driver names and URLs that are used for those connection techniques, vary from driver to driver. See “Connect to a data source using SQLJ” for more information.

**Support for scrollable and updatable iterators**

SQLJ with the IBM Data Server Driver for JDBC and SQLJ supports scrollable and updatable iterators.

The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver) supports scrollable cursors but not updatable iterators.

**Dynamic execution of SQL statements under WebSphere Application Server**

For WebSphere Application Server Version 5.0.1 and above, if you customize your SQLJ program, SQL statements are executed statically.

**Alternative names for db2sqljcustomize and db2sqljprint are not supported**

The DB2 JDBC Type 2 Driver originally used the name db2profc for the SQLJ profile customizer command, and the name db2profp for the SQLJ profile printer command. For the IBM Data Server Driver for JDBC and SQLJ, the SQLJ profile customizer command is named db2sqljcustomize, and the SQLJ profile printer command is named db2sqljprint. In previous releases of DB2 Database for Linux, UNIX, and Windows, db2profc was accepted as an alternative name for db2sqljcustomize, and db2profp was accepted as an alternative name for db2sqljprint. These alternative names are no longer accepted.
SDK for Java differences that affect the IBM Data Server Driver for JDBC and SQLJ

Differences in the behavior among versions of the SDK for Java can cause variations in the results that you receive when you run programs under the IBM Data Server Driver for JDBC and SQLJ.

Retrieved values for DBCS substitution characters

When you retrieve a DBCS substitution character, such as X'FCFC' in code page Cp943, from a database table, the retrieved value differs, depending on whether you are using an IBM SDK for Java or a Sun SDK for Java.

For a Sun SDK for Java, the substitution character is retrieved as U+0000. For an IBM SDK for Java, the substitution character is retrieved as X'FFFD'.

Supported code pages

IBM SDKs for Java support more DBCS code pages than Sun SDKs for Java. Therefore, if you get errors because of unsupported code pages with a Sun SDK for Java, try using an IBM SDK for Java.

IBM SDK for Java requirement for encryption

The IBM SDKs for Java support 256-bit encryption, but the Sun SDKs for Java do not have this support. Therefore, if you use any of the IBM Data Server Driver for JDBC and SQLJ security mechanisms that include encryption, you need to use an IBM SDK for Java.

Support for system monitoring

Support for system monitoring in the IBM Data Server Driver for JDBC and SQLJ includes collection of core driver time and network I/O time. Retrieval of this information requires capabilities that are in any SDK for Java Version 5 or later. However, the IBM SDK for Java Version 1.4.2 also has support that enables collection of core driver time and network I/O time. If you use the IBM SDK for Java Version 1.4.2, the core driver time and network I/O time are rounded to the nearest microsecond. If you use an SDK for Java Version 5 or later, the core driver time and network I/O time are rounded to the nearest nanosecond.

Error codes issued by the IBM Data Server Driver for JDBC and SQLJ

Error codes in the ranges +4200 to +4299, +4450 to +4499, -4200 to -4299, and -4450 to -4499 are reserved for the IBM Data Server Driver for JDBC and SQLJ.

When you call the SQLException.getMessage method after a IBM Data Server Driver for JDBC and SQLJ error occurs, a string is returned that includes:

- Whether the connection is a type 2 or type 4 connection
- Diagnostic information for IBM Software Support
- The level of the driver
- An explanatory message
- The error code
- The SQLSTATE

For example:
Currently, the IBM Data Server Driver for JDBC and SQLJ issues the following error codes:

Table 113. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Message text and explanation</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+4204</td>
<td>Errors were encountered and tolerated as specified by the RETURN DATA UNTIL clause.</td>
<td>02506</td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> Tolerated errors include federated connection, authentication, and authorization errors. This warning applies only to connections to DB2 Database for Linux, UNIX, and Windows servers. It is issued only when a cursor operation, such as a ResultSet.next or ResultSet.previous call, returns false.</td>
<td></td>
</tr>
<tr>
<td>+4222</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> A warning condition occurred during connection to the data source.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4223</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> A warning condition occurred during initialization.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4225</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> A warning condition occurred when data was sent to a server or received from a server.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4226</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> A warning condition occurred during customization or bind.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4228</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> An warning condition occurred that does not fit in another category.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4450</td>
<td>Feature not supported: feature-name</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Message text and explanation</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>+4460</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> The specified value is not a valid option.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4461</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> The specified value is invalid or out of range.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4462</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> A required value is missing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4470</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> The requested operation cannot be performed because the target resource is closed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4471</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> The requested operation cannot be performed because the target resource is in use.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4472</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> The requested operation cannot be performed because the target resource is unavailable.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>+4474</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> The requested operation cannot be performed because the target resource cannot be changed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4200</td>
<td>Invalid operation: An invalid COMMIT or ROLLBACK has been called in an XA environment during a Global Transaction.</td>
<td>2D521</td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> An application that was in a global transaction in an XA environment issued a commit or rollback. A commit or rollback operation in a global transaction is invalid.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 113. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Message text and explanation</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4201</td>
<td>Invalid operation: setAutoCommit(true) is not allowed during a Global Transaction.</td>
<td>2D521</td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: An application that was in a global transaction in an XA environment executed the setAutoCommit(true) statement. Issuing setAutoCommit(true) in a global transaction is invalid.</td>
<td></td>
</tr>
<tr>
<td>-4203</td>
<td>Error executing function. Server returned rc.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: An error occurred on an XA connection during execution of an SQL statement. For network optimization, the IBM Data Server Driver for JDBC and SQLJ delays some XA flows until the next SQL statement is executed. If an error occurs in a delayed XA flow, that error is reported as part of the SQLException that is thrown by the current SQL statement.</td>
<td></td>
</tr>
<tr>
<td>-4210</td>
<td>Timeout getting a transport object from pool.</td>
<td>57033</td>
</tr>
<tr>
<td>-4211</td>
<td>Timeout getting an object from pool.</td>
<td>57033</td>
</tr>
<tr>
<td>-4212</td>
<td>Sysplex member unavailable.</td>
<td></td>
</tr>
<tr>
<td>-4213</td>
<td>Timeout.</td>
<td>57033</td>
</tr>
<tr>
<td>-4214</td>
<td><code>text-from-getMessage</code></td>
<td>28000</td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: Authorization failed. <strong>User response</strong>: Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4220</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: An error occurred during character conversion. <strong>User response</strong>: Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4221</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: An error occurred during encryption or decryption. <strong>User response</strong>: Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4222</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: An error occurred during connection to the data source. <strong>User response</strong>: Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4223</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: An error occurred during initialization. <strong>User response</strong>: Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Message text and explanation</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>-4224</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> An error occurred during resource cleanup.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4225</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> An error occurred when data was sent to a server or received from a server.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4226</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> An error occurred during customization or bind.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4227</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> An error occurred during reset.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4228</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> An error occurred that does not fit in another category.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4229</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> An error occurred during a batch execution.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4450</td>
<td>Feature not supported: feature-name</td>
<td>0A504</td>
</tr>
<tr>
<td>-4460</td>
<td>text-from-getMessage</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> The specified value is not a valid option.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4461</td>
<td>text-from-getMessage</td>
<td>42815</td>
</tr>
<tr>
<td></td>
<td><strong>Explaination:</strong> The specified value is invalid or out of range.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call SQLException.getMessage to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Message text and explanation</td>
<td>SQLSTATE</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>-4462</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: A required value is missing.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4463</td>
<td><code>text-from-getMessage</code></td>
<td>42601</td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: The specified value has a syntax error.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4470</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: The requested operation cannot be performed because the target resource is closed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4471</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: The requested operation cannot be performed because the target resource is in use.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4472</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: The requested operation cannot be performed because the target resource is unavailable.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4473</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: The requested operation cannot be performed because the target resource is no longer available.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4474</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: The requested operation cannot be performed because the target resource cannot be changed.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4475</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation</strong>: The requested operation cannot be performed because access to the target resource is restricted.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response</strong>: Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 113. Error codes issued by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Message text and explanation</th>
<th>SQLSTATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4476</td>
<td><code>text-from-getMessage</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> The requested operation cannot be performed because the operation is not allowed on the target resource.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-4496</td>
<td>An SQL OPEN for a held cursor was issued on an XA connection. The JDBC driver does not allow a held cursor to be opened on the database server for an XA connection.</td>
<td></td>
</tr>
<tr>
<td>-4497</td>
<td>The application must issue a rollback. The unit of work has already been rolled back in the DB2 server, but other resource managers involved in the unit of work might not have rolled back their changes. To ensure integrity of the application, all SQL requests are rejected until the application issues a rollback.</td>
<td></td>
</tr>
<tr>
<td>-4498</td>
<td>A connection failed but has been reestablished. Host name or IP address: <code>host-name</code>, service name or port number: <code>port</code>, special register modification indicator: <code>rc</code>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> <code>host-name</code> and <code>port</code> indicate the data source at which the connection is reestablished. <code>rc</code> indicates whether SQL statements that set special register values were executed again:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 SQL statements that set special register values were executed again.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 SQL statements that set special register values might not have been executed again.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For client reroute against DB2 for z/OS servers, special register values that were set after the last commit point are not re-established.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The application is rolled back to the previous commit point. The connection state and global resources such as global temporary tables and open held cursors might not be maintained.</td>
<td></td>
</tr>
<tr>
<td>-4499</td>
<td><code>text-from-getMessage</code></td>
<td>08001 or 58009</td>
</tr>
<tr>
<td></td>
<td><strong>Explanation:</strong> A fatal error occurred that resulted in a disconnect from the data source. The existing connection has become unusable.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>User response:</strong> Call <code>SQLException.getMessage</code> to retrieve specific information about the problem.</td>
<td></td>
</tr>
<tr>
<td>-30108</td>
<td>Client reroute exception for the Sysplex.</td>
<td>08506</td>
</tr>
<tr>
<td>-99999</td>
<td>The IBM Data Server Driver for JDBC and SQLJ issued an error that does not yet have an error code.</td>
<td></td>
</tr>
</tbody>
</table>
SQLSTATEs issued by the IBM Data Server Driver for JDBC and SQLJ

SQLSTATEs in the range 46600 to 466ZZ are reserved for the IBM Data Server Driver for JDBC and SQLJ.

The following table lists the SQLSTATEs that are generated or used by the IBM Data Server Driver for JDBC and SQLJ.

Table 114. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ

<table>
<thead>
<tr>
<th>SQLSTATE class</th>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>01xxx</td>
<td>Warning</td>
<td></td>
</tr>
<tr>
<td>02xxx</td>
<td>No data</td>
<td></td>
</tr>
<tr>
<td>02501</td>
<td></td>
<td>The cursor position is not valid for a FETCH of the current row.</td>
</tr>
<tr>
<td>02506</td>
<td>Tolerable error</td>
<td></td>
</tr>
<tr>
<td>08xxx</td>
<td>Connection exception</td>
<td></td>
</tr>
<tr>
<td>08001</td>
<td>The application requester is unable to establish the connection.</td>
<td></td>
</tr>
<tr>
<td>08003</td>
<td>A connection does not exist</td>
<td></td>
</tr>
<tr>
<td>08004</td>
<td>The application server rejected establishment of the connection</td>
<td></td>
</tr>
<tr>
<td>08506</td>
<td>Client reroute exception</td>
<td></td>
</tr>
<tr>
<td>0Axxx</td>
<td>Feature not supported</td>
<td></td>
</tr>
<tr>
<td>0A502</td>
<td>The action or operation is not enabled for this database instance</td>
<td></td>
</tr>
<tr>
<td>0A504</td>
<td>The feature is not supported by the driver</td>
<td></td>
</tr>
<tr>
<td>22xxx</td>
<td>Data exception</td>
<td></td>
</tr>
<tr>
<td>22007</td>
<td>The string representation of a datetime value is invalid</td>
<td></td>
</tr>
<tr>
<td>22021</td>
<td>A character is not in the coded character set</td>
<td></td>
</tr>
<tr>
<td>23xxx</td>
<td>Constraint violation</td>
<td></td>
</tr>
<tr>
<td>23502</td>
<td>A value that is inserted into a column or updates a column is null, but the column cannot contain null values.</td>
<td></td>
</tr>
<tr>
<td>24xxx</td>
<td>Invalid cursor state</td>
<td></td>
</tr>
<tr>
<td>24501</td>
<td>The identified cursor is not open</td>
<td></td>
</tr>
<tr>
<td>28xxx</td>
<td>Authorization exception</td>
<td></td>
</tr>
<tr>
<td>28000</td>
<td>Authorization name is invalid.</td>
<td></td>
</tr>
<tr>
<td>2Dxxx</td>
<td>Invalid transaction termination</td>
<td></td>
</tr>
<tr>
<td>2D521</td>
<td>SQL COMMIT or ROLLBACK are invalid in the current operating environment.</td>
<td></td>
</tr>
<tr>
<td>34xxx</td>
<td>Invalid cursor name</td>
<td></td>
</tr>
<tr>
<td>34000</td>
<td>Cursor name is invalid.</td>
<td></td>
</tr>
<tr>
<td>3Bxxx</td>
<td>Invalid savepoint</td>
<td></td>
</tr>
<tr>
<td>3B503</td>
<td>A SAVEPOINT, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT statement is not allowed in a trigger or global transaction.</td>
<td></td>
</tr>
<tr>
<td>40xxx</td>
<td>Transaction rollback</td>
<td></td>
</tr>
</tbody>
</table>
### Table 114. SQLSTATEs returned by the IBM Data Server Driver for JDBC and SQLJ (continued)

<table>
<thead>
<tr>
<th>SQLSTATE class</th>
<th>SQLSTATE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>42xxx</td>
<td>42601</td>
<td>A character, token, or clause is invalid or missing</td>
</tr>
<tr>
<td></td>
<td>42734</td>
<td>A duplicate parameter name, SQL variable name, cursor name, condition name, or label was detected.</td>
</tr>
<tr>
<td></td>
<td>42807</td>
<td>The INSERT, UPDATE, or DELETE is not permitted on this object</td>
</tr>
<tr>
<td></td>
<td>42808</td>
<td>A column identified in the insert or update operation is not updateable</td>
</tr>
<tr>
<td></td>
<td>42815</td>
<td>The data type, length, scale, value, or CCSID is invalid</td>
</tr>
<tr>
<td></td>
<td>42820</td>
<td>A numeric constant is too long, or it has a value that is not within the range of its data type</td>
</tr>
<tr>
<td></td>
<td>42968</td>
<td>The connection failed because there is no current software license.</td>
</tr>
<tr>
<td>57xxx</td>
<td>57033</td>
<td>A deadlock or timeout occurred without automatic rollback</td>
</tr>
<tr>
<td>58xxx</td>
<td>58008</td>
<td>System error</td>
</tr>
<tr>
<td></td>
<td>58009</td>
<td>Execution failed due to a distribution protocol error that caused deallocation of the conversation</td>
</tr>
<tr>
<td></td>
<td>58012</td>
<td>The bind process with the specified package name and consistency token is not active</td>
</tr>
<tr>
<td></td>
<td>58014</td>
<td>The DDM command is not supported</td>
</tr>
<tr>
<td></td>
<td>58015</td>
<td>The DDM object is not supported</td>
</tr>
<tr>
<td></td>
<td>58016</td>
<td>The DDM parameter is not supported</td>
</tr>
<tr>
<td></td>
<td>58017</td>
<td>The DDM parameter value is not supported</td>
</tr>
</tbody>
</table>

### How to find IBM Data Server Driver for JDBC and SQLJ version and environment information

To determine the version of the IBM Data Server Driver for JDBC and SQLJ, as well as information about the environment in which the driver is running, run the DB2Jcc utility on the command line.

**DB2Jcc syntax**

```bash
```
DB2Jcc option descriptions

-version
Specifies that the IBM Data Server Driver for JDBC and SQLJ displays its name and version.

-configuration
Specifies that the IBM Data Server Driver for JDBC and SQLJ displays its name and version, and information about its environment, such as information about the Java runtime environment, operating system, path information, and license restrictions.

-help
Specifies that the DB2Jcc utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

DB2Jcc sample output

The following output is the result of invoking DB2Jcc with the -configuration parameter.

Figure 53. Sample DB2Jcc output

(mymachine) /home/myusrid $ java com.ibm.db2.jcc.DB2Jcc -version
[jcc] Driver: IBM DB2 JDBC Universal Driver Architecture 3.50.137

(mymachine) /home/myusrid $ java com.ibm.db2.jcc.DB2Jcc -configuration
[jcc] BEGIN TRACE_DRIVER_CONFIGURATION
[jcc] Driver: IBM DB2 JDBC Universal Driver Architecture 3.50.137
[jcc] Compatible JRE versions: { 1.4, 1.5 }
[jcc] Target server licensing restrictions: { z/OS: enabled; SQLDS: enabled; iSeries: enabled; DB2 for Unix/Windows: enabled; Cloudscape: enabled; Informix: enabled }
[jcc] Range checking enabled: true
[jcc] Bug check level: 0xff
[jcc] Default fetch size: 64
[jcc] Default isolation: 2
[jcc] Collect performance statistics: false
[jcc] No security manager detected.
[jcc] Access to package sun.io is permitted by security manager.
[jcc] JDBC 1 system property jdbc.drivers = null
[jcc] Java Runtime Environment version 1.4.2
[jcc] Java Runtime Environment vendor = IBM Corporation
[jcc] Java installation directory = /wsdb/v91/bldsupp/AIXSL64/jdk1.4.2_sr1/sh/..
[jre]
[jcc] Java Virtual Machine specification version = 1.0
[jcc] Java Virtual Machine implementation version = 1.4.2
[jcc] Java Virtual Machine implementation vendor = IBM Corporation
[jcc] Java Virtual Machine implementation name = Classic VM
[jcc] Java Runtime Environment specification version = 1.4
[jcc] Java Runtime Environment specification vendor = Sun Microsystems Inc.
[jcc] Java Runtime Environment specification name = Java Platform API Specification
[jcc] Java class format version number = 48.0
[jcc] Java class path = :
/home/myusrid/sql1ib/java/db2java.zip:/lib/classes.zip:/test:/home/myusrid/sql1ib/java/sqlj.zip:/test:/home/myusrid/sql1ib/java/sqlj.jar:/home/myusrid/sql1ib/java/db2jcc_license_cisuz.jar:
[jcc] Java native library path = /wsdb/v91/bldsupp/AIXSL64/jdk1.4.2_sr1/sh/..
Commands for SQLJ program preparation

To prepare SQLJ programs for execution, you use commands to translate SQLJ source code into Java source code, compile the Java source code, create and customize SQLJ serialized profiles, and bind DB2 packages.

sqlj - SQLJ translator

The sqlj command translates an SQLJ source file into a Java source file and zero or more SQLJ serialized profiles. By default, the sqlj command also compiles the Java source file.

Authorization

None

Command syntax

```
sqlj [ -help | -dir=directory | -d=directory | -props=properties-file ]

-compile=true | -compile=false

-encoding=encoding | -db2optimize

-ser2class | -status | -version | -C-help | -C-compiler-option

-JVM-option | -SQLJ-source-file-name
```
Command parameters

-help
  Specifies that the SQLJ translator describes each of the options that the translator supports. If any other options are specified with -help, they are ignored.

-dir=directory
  Specifies the name of the directory into which SQLJ puts .java files that are generated by the translator and .class files that are generated by the compiler. The default is the directory that contains the SQLJ source files.

  The translator uses the directory structure of the SQLJ source files when it puts the generated files in directories. For example, suppose that you want the translator to process two files:
  
  • file1.sqlj, which is not in a Java package
  • file2.sqlj, which is in Java package sqlj.test

  Also suppose that you specify the parameter -dir=/src when you invoke the translator. The translator puts the Java source file for file1.sqlj in directory /src and puts the Java source file for file2.sqlj in directory /src/sqlj/test.

-d=directory
  Specifies the name of the directory into which SQLJ puts the binary files that are generated by the translator and compiler. These files include the .ser files, the name_SjProfileKeys.class files, and the .class files that are generated by the compiler.

  The default is the directory that contains the SQLJ source files.

  The translator uses the directory structure of the SQLJ source files when it puts the generated files in directories. For example, suppose that you want the translator to process two files:
  
  • file1.sqlj, which is not in a Java package
  • file2.sqlj, which is in Java package sqlj.test

  Also suppose that you specify the parameter -d=/src when you invoke the translator. The translator puts the serialized profiles for file1.sqlj in directory /src and puts the serialized profiles for file2.sqlj in directory /src/sqlj/test.

-compile=true|false
  Specifies whether the SQLJ translator compiles the generated Java source into bytecodes.

  true
    The translator compiles the generated Java source code. This is the default.

  false
    The translator does not compile the generated Java source code.

-linemap=no|yes
  Specifies whether line numbers in Java exceptions match line numbers in the SQLJ source file (the .sqlj file), or line numbers in the Java source file that is generated by the SQLJ translator (the .java file).

  no  Line numbers in Java exceptions match line numbers in the Java source file. This is the default.

  yes  Line numbers in Java exceptions match line numbers in the SQLJ source file.
-smap=no | yes
Specifies whether the SQLJ translator generates a source map (SMAP) file for each SQLJ source file. An SMAP file is used by some Java language debug tools. This file maps lines in the SQLJ source file to lines in the Java source file that is generated by the SQLJ translator. The file is in the Unicode UTF-8 encoding scheme. Its format is described by Original Java Specification Request (JSR) 45, which is available from this web site:
http://www.jcp.org

no  Do not generated SMAP files. This is the default.

yes  Generate SMAP files. An SMAP file name is SQLJ-source-file-name.java.smap. The SQLJ translator places the SMAP file in the same directory as the generated Java source file.

-encoding=encoding-name
Specifies the encoding of the source file. Examples are JIS or EUC. If this option is not specified, the default converter for the operating system is used.

db2optimize
Specifies that the SQLJ translator generates code for a connection context class that is optimized for DB2. -db2optimize optimizes the code for the user-defined context but not the default context.

When you run the SQLJ translator with the -db2optimize option, if your applications use JDBC 3.0 or earlier functions, the IBM Data Server Driver for JDBC and SQLJ file db2jcc.jar must be in the CLASSPATH for compiling the generated Java application. If your applications use JDBC 4.0 or earlier functions, the IBM Data Server Driver for JDBC and SQLJ file db2jcc4.jar must be in the CLASSPATH for compiling the generated Java application.

-ser2class
Specifies that the SQLJ translator converts .ser files to .class files.

-status
Specifies that the SQLJ translator displays status messages as it runs.

-version
Specifies that the SQLJ translator displays the version of the IBM Data Server Driver for JDBC and SQLJ. The information is in this form:
IBM SQLJ xxx.xxxx.xx

-C-help
Specifies that the SQLJ translator displays help information for the Java compiler.

-Ccompiler-option
Specifies a valid Java compiler option that begins with a dash (-). Do not include spaces between -C and the compiler option. If you need to specify multiple compiler options, precede each compiler option with -C. For example:
-C-g -C-verbose

All options are passed to the Java compiler and are not used by the SQLJ translator, except for the following options:

-classpath
Specifies the user class path that is to be used by the SQLJ translator and the Java compiler. This value overrides the CLASSPATH environment variable.
Specifies the source code path that the SQLJ translator and the Java compiler search for class or interface definitions. The SQLJ translator searches for .sqlj and .java files only in directories, not in JAR or zip files.

-JVM-option
Specifies an option that is to be passed to the Java virtual machine (JVM) in which the sqlj command runs. The option must be a valid JVM option that begins with a dash (-). Do not include spaces between -J and the JVM option. If you need to specify multiple JVM options, precede each compiler option with -J. For example:

-J-Xmx128m -J-Xms2M

SQLJ-source-file-name
Specifies a list of SQLJ source files to be translated. This is a required parameter. All SQLJ source file names must have the extension .sqlj.

Output
For each source file, program-name.sqlj, the SQLJ translator produces the following files:
- The generated source program
  The generated source file is named program-name.java.
- A serialized profile file for each connection context class that is used in an SQLJ executable clause
  A serialized profile name is of the following form:
  program-name_SJProfileIDNumber.ser
- If the SQLJ translator invokes the Java compiler, the class files that the compiler generates.

Examples
sqlj -encoding=UTF8 -C-O MyApp.sqlj

db2sqljcustomize - SQLJ profile customizer

db2sqljcustomize processes an SQLJ profile, which contains embedded SQL statements.

By default, db2sqljcustomize produces four DB2 packages: one for each isolation level. db2sqljcustomize augments the profile with DB2-specific information for use at run time.

Authorization
The privilege set of the process must include one of the following authorities:
- DBADM authority
- If the package does not exist, the BINDADD privilege, and one of the following privileges:
  - CREATEIN privilege
  - IMPLICIT_SCHEMA authority on the database if the schema name of the package does not exist
- If the package exists:
  - ALTERIN privilege on the schema
  - BIND privilege on the package
The user also needs all privileges that are required to compile any static SQL statements in the application. Privileges that are granted to groups are not used for authorization checking of static statements.

**Command syntax**

```
db2sqljcustomize [-help]

-url jdbc:db2://server[:port]/database[:property=value;]
-datasource JNDI-name

-user user-ID

-automaticallybind [YES|NO]
-password password

-bindoptions "options-string"
-storebindoptions
-collection collection-name

-qualifier qualifier-name
-rootpkgname package-name-stem
-singlepkgname package-name

-longpkgname
-staticpositioned [NO|YES]

-onlinecheck [YES|NO]

-pkgversion [AUTO|version-id]
```
options-string:

DB2 for z/OS options:
Notes:
1 These options can be specified in any order.

DB2 Database for Linux, UNIX, and Windows options:

Notes:
1 These options can be specified in any order.

Command parameters
- help
  Specifies that the SQLJ customizer describes each of the options that the customizer supports. If any other options are specified with -help, they are ignored.

- url
  Specifies the URL for the data source for which the profile is to be customized. A connection is established to the data source that this URL represents if the -automaticbind or -onlinecheck option is specified as YES or defaults to YES. The variable parts of the -url value are:

  server
    The domain name or IP address of the z/OS system on which the DB2 subsystem resides.
**port**

The TCP/IP server port number that is assigned to the DB2 subsystem. The default is 446.

**-url**

Specifies the URL for the data source for which the profile is to be customized. A connection is established to the data source that this URL represents if the -automaticbind or -onlinecheck option is specified as YES or defaults to YES. The variable parts of the -url value are:

- **server**
  
The domain name or IP address of the operating system on which the database server resides.

- **port**
  
The TCP/IP server port number that is assigned to the database server. The default is 446.

- **database**
  
A name for the database server for which the profile is to be customized.

  If the connection is to a DB2 for z/OS server, database is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:

  ```sql
  SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
  ```

  If the connection is to a DB2 Database for Linux, UNIX, and Windows server, database is the database name that is defined during installation.

  If the connection is to an IBM Cloudscape server, the database is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:

  
  "c:/databases/testdb"

**property=value**

A property for the JDBC connection.

- **datasource JNDI-name**

Specifies the logical name of a DataSource object that was registered with JNDI. The DataSource object represents the data source for which the profile is to be customized. A connection is established to the data source if the -automaticbind or -onlinecheck option is specified as YES or defaults to YES. Specifying -datasource is an alternative to specifying -url. The DataSource object must represent a connection that uses IBM Data Server Driver for JDBC and SQLJ type 4 connectivity.

**-user user-ID**

Specifies the user ID to be used to connect to the data source for online checking or binding a package. You must specify -user if you specify -url. You must specify -user if you specify -datasource, and the DataSource object that JNDI-name represents does not contain a user ID.

**-password password**

Specifies the password to be used to connect to the data source for online checking or binding a package. You must specify -password if you specify -url. You must specify -password if you specify -datasource, and the DataSource object that JNDI-name represents does not contain a password.
**-automaticbind**  YES | NO  
Specifies whether the customizer binds DB2 packages at the data source that is specified by the -url parameter.

The default is YES.

The number of packages and the isolation levels of those packages are controlled by the -rootpkgname and -singlepkgname options.

Before the bind operation can work, the following conditions need to be met:
- TCP/IP and DRDA must be installed at the target data source.
- Valid -url, -username, and -password values must be specified.
- The -username value must have authorization to bind a package at the target data source.

**-pkgversion**  AUTO | version-id  
Specifies the package version that is to be used when packages are bound at the server for the serialized profile that is being customized. db2sqljcustomize stores the version ID in the serialized profile and in the DB2 package. Run-time version verification is based on the consistency token, not the version name. To automatically generate a version name that is based on the consistency token, specify -pkgversion AUTO.

The default is that there is no version.

**-bindoptions**  options-string  
Specifies a list of options, separated by spaces. These options have the same function as DB2 precompile and bind options with the same names. If you are preparing your program to run on a DB2 for z/OS system, specify DB2 for z/OS options. If you are preparing your program to run on a DB2 Database for Linux, UNIX, and Windows system, specify DB2 Database for Linux, UNIX, and Windows options.

*Notes on bind options:*
- Specify ISOLATION only if you also specify the -singlepkgname option.
- The value for STATICREADONLY is YES for servers that support STATICREADONLY, and NO for other servers. When you specify STATICREADONLY YES, DB2 processes ambiguous cursors as if they were read-only cursors. For troubleshooting iterator declaration errors, you need to explicitly specify STATICREADONLY NO, or declare iterators so that they are unambiguous. For example, if you want an iterator to be unambiguously updatable, declare the iterator to implement sqlj.runtime.ForUpdate. If you want an iterator to be read-only, include the FOR READ ONLY clause in SELECT statements that use the iterator.

*Important:* Specify only those program preparation options that are appropriate for the data source at which you are binding a package. Some values and defaults for the IBM Data Server Driver for JDBC and SQLJ are different from the values and defaults for DB2.

**-storebindoptions**  
Specifies that values for the -bindoptions and -staticpositioned parameters are stored in the serialized profile. If db2sqljbinder is invoked without the -bindoptions or -staticpositioned parameter, the values that are stored in the serialized profile are used during the bind operation. When multiple serialized profiles are specified for one invocation of db2sqljcustomize, the parameter values are stored in each serialized profile. The stored values are displayed in the output from the db2sqljprint utility.
-collection  collection-name
The qualifier for the packages that db2sqljcustomize binds. db2sqljcustomize
stores this value in the customized serialized profile, and it is used when the
associated packages are bound. If you do not specify this parameter,
db2sqljcustomize uses a collection ID of NULLID.

-onlinecheck  YES|NO
Specifies whether online checking of data types in the SQLJ program is to be
performed. The -url or -datasource option determines the data source that is to
be used for online checking. The default is YES if the -url or -datasource
parameter is specified. Otherwise, the default is NO.

-qualifier  qualifier-name
Specifies the qualifier that is to be used for unqualified objects in the SQLJ
program during online checking. This value is not used as the qualifier when
the packages are bound.

-rootpkgname|-singlepkgname
Specifies the names for the packages that are associated with the program. If
-automaticbind is NO, these package names are used when db2sqljbinder runs.
The meanings of the parameters are:

-rootpkgname  package-name-stem
Specifies that the customizer creates four packages, one for each of the four
DB2 isolation levels. The names for the four packages are:

  package-name-stem1
  For isolation level UR

  package-name-stem2
  For isolation level CS

  package-name-stem3
  For isolation level RS

  package-name-stem4
  For isolation level RR

If -longpkgname is not specified, package-name-stem must be an
alphanumeric string of seven or fewer bytes.

If -longpkgname is specified, package-name-stem must be an alphanumeric
string of 127 or fewer bytes.

-singlepkgname  package-name
Specifies that the customizer creates one package, with the name
package-name. If you specify this option, your program can run at only one
isolation level. You specify the isolation level for the package by specifying
the ISOLATION option in the -bindoptions options string.

If -longpkgname is not specified, package-name must be an alphanumeric
string of eight or fewer bytes.

If -longpkgname is specified, package-name must be an alphanumeric
string of 128 or fewer bytes.

Using the -singlepkgname option is not recommended.

Recommendation: If the target data source is DB2 for z/OS, use uppercase
characters for the package-name-stem or package-name value. DB2 for z/OS
systems that are defined with certain CCSID values cannot tolerate lowercase
characters in package names or collection names.
If you do not specify -rootpkgname or -singlepkgname, db2sqljcustomize generates four package names that are based on the serialized profile name. A serialized profile name is of the following form:

```
program-name_SJProfileIDNumber.ser
```

The four generated package names are of the following form:

```
Bytes-from-program-nameIDNumberPkgIsolation
```

Table 115 shows the parts of a generated package name and the number of bytes for each part.

The maximum length of a package name is maxlen. maxlen is 8 if -longpkgname is not specified. maxlen is 128 if -longpkgname is specified.

### Table 115. Parts of a package name that is generated by db2sqljcustomize

<table>
<thead>
<tr>
<th>Package name part</th>
<th>Number of bytes</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes-from-program-name</td>
<td>( m = \min(\text{Length}(\text{program-name}), \text{maxlen} - 1 - \text{Length}(\text{IDNumber})) )</td>
<td>First ( m ) bytes of \text{program-name}, in uppercase</td>
</tr>
<tr>
<td>IDNumber</td>
<td>\text{Length}(\text{IDNumber})</td>
<td>IDNumber</td>
</tr>
<tr>
<td>PkgIsolation</td>
<td>1</td>
<td>1, 2, 3, or 4. This value represents the transaction isolation level for the package. See Table 116</td>
</tr>
</tbody>
</table>

Table 116 shows the values of the PkgIsolation portion of a package name that is generated by db2sqljcustomize.

### Table 116. PkgIsolation values and associated isolation levels

<table>
<thead>
<tr>
<th>PkgNumber value</th>
<th>Isolation level for package</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Uncommitted read (UR)</td>
</tr>
<tr>
<td>2</td>
<td>Cursor stability (CS)</td>
</tr>
<tr>
<td>3</td>
<td>Read stability (RS)</td>
</tr>
<tr>
<td>4</td>
<td>Repeatable read (RR)</td>
</tr>
</tbody>
</table>

**Example:** Suppose that a profile name is ThisIsMyProg_SJProfile111.ser. The db2sqljcustomize option -longpkgname is not specified. Therefore, \text{Bytes-from-program-name} is the first four bytes of ThisIsMyProg, translated to uppercase, or THIS. IDNumber is 111. The four package names are:

- THIS1111
- THIS1112
- THIS1113
- THIS1114

**Example:** Suppose that a profile name is ThisIsMyProg_SJProfile111.ser. The db2sqljcustomize option -longpkgname is specified. Therefore, \text{Bytes-from-program-name} is ThisIsMyProg, translated to uppercase, or THISISMYPROG. IDNumber is 111. The four package names are:

- THISISMYPROG1111
- THISISMYPROG1112
- THISISMYPROG1113
- THISISMYPROG1114

**Example:** Suppose that a profile name is A_SJProfile0.ser. \text{Bytes-from-program-name} is A. IDNumber is 0. Therefore, the four package names are:
Letting `db2sqljcustomize` generate package names is not recommended. If any generated package names are the same as the names of existing packages, `db2sqljcustomize` overwrites the existing packages. To ensure uniqueness of package names, specify `-rootpkgname`.

**-longpkgname**

Specifies that the names of the DB2 packages that `db2sqljcustomize` generates can be up to 128 bytes. Use this option only if you are binding packages at a server that supports long package names. If you specify `-singlepkgname` or `-rootpkgname`, you must also specify `-longpkgname` under the following conditions:

- The argument of `-singlepkgname` is longer than eight bytes.
- The argument of `-rootpkgname` is longer than seven bytes.

**-staticpositioned** NO | YES

For iterators that are declared in the same source file as positioned UPDATE statements that use the iterators, specifies whether the positioned UPDATEs are executed as statically bound statements. The default is NO. NO means that the positioned UPDATEs are executed as dynamically prepared statements.

**-zosDescProcParms**

Specifies that `db2sqljcustomize` queries the DB2 catalog at the target data source to determine the SQL parameter data types that correspond to the host variables in CALL statements.

- `zosDescProcParms` applies only to programs that run on DB2 for z/OS data servers.

If `-zosDescProcParms` is specified, and the authorization ID under which `db2sqljcustomize` runs does not have read access to the `SYSIBM.SYSROUTINES` catalog table, `db2sqljcustomize` returns an error and uses the host variable data types in the CALL statements to determine the SQL data types.

Specification of `-zosDescProcParms` can lead to more efficient storage usage at run time. If SQL data type information is available, SQLJ has information about the length and precision of INOUT and OUT parameters, so it allocates only the amount of memory that is needed for those parameters. Availability of SQL data type information can have the biggest impact on storage usage for character INOUT parameters, LOB OUT parameters, and decimal OUT parameters.

When `-zosDescProcParms` is specified, the DB2 data server uses the specified or default value of `-zosProcedurePath` to resolve unqualified names of stored procedures for which SQL data type information is requested.

If `-zosDescProcParms` is not specified, `db2sqljcustomize` uses the host variable data types in the CALL statements to determine the SQL data types. If `db2sqljcustomize` determines the wrong SQL data type, an SQL error might occur at run time. For example, if the Java host variable type is String, and the corresponding stored procedure parameter type is VARCHAR FOR BIT DATA, an SQL run-time error such as -4220 might occur.

**-zosProcedurePath** procedure-path

Specifies a list of schema names that DB2 for z/OS uses to resolve unqualified stored procedure names during online checking of an SQLJ program.
-zosProcedurePath applies to programs that are to be run on DB2 for z/OS database servers only.

The list is a String value that is a comma-separated list of schema names that is enclosed in double quotation marks. The DB2 database server inserts that list into the SQL path for resolution of unqualified stored procedure names. The SQL path is:

SYSIBM, SYSFUN, SYSPROC, procedure-path, qualifier-name, user-ID

qualifier-name is the value of the -qualifier parameter, and user-ID is the value of the -user parameter.

The DB2 database server tries the schema names in the SQL path from left to right until it finds a match with the name of a stored procedure that exists on that database server. If the DB2 database server finds a match, it obtains the information about the parameters for that stored procedure from the DB2 catalog. If the DB2 database server does not find a match, SQLJ sets the parameter data without any DB2 catalog information.

If -zosProcedurePath is not specified, the DB2 database server uses this SQL path:

SYSIBM, SYSFUN, SYSPROC, qualifier-name, user-ID

If the -qualifier parameter is not specified, the SQL path does not include qualifier-name.

-genDBRM
Specifies that db2sqljcustomise generates database request modules (DBRMs). Those DBRMs can be used to create DB2 for z/OS plans and packages.

-genDBRM applies to programs that are to be run on DB2 for z/OS database servers only.

If -genDBRM and -automaticbind NO are specified, db2sqljcustomise creates the DBRMs but does not bind them into DB2 packages. If -genDBRM and -automaticbind YES are specified, db2sqljcustomise creates the DBRMs and binds them into DB2 packages.

One DBRM is created for each DB2 isolation level. The naming convention for the generated DBRM files is the same as the naming convention for packages. For example, if -rootpkgname SQLJSA0 is specified, and -genDBRM is also specified, the names of the four DBRM files are:

- SQLJSA01
- SQLJSA02
- SQLJSA03
- SQLJSA04

-DBRMDir directory-name
When -genDBRM is specified, -DBRMDir specifies the local directory into which db2sqljcustomise puts the generated DBRM files. The default is the current directory.

-DBRMdir applies to programs that are to be run on DB2 for z/OS database servers only.

-tracefile file-name
Enables tracing and identifies the output file for trace information. This option should be specified only under the direction of IBM Software Support.
-tracelevel
  If -tracefile is specified, indicates what to trace while db2sqljcustomize runs.
  The default is TRACE_SQLJ. This option should be specified only under the
direction of IBM Software Support.

serialized-profile-name | file-name.grp
  Specifies the names of one or more serialized profiles that are to be
customized. The specified serialized profile must be in a directory that is
named in the CLASSPATH environment variable.

A serialized profile name is of the following form:
  program-name_SJProfileIDNumber.ser

You can specify the serialized profile name with or without the .ser extension.

program-name is the name of the SQLJ source program, without the extension
.sqlj. n is an integer between 0 and m-1, where m is the number of serialized
profiles that the SQLJ translator generated from the SQLJ source program.

You can specify serialized profile names in one of the following ways:

• List the names in the db2sqljcustomize command. Multiple serialized profile
  names must be separated by spaces.

• Specify the serialized profile names, one on each line, in a file with the name
  file-name.grp, and specify file-name.grp in the db2sqljcustomize command.

If you specify more than one serialized profile name, and if you specify or use
the default value of -automaticbind YES, db2sqljcustomize binds a single DB2
package from the profiles. When you use db2sqljcustomize to create a single
DB2 package from multiple serialized profiles, you must also specify the
-rootpkgname or -singlepkgname option.

If you specify more than one serialized profile name, and you specify
-automaticbind NO, if you want to bind the serialized profiles into a single
DB2 package when you run db2sqljbind, you need to specify the same list of
serialized profile names, in the same order, in db2sqljcustomize and
db2sqljbind.

If you specify one or more file-name.grp files, and you specify -automaticbind
NO, when you run db2sqljbind, you must specify that same list of files, and in
the same order in which the files were customized.

You cannot run db2sqljcustomize on individual files, and then group those files
when you run db2sqljbind.

Output

When db2sqljcustomize runs, it creates a customized serialized profile. It also
creates DB2 packages, if the automaticbind value is YES.

Examples
  db2sqljcustomize -user richler -password mordecai
  -url jdbc:db2:/server:50000/sample -collection duddy
  -bindoptions "EXPLAIN YES" pgmname_SJProfile0.ser

Usage notes

Online checking is always recommended: It is highly recommended that you use
online checking when you customize your serialized profiles. Online checking
determines information about the data types and lengths of DB2 host variables, and is especially important for the following items:

- **Predicates with java.lang.String host variables and CHAR columns**
  
  Unlike character variables in other host languages, Java String host variables are not declared with a length attribute. To optimize a query properly that contains character host variables, DB2 needs the length of the host variables. For example, suppose that a query has a predicate in which a String host variable is compared to a CHAR column, and an index is defined on the CHAR column. If DB2 cannot determine the length of the host variable, it might do a table space scan instead of an index scan. Online checking avoids this problem by providing the lengths of the corresponding character columns.

- **Predicates with java.lang.String host variables and GRAPHIC columns**
  
  Without online checking, DB2 might issue a bind error (SQLCODE -134) when it encounters a predicate in which a String host variable is compared to a GRAPHIC column.

- **Column names in the result table of an SQLJ SELECT statement at a remote server:**
  
  Without online checking, the driver cannot determine the column names for the result table of a remote SELECT.

**Customizing multiple serialized profiles together:** Multiple serialized profiles can be customized together to create a single DB2 package. If you do this, and if you specify -staticpositioned YES, any positioned UPDATE or DELETE statement that references a cursor that is declared earlier in the package executes statically, even if the UPDATE or DELETE statement is in a different source file from the cursor declaration. If you want -staticpositioned YES behavior when your program consists of multiple source files, you need to order the profiles in the db2sqljcustomize command to cause cursor declarations to be ahead of positioned UPDATE or DELETE statements in the package. To do that, list profiles that contain SELECT statements that assign result tables to iterators before profiles that contain the positioned UPDATE or DELETE statements that reference those iterators.

**Using a customized serialized profile at one data source that was customized at another data source:** You can run db2sqljcustomize to produce a customized serialized profile for an SQLJ program at one data source, and then use that profile at another data source. You do this by running db2sqljbind multiple times on customized serialized profiles that you created by running db2sqljcustomize once. When you run the programs at these data sources, the DB2 objects that the programs access must be identical at every data source. For example, tables at all data sources must have the same encoding schemes and the same columns with the same data types.

**Using the -collection parameter:** db2sqljcustomize stores the DB2 collection name in each customized serialized profile that it produces. When an SQLJ program is executed, the driver uses the collection name that is stored in the customized serialized profile to search for packages to execute. The name that is stored in the customized serialized profile is determined by the value of the -collection parameter. Only one collection ID can be stored in the serialized profile. However, you can bind the same serialized profile into multiple package collections by specifying the COLLECTION option in the -bindoptions parameter. To execute a package that is in a collection other than the collection that is specified in the serialized profile, include a SET CURRENT PACKAGESET statement in the program.
Using the VERSION parameter: Use the VERSION parameter to bind two or more versions of a package for the same SQLJ program into the same collection. You might do this if you have changed an SQLJ source program, and you want to run the old and new versions of the program.

To maintain two versions of a package, follow these steps:

1. Change the code in your source program.
2. Translate the source program to create a new serialized profile. Ensure that you do not overwrite your original serialized profile.
3. Run db2sqljcustomize to customize the serialized profile and create DB2 packages with the same package names and in the same collection as the original packages. Do this by using the same values for -rootpkgname and -collection when you bind the new packages that you used when you created the original packages. Specify the VERSION option in the -bindoptions parameter to put a version ID in the new customized serialized profile and in the new packages.

   It is essential that you specify the VERSION option when you perform this step. If you do not, you overwrite your original packages.

When you run the old version of the program, DB2 loads the old versions of the packages. When you run the new version of the program, DB2 loads the new versions of the packages.

Binding packages and plans on DB2 for z/OS: You can use the db2sqljcustomize -genDBRM parameter to create DBRMs on your local system. You can then transfer those DBRMs to a DB2 for z/OS system, and bind them into packages or plans there. If you plan to use this technique, you need to transfer the DBRM files to the z/OS system as binary files, to a partitioned data set with record format FB and record length 80. When you bind the packages or plans, you need to specify the following bind option values:

ENCODING(EBCDIC)
   The IBM Data Server Driver for JDBC and SQLJ on DB2 for z/OS requires EBCDIC encoding for your packages and plans.

DYNAMICRULES(BIND)
   This option ensures consistent authorization rules when SQLJ uses dynamic SQL. SQLJ uses dynamic SQL for positioned UPDATE or DELETE operations that involve multiple SQLJ programs.

DBPROTOCOL(DRDA)
   Private protocol is deprecated in DB2 for z/OS Version 9.1, and no longer supported in DB2 for z/OS Version 10. You should use DBPROTOCOL(DRDA) for all applications.

db2sqljbind - SQLJ profile binder

db2sqljbind binds DB2 packages for a serialized profile that was previously customized with the db2sqljcustomize command.

Applications that run with the IBM Data Server Driver for JDBC and SQLJ require packages but no plans. If the db2sqljcustomize -automaticbind option is specified as YES or defaults to YES, db2sqljcustomize binds packages for you at the data source that you specify in the -url parameter. However, if -automaticbind is NO, if a bind fails when db2sqljcustomize runs, or if you want to create identical packages at multiple locations for the same serialized profile, you can use the db2sqljbind command to bind packages.
Authorization

The privilege set of the process must include one of the following authorities:

- DBADM authority
- If the package does not exist, the BINDADD privilege, and one of the following privileges:
  - CREATEIN privilege
  - IMPLICIT_SCHEMA authority on the database if the schema name of the package does not exist
- If the package exists:
  - ALTERIN privilege on the schema
  - BIND privilege on the package

The user also needs all privileges that are required to compile any static SQL statements in the application. Privileges that are granted to groups are not used for authorization checking of static statements.

Command syntax

db2sqljbind
  -url jdbc:db2://server/database
  -user user-ID -password password
  -bindoptions "options-string"
  -staticpositioned NO
  -staticpositioned YES
  -genDBRM
  -DBRMDir directory-name
  -tracelevel TRACE_SQLJ
  -tracefile file-name
  -tracelevel TRACE_NONE
  -tracelevel TRACE_CONNECTION_CALLS
  -tracelevel TRACE_STATEMENT_CALLS
  -tracelevel TRACE_RESULT_SET_CALLS
  -tracelevel TRACE_DRIVER_CONFIGURATION
  -tracelevel TRACE_CONNECTS
  -tracelevel TRACE_DRDA_FLOWS
  -tracelevel TRACE_RESULT_SET_META_DATA
  -tracelevel TRACE_PARAMETER_META_DATA
  -tracelevel TRACE_DIAGNOSTICS
  -tracelevel TRACE_SQLJ
  -tracelevel TRACE_XA_CALLS
  -tracelevel TRACE_TRACEPOINTS
  -tracelevel TRACE_ALL
options-string:

DB2 for z/OS options:

(1) ACTION(REPLACE)
REPLVER(version-id)

ACTION(ADD)

DBPROTOCOL(DRDA)

DEGREE(1)

DEGREE(ANY)

EXPLAIN(NO)

IMMEDWRITE(NO)

ISOLATION(RR)

NOREOPT(VARS)

OWNER(authorization-ID)

RELEASE(COMMIT)

SQLERROR(NOPACKAGE)

VALIDATE(RUN)

Notes:
1. These options can be specified in any order.

DB2 Database for Linux, UNIX, and Windows options

(1) BLOCKING UNAMBIG

DEGREE 1

EXPLAIN NO

EXPLSNAP NO

FEDERATED NO

FUNCPATH schema-name

INSERT DEF

ISOLATION CS

OWNER authorization-ID

IMMEDWRITE(YES)

ISOLATION(RS)

REOPT(VARS)

OPTHINT(hint-ID)

IMMEDWRITE(PH1)

ISOLATION(CS)

REOPT(VARS)

OPTHINT(hint-ID)

IMMEDWRITE(YES)

ISOLATION(UR)

REOPT(VARS)

OPTHINT(hint-ID)

IMMEDWRITE(NO)

ISOLATION(RS)

REOPT(VARS)

OPTHINT(hint-ID)

IMMEDWRITE(NO)

ISOLATION(RR)

REOPT(VARS)

OPTHINT(hint-ID)

IMMEDWRITE(NO)

ISOLATION(RS)

REOPT(VARS)

OPTHINT(hint-ID)

IMMEDWRITE(NO)

ISOLATION(UR)

REOPT(VARS)

OPTHINT(hint-ID)
Notes:
1 These options can be specified in any order.

**Command parameters**

- **-help**
  Specifies that `db2sqljbind` describes each of the options that it supports. If any other options are specified with `-help`, they are ignored.

- **-url**
  Specifies the URL for the data source for which the profile is to be customized. A connection is established to the data source that this URL represents if the `-automaticbind` or `-onlinecheck` option is specified as YES or defaults to YES. The variable parts of the `-url` value are:
  - **server**
    The domain name or IP address of the operating system on which the database server resides.
  - **port**
    The TCP/IP server port number that is assigned to the database server. The default is 446.
  - **database**
    A name for the database server for which the profile is to be customized.
    - If the connection is to a DB2 for z/OS server, `database` is the DB2 location name that is defined during installation. All characters in this value must be uppercase characters. You can determine the location name by executing the following SQL statement on the server:
      ```sql
      SELECT CURRENT SERVER FROM SYSIBM.SYSDUMMY1;
      ```
    - If the connection is to a DB2 Database for Linux, UNIX, and Windows server, `database` is the database name that is defined during installation.
    - If the connection is to an IBM Cloudscape server, the `database` is the fully-qualified name of the file that contains the database. This name must be enclosed in double quotation marks ("). For example:
      ```
      "c:/databases/testdb"
      ```
  - **property=value**
    A property for the JDBC connection.

- **-user user-ID**
  Specifies the user ID to be used to connect to the data source for binding the package.

- **-password password**
  Specifies the password to be used to connect to the data source for binding the package.
-bindoptions  options-string
   Specifies a list of options, separated by spaces. These options have the same
   function as DB2 precompile and bind options with the same names. If you are
   preparing your program to run on a DB2 for z/OS system, specify DB2 for
   z/OS options. If you are preparing your program to run on a DB2 Database for
   Linux, UNIX, and Windows system, specify DB2 Database for Linux, UNIX,
   and Windows options.

   Notes on bind options:
   • Specify VERSION only if the following conditions are true:
     – If you are binding a package at a DB2 Database for Linux, UNIX, and
       Windows system, the system is at Version 8 or later.
     – You rerun the translator on a program before you bind the associated
       package with a new VERSION value.
   • The value for STATICREADONLY is YES for servers that support
     STATICREADONLY, and NO for other servers. When you specify
     STATICREADONLY YES, DB2 processes ambiguous cursors as if they were
     read-only cursors. For troubleshooting iterator declaration errors, you need
     to explicitly specify STATICREADONLY NO, or declare iterators so that they
     are unambiguous. For example, if you want an iterator to be unambiguously
     updatable, declare the iterator to implement sqlj.runtime.ForUpdate. If you
     want an iterator to be read-only, include the FOR READ ONLY clause in
     SELECT statements that use the iterator.

   Important: Specify only those program preparation options that are
   appropriate for the data source at which you are binding a package. Some
   values and defaults for the IBM Data Server Driver for JDBC and SQLJ are
   different from the values and defaults for DB2.

-staticpositioned NO|YES
   For iterators that are declared in the same source file as positioned UPDATE
   statements that use the iterators, specifies whether the positioned UPDATEs
   are executed as statically bound statements. The default is NO. NO means that
   the positioned UPDATEs are executed as dynamically prepared statements.
   This value must be the same as the -staticpositioned value for the previous
   db2sqljcustomize invocation for the serialized profile.

-genDBRM
   Specifies that db2sqljbind generates database request modules (DBRMs) from
   the serialized profile, and that db2sqljbind does not perform remote bind
   operations.

   -genDBRM applies to programs that are to be run on DB2 for z/OS database
   servers only.

-DBRMDir directory-name
   When -genDBRM is specified, -DBRMDir specifies the local directory into
   which db2sqljbind puts the generated DBRM files. The default is the current
directory.

   -DBRMDir applies to programs that are to be run on DB2 for z/OS database
   servers only.

-tracefile  file-name
   Enables tracing and identifies the output file for trace information. This option
   should be specified only under the direction of IBM Software Support.
-tracelevel
If -tracefile is specified, indicates what to trace while db2sqljcustomize runs.
The default is TRACE_SQLJ. This option should be specified only under the
direction of IBM Software Support.

serialized-profile-name[|file-name.grp]
Specifies the names of one or more serialized profiles from which the package
is bound. A serialized profile name is of the following form:

    program-name_SJProfileIDNumber.ser

`program-name` is the name of the SQLJ source program, without the extension
.sqlj. `n` is an integer between 0 and `m-1`, where `m` is the number of serialized
profiles that the SQLJ translator generated from the SQLJ source program.

You can specify serialized profile names in one of the following ways:
- List the names in the db2sqljcustomize command. Multiple serialized profile
  names must be separated by spaces.
- Specify the serialized profile names, one on each line, in a file with the name
  `file-name.grp`, and specify `file-name.grp` in the db2sqljbind command.

If you specify more than one serialized profile name to bind a single DB2
package from several serialized profiles, you must have specified the same
serialized profile names, in the same order, when you ran db2sqljcustomize.

If you specify one or more `file-name.grp` files, you must have run
db2sqljcustomize once with that same list of files. The order in which you
specify the files in db2sqljbind must be the same as the order in
db2sqljcustomize.

You cannot run db2sqljcustomize on individual files, and then group those files
when you run db2sqljbind.

Examples

db2sqljbind -user richler -password mordecai
  -url jdbc:db2://server:50000/sample -bindoptions "EXPLAIN YES"
  pgmname_SJProfile0.ser

Usage notes

Package names produced by db2sqljbind: The names of the packages that are
created by db2sqljbind are the names that you specified using the-rootpkgname or
-singlepkgname parameter when you ran db2sqljcustomize. If you did not specify
-rootpkgname or -singlepkgname, the package names are the first seven bytes of
the profile name, appended with the isolation level character.

DYNAMICRULES value for db2sqljbind: The DYNAMICRULES bind option
determines a number of run-time attributes for the DB2 package. Two of those
attributes are the authorization ID that is used to check authorization, and the
qualifier that is used for unqualified objects. To ensure the correct authorization for
dynamically executed positioned UPDATE and DELETE statements in SQLJ
programs, db2sqljbind always binds the DB2 packages with the
DYNAMICRULES(BIND) option. You cannot modify this option. The
DYNAMICRULES(BIND) option causes the SET CURRENT SQLID statement and
the SET CURRENT SCHEMA statement to have no impact on an SQLJ program,
because those statements affect only dynamic statements that are bound with
DYNAMICRULES values other than BIND.
With DYNAMICRULES(BIND), unqualified table, view, index, and alias names in dynamic SQL statements are implicitly qualified with value of the bind option QUALIFIER. If you do not specify QUALIFIER, DB2 uses the authorization ID of the package owner as the implicit qualifier. If this behavior is not suitable for your program, you can use one of the following techniques to set the correct qualifier:

- Force positioned UPDATE and DELETE statements to execute statically. You can use the -staticpositioned YES option of db2sqljcustomize or db2sqljbind to do this if the cursor (iterator) for a positioned UPDATE or DELETE statement is in the same package as the positioned UPDATE or DELETE statement.
- Fully qualify DB2 table names in positioned UPDATE and positioned DELETE statements.

**db2sqljprint - SQLJ profile printer**

`db2sqljprint` prints the contents of the customized version of a profile as plain text.

**Authorization**

None

**Command syntax**

```
$db2sqljprint <profilename>
```

**Command parameters**

`profilename`

Specifies the relative or absolute name of an SQLJ profile file. When an SQLJ file is translated into a Java source file, information about the SQL operations it contains is stored in SQLJ-generated resource files called profiles. Profiles are identified by the suffix _SJProfileN (where N is an integer) following the name of the original input file. They have a .ser extension. Profile names can be specified with or without the .ser extension.

**Examples**

```
db2sqljprint pgmname_SJProfile0.ser
```
Appendix A. Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- **DB2 Information Center**
  - Topics (Task, concept and reference topics)
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- **DB2 books**
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - Printed books
- **Command line help**
  - Command help
  - Message help

**Note:** The DB2 Information Center topics are updated more frequently than either the PDF or the hard-copy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com®.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at [http://www.ibm.com/software/data/sw-library/](http://www.ibm.com/software/data/sw-library/).

**Documentation feedback**

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

If you would like to help IBM make the IBM Information Management products easier to use, take the Consumability Survey: [http://www.ibm.com/software/data/info/consumability-survey/](http://www.ibm.com/software/data/info/consumability-survey/).

Although the tables identify books available in print, the books might not be available in your country or region.

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed in the following table.

**Note:** The DB2 Information Center is updated more frequently than either the PDF or the hard-copy books.

### Table 117. DB2 technical information

<table>
<thead>
<tr>
<th>Name</th>
<th>Form Number</th>
<th>Available in print</th>
<th>Last updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administrative API Reference</td>
<td>SC23-5842-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Administrative Routines and Views</td>
<td>SC23-5843-03</td>
<td>No</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Call Level Interface Guide and Reference, Volume 1</td>
<td>SC23-5844-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Call Level Interface Guide and Reference, Volume 2</td>
<td>SC23-5845-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Command Reference</td>
<td>SC23-5846-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Data Movement Utilities Guide and Reference</td>
<td>SC23-5847-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Data Recovery and High Availability Guide and Reference</td>
<td>SC23-5848-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Data Servers, Databases, and Database Objects Guide</td>
<td>SC23-5849-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Database Security Guide</td>
<td>SC23-5850-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Developing ADO.NET and OLE DB Applications</td>
<td>SC23-5851-02</td>
<td>Yes</td>
<td>April, 2009</td>
</tr>
<tr>
<td>Developing Embedded SQL Applications</td>
<td>SC23-5852-02</td>
<td>Yes</td>
<td>April, 2009</td>
</tr>
<tr>
<td>Developing Java Applications</td>
<td>SC23-5853-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Developing Perl and PHP Applications</td>
<td>SC23-5854-02</td>
<td>No</td>
<td>April, 2009</td>
</tr>
<tr>
<td>Developing User-defined Routines (SQL and External)</td>
<td>SC23-5855-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Name</td>
<td>Form Number</td>
<td>Available in print</td>
<td>Last updated</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>-------------</td>
<td>--------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Getting Started with Database Application Development</td>
<td>GC23-5856-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Getting Started with DB2 installation and administration on Linux and Windows</td>
<td>GC23-5857-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Internationalization Guide</td>
<td>SC23-5858-02</td>
<td>Yes</td>
<td>April, 2009</td>
</tr>
<tr>
<td>Message Reference, Volume 1</td>
<td>GI11-7855-00</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Message Reference, Volume 2</td>
<td>GI11-7856-00</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Migration Guide</td>
<td>GC23-5859-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Net Search Extender Administration and User’s Guide</td>
<td>SC23-8509-02</td>
<td>Yes</td>
<td>April, 2009</td>
</tr>
<tr>
<td>Partitioning and Clustering Guide</td>
<td>SC23-5860-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Query Patroller Administration and User’s Guide</td>
<td>SC23-8507-01</td>
<td>Yes</td>
<td>April, 2009</td>
</tr>
<tr>
<td>Quick Beginnings for IBM Data Server Clients</td>
<td>GC23-5863-03</td>
<td>No</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Quick Beginnings for DB2 Servers</td>
<td>GC23-5864-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Spatial Extender and Geodetic Data Management Feature User’s Guide and Reference</td>
<td>SC23-8508-02</td>
<td>Yes</td>
<td>April, 2009</td>
</tr>
<tr>
<td>SQL Reference, Volume 1</td>
<td>SC23-5861-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>SQL Reference, Volume 2</td>
<td>SC23-5862-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>System Monitor Guide and Reference</td>
<td>SC23-5865-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Text Search Guide</td>
<td>SC23-5866-02</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Troubleshooting Guide</td>
<td>GI11-7857-03</td>
<td>No</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Tuning Database Performance</td>
<td>SC23-5867-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Visual Explain Tutorial</td>
<td>SC23-5868-00</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>What's New</td>
<td>SC23-5869-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Workload Manager Guide and Reference</td>
<td>SC23-5870-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>pureXML Guide</td>
<td>SC23-5871-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>XQuery Reference</td>
<td>SC23-5872-02</td>
<td>No</td>
<td>April, 2009</td>
</tr>
</tbody>
</table>
Table 118. DB2 Connect-specific technical information

<table>
<thead>
<tr>
<th>Name</th>
<th>Form Number</th>
<th>Available in print</th>
<th>Last updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick Beginnings for DB2 Connect Personal Edition</td>
<td>GC23-5839-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>Quick Beginnings for DB2 Connect Servers</td>
<td>GC23-5840-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
<tr>
<td>DB2 Connect User’s Guide</td>
<td>SC23-5841-03</td>
<td>Yes</td>
<td>December, 2010</td>
</tr>
</tbody>
</table>

Table 119. Information Integration technical information

<table>
<thead>
<tr>
<th>Name</th>
<th>Form Number</th>
<th>Available in print</th>
<th>Last updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Integration: Administration Guide for Federated Systems</td>
<td>SC19-1020-01</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Information Integration: ASNCLP Program Reference for Replication and Event Publishing</td>
<td>SC19-1018-02</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Information Integration: Configuration Guide for Federated Data Sources</td>
<td>SC19-1034-01</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Information Integration: SQL Replication Guide and Reference</td>
<td>SC19-1030-01</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Information Integration: Introduction to Replication and Event Publishing</td>
<td>GC19-1028-01</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the DB2 PDF Documentation DVD are unavailable in print. For example, neither volume of the DB2 Message Reference is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation DVD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation DVD are available in print.

Note: The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at [http://publib.boulder.ibm.com/infocenter/db2luw/v9r5](http://publib.boulder.ibm.com/infocenter/db2luw/v9r5)

To order printed DB2 books:
- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at [http://www.ibm.com/shop/](http://www.ibm.com/shop/)
publications/order. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.

- To order printed DB2 books from your local IBM representative:
  1. Locate the contact information for your local representative from one of the following Web sites:
     - The IBM directory of world wide contacts at www.ibm.com/planetwide
       publications/order. You will need to select your country, region, or language to access appropriate publications home page for your location. From this page, follow the “About this site” link.
  2. When you call, specify that you want to order a DB2 publication.
  3. Provide your representative with the titles and form numbers of the books that you want to order. For titles and form numbers, see “DB2 technical library in hardcopy or PDF format” on page 514.

Displaying SQL state help from the command line processor

DB2 returns an SQLSTATE value for conditions that could be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

To invoke SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where sqlstate represents a valid five-digit SQL state and class code represents the first two digits of the SQL state.

For example, ? 08003 displays help for the 08003 SQL state, and ? 08 displays help for the 08 class code.

Accessing different versions of the DB2 Information Center

For DB2 Version 9.8 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/

For DB2 Version 9.7 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/

For DB2 Version 9.5 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r3/

For DB2 Version 9.1 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/

For DB2 Version 8 topics, go to the DB2 Information Center URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/
Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

- To display topics in your preferred language in the Internet Explorer browser:
  1. In Internet Explorer, click the Tools —> Internet Options —> Languages... button. The Language Preferences window opens.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     - To add a new language to the list, click the Add... button.
     
     **Note:** Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
     
     - To move a language to the top of the list, select the language and click the Move Up button until the language is first in the list of languages.
  3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

- To display topics in your preferred language in a Firefox or Mozilla browser:
  1. Select the button in the Languages section of the Tools —> Options —> Advanced dialog. The Languages panel is displayed in the Preferences window.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     - To add a new language to the list, click the Add... button to select a language from the Add Languages window.
     
     - To move a language to the top of the list, select the language and click the Move Up button until the language is first in the list of languages.
  3. Clear the browser cache and then refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you might have to also change the regional settings of your operating system to the locale and language of your choice.

Updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

Updating your locally-installed DB2 Information Center requires that you:

1. Stop the DB2 Information Center on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. Non-Administrative and Non-Root DB2 Information Centers always run in stand-alone mode.

2. Use the update feature to see what updates are available. If there are updates that you would like to install, you can use the update feature to obtain and install them.
Note: If your environment requires installing the DB2 Information Center updates on a machine that is not connected to the internet, you have to mirror the update site to a local file system using a machine that is connected to the internet and has the DB2 Information Center installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site. If update packages are available, use the update feature to get the packages. However, the update feature is only available in stand-alone mode.

3. Stop the stand-alone Information Center, and restart the DB2 Information Center on your computer.

Note: On Windows Vista, the commands listed below must be run as an administrator. To launch a command prompt or graphical tool with full administrator privileges, right-click on the shortcut and then select Run as administrator.

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center.
   - On Windows, click Start → Control Panel → Administrative Tools → Services. Then right-click on DB2 Information Center service and select Stop.
   - On Linux, enter the following command:
     ```bash
     /etc/init.d/db2icdv95 stop
     ```

2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `Program_files\IBM\DB2 Information Center\Version 9.5` directory, where `Program_files` represents the location of the Program Files directory.
     c. Navigate from the installation directory to the `doc\bin` directory.
     d. Run the `help_start.bat` file:
        ```bash
        help_start.bat
        ```
   - On Linux:
     a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `/opt/ibm/db2ic/V9.5` directory.
     b. Navigate from the installation directory to the `doc/bin` directory.
     c. Run the `help_start` script:
        ```bash
        help_start
        ```

The system’s default Web browser launches to display the stand-alone Information Center.

3. Click the Update button (更新). On the right-hand panel of the Information Center, click Find Updates. A list of updates for existing documentation displays.

4. To initiate the installation process, check the selections you want to install, then click Install Updates.

5. After the installation process has completed, click Finish.

6. Stop the stand-alone Information Center:
On Windows, navigate to the installation directory's `doc\bin` directory, and run the `help_end.bat` file:

```
help_end.bat
```

**Note:** The `help_end` batch file contains the commands required to safely terminate the processes that were started with the `help_start` batch file. Do not use `Ctrl-C` or any other method to terminate `help_start.bat`.

On Linux, navigate to the installation directory’s `doc/bin` directory, and run the `help_end` script:

```
help_end
```

**Note:** The `help_end` script contains the commands required to safely terminate the processes that were started with the `help_start` script. Do not use any other method to terminate the `help_start` script.

7. Restart the **DB2 Information Center**.
   - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click on **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:
     ```
     /etc/init.d/db2icdv95 start
     ```

The updated **DB2 Information Center** displays the new and updated topics.

---

**DB2 tutorials**

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

**Before you begin**

You can view the XHTML version of the tutorial from the Information Center at [http://publib.boulder.ibm.com/infocenter/db2help/](http://publib.boulder.ibm.com/infocenter/db2help/)

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

**DB2 tutorials**

To view the tutorial, click on the title.

**“pureXML®” in pureXML Guide**

Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

**“Visual Explain” in Visual Explain Tutorial**

Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

---

**DB2 troubleshooting information**

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 database products.

**DB2 documentation**

Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Database fundamentals section of the DB2 Information Center. There you will find information on how to isolate and identify problems.
problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 database products.

**DB2 Technical Support Web site**

Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes, Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.


### Terms and Conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

**Personal use:** You may reproduce these Publications for your personal, non-commercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these Publications, or any portion thereof, without the express consent of IBM.

**Commercial use:** You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
Appendix B. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user’s responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

This document may provide links or references to non-IBM Web sites and resources. IBM makes no representations, warranties, or other commitments whatsoever about any non-IBM Web sites or third-party resources that may be referenced, accessible from, or linked from this document. A link to a non-IBM Web site does not mean that IBM endorses the content or use of such Web site or
its owner. In addition, IBM is not a party to or responsible for any transactions you
may enter into with third parties, even if you learn of such parties (or use a link to
such parties) from an IBM site. Accordingly, you acknowledge and agree that IBM
is not responsible for the availability of such external sites or resources, and is not
responsible or liable for any content, services, products, or other materials on or
available from those sites or resources. Any software provided by third parties is
subject to the terms and conditions of the license that accompanies that software.

IBM may use or distribute any of the information you supply in any way it
believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose
of enabling: (i) the exchange of information between independently created
programs and other programs (including this one) and (ii) the mutual use of the
information that has been exchanged, should contact:

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
CANADA

Such information may be available, subject to appropriate terms and conditions,
including in some cases payment of a fee.

The licensed program described in this document and all licensed material
available for it are provided by IBM under terms of the IBM Customer Agreement,
IBM International Program License Agreement, or any equivalent agreement
between us.

Any performance data contained herein was determined in a controlled
environment. Therefore, the results obtained in other operating environments may
vary significantly. Some measurements may have been made on development-level
systems, and there is no guarantee that these measurements will be the same on
generally available systems. Furthermore, some measurements may have been
estimated through extrapolation. Actual results may vary. Users of this document
should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of
those products, their published announcements, or other publicly available sources.
IBM has not tested those products and cannot confirm the accuracy of
performance, compatibility, or any other claims related to non-IBM products.
Questions on the capabilities of non-IBM products should be addressed to the
suppliers of those products.

All statements regarding IBM’s future direction or intent are subject to change or
withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business
operations. To illustrate them as completely as possible, the examples include the
names of individuals, companies, brands, and products. All of these names are
fictitious, and any similarity to the names and addresses used by an actual
business enterprise is entirely coincidental.

COPYRIGHT LICENSE:
This information may contain sample application programs, in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

**Trademarks**

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.
## Index

### Special characters
- only methods
  - `DB2Administrator` class 390
  - `DB2CataloguedDatabase` class 403
- only properties
  - `DB2Administrator` class 390
  - `DB2CataloguedDatabase` class 403

### A
- alternative security mechanism
  - IBM Data Server Driver for JDBC and SQLJ 178
- applets
  - JDBC building 191
  - SQLJ building 193
  - usage 194
- application development
  - high availability
    - connections to IDS 236
    - direct connections to DB2 for z/OS servers 248
  - JDBC application programming 21
  - SQLJ 113
- application programming for high availability
  - connections to DB2 Database for Linux, UNIX, and Windows 223
- applications
  - Java 2 Platform, Enterprise Edition 249
- ARRAY parameters
  - invoking stored procedures from JDBC programs 74
  - invoking stored procedures from SQLJ programs 153
- assignment-clause
  - SQLJ 363
- auto-generated keys
  - retrieving for `DELETE`, JDBC application 79
  - retrieving for `INSERT`, JDBC application 77
  - retrieving for `MERGE`, JDBC application 79
  - retrieving for `UPDATE`, JDBC application 79
  - retrieving in JDBC application 76
- autocommit modes
  - default JDBC 100
- automatic client reroute
  - client applications 215
  - DB2 for z/OS 247
  - IDS servers 232
- automatic client reroute support, client operation 220
- automatically generated keys
  - retrieving
    - `DELETE` statement, JDBC application 79
    - `INSERT` statement, JDBC application 77
    - `MERGE` statement, JDBC application 76
    - `UPDATE` statement, JDBC application 79

### B
- batch queries
  - JDBC 47
  - SQLJ 41
  - BatchUpdateException exception
    - retrieving information 107
- books
  - printed
    - ordering 516
  - CallableStatement class 58
- client affinities
  - .NET 224, 237
  - CLI 224, 237
  - IBM Data Server Driver for JDBC and SQLJ 224, 237, 238
- client affinities, example of enabling
  - Java clients 225, 238
- client application
  - automatic client reroute 215
  - high availability 215
  - transaction-level load balancing 215
- client configuration, automatic client reroute support
  - DB2 Database for Linux, UNIX, and Windows 216
  - IDS 228
  - Sysplex workload balancing
    - DB2 for z/OS 242
- client info properties
  - IBM Data Server Driver for JDBC and SQLJ 84, 85
  - JDBC 99
  - SQLJ 357
- commits
  - SQLJ transactions 163
  - transactions
    - JDBC 99
  - configuration
    - JDBC 16
    - SQLJ 16
  - configuration properties
    - customizing 16
    - details 312
  - parameters 16
  - connection context
    - class 115
    - closing 164
    - default 115
    - object 115
  - connection declaration clause
    - SQLJ 357
  - connection pooling
    - overview 259

© Copyright IBM Corp. 2006, 2010
connections
  closing
    importance 110, 164
data sources using SQLJ 115
DataSource interface 30
existing 121
containers
  Java 2 Platform, Enterprise Edition 250
countx clause
  SQLJ 359, 360

data
  retrieving
    JDBC 45
data sources
  connecting to
    DriverManager 26
    JDBC 23
    JDBC DataSource 30
data type mappings
  Java types to other types 261
DatabaseMetaData methods 35


databases
  compatibility
    IBM Data Server Driver for JDBC and SQLJ 3


DataSource interface
  SQLJ
    connection technique 3 118
    connection technique 4 120

DataSource objects
  creating 33
  deploying 33
date value adjustment
  JDBC applications 267
  SQLJ applications 267
DB2 Database for Linux, UNIX, and Windows
  client configuration, automatic client reroute support 216
  high-availability support 216
DB2 Database for Linux, UNIX, and Windows high availability support, example of enabling
  IBM Data Server Driver for JDBC and SQLJ 219
DB2 Database for Linux, UNIX, and Windows, connections application programming for high availability 223

DB2 for z/OS
  binding plans and packages 494
  client configuration, Sysplex workload balancing 242
  direct connections 246, 248
  server access from Java programs 17
  Sysplex support
    overview 240

DB2 Information Center
  languages 518
  updating 518
  versions 517
  viewing in different languages 518

DB2 JDBC Type 2 Driver
  DriverManager interface 25
  security 189
  SQLException 109
  SQLWarning 109
DB2Administrator class 390
DB2BaseDataSource class 390
DB2Binder utility 8
DB2CallableStatement interface 397
DB2CataloguedDatabase class 403


DB2ClientRenrouteServerList class 404
DB2Connection interface 405
DB2ConnectionPoolDataSource class 422
DB2DatabaseMetaData interface 424
DB2Diagnosable interface
  retrieving the SQLCA 163
DB2Diagnosable interface 425
DB2ExceptionFormatter class 426
DB2FileReference class 426
DB2JCCPlugin interface 427
DB2LobTableCreator utility 15
DB2ParameterMetaData interface 428
DB2ParameterMetaData methods 65
DB2PooledConnection interface 428
DB2PoolMonitor class 431
DB2PreparedStatement interface 434
DB2ResultSet interface 443
DB2ResultSetMetaData interface 444
DB2RowID interface 445
DB2SimpleDataSource class
  definition 33
details 445
DB2Sqlca class 446
db2sqljbind command 506
db2sqljcustomize command 494
db2sqlprint
  formation ICC customized profile 200
db2sqlprint command
details 512
  formatting information about SQLJ customized profile 199
DB2statement interface 447
DB2SystemMonitor interface 450
DB2TXAResolutionUtil 18
DB2TraceManager class 453
DB2TraceManagerMXBean interface 456
DB2Types class 459
DB2XADatasource class 460
DB2Xml interface 462
DBBatchUpdateException interface 389
default connection context 121
deregisterDB2XMLObject method 97

distinct types
  JDBC applications 73
  SQLJ applications 153
distributed transactions
  example 252
documentation
  overview 513
  PDF 514
  printed 514
terms and conditions of use 521


DriverManager interface
  DB2 JDBC Type 2 Driver 25
  SQLJ
    SQLJ connection technique 1 115
    SQLJ connection technique 2 117
drivers
  determining IBM Data Server Driver for JDBC and SQLJ version 489
dynamic data format 145


encryption
  IBM Data Server Driver for JDBC and SQLJ 171
  Enterprise Java Beans
    overview 257
environment variables
  JDBC 16
  SQLJ 16
errors
  SQLJ 163
escape syntax
  IBM Data Server Driver for JDBC and SQLJ 354
examples
  deregisterDB2XMLObject 97
  registerDB2XMLSchema 97
exceptions
  IBM Data Server Driver for JDBC and SQLJ 100
executeUpdate methods 40
extended client information 83

G
getCause method 100
databaseProductName method 36
databaseProductVersion method 36

H
calendar 518
configuring language 518
SQL statements 517
high availability
  client application 215
  IDS 227
high-availability support
  DB2 Database for Linux, UNIX, and Windows 216
host expressions
  SQLJ 122, 354

I
IBM data server clients
  automatic client reroute
    DB2 for z/OS 247
    IDS 232
IBM Data Server Driver for JDBC and SQLJ
  client info properties 84
  compatibility with databases 3
  connecting to data sources 26
  connection concentrator monitoring 205
  DB2 for z/OS servers 17
  DB2T4XAThreadPoolUtil utility 18
errors 481
example of enabling DB2 Database for Linux, UNIX, and Windows high availability support 219
example of enabling IDS high availability support 231
example of enabling Sysplex workload balancing 244
exceptions 100
extended client information 83
installing 5
JDBC extensions 388
Kerberos security 173
LOB support
  JDBC 66, 68
  SQLJ 145
properties 270
security
details 167
encrypted password 171
encrypted user ID 171

IBM Data Server Driver for JDBC and SQLJ (continued)
  security (continued)
    plug-ins 176
    user ID and password 169
    user ID-only 170
    SQL escape syntax 354
    SQLExceptions 103
    SQLSTATEs 488
    trace program example 201
    tracing with configuration parameters example 201
    trusted context support 180
    type 2 connectivity
      overview 32
    type 4 connectivity 32
    version determination 489
    warnings 100
XML support 155
IBM Data Server Driver for JDBC and SQLJ-only fields
  DB2Types class 459
IBM Data Server Driver for JDBC and SQLJ-only methods
  DB2BaseDataSource class 390
  DB2CallableStatement interface 397
  DB2ClientRerouteServerList class 404
  DB2Connection interface 405
  DB2ConnectionPoolDataSource class 422
  DB2DatabaseMetaData interface 424
  DB2Diagnostics interface 425
  DB2ExceptionFormatter class 426
  DB2FileReference class 426
  DB2JCCPlugin interface 427
  DB2ParameterMetaData interface 428
  DB2PooledConnection interface 428
  DB2PoolMonitor class 431
  DB2PreparedStatement interface 434
  DB2ResultSet interface 443
  DB2SystemMonitor interface 450
  DB2TraceManager class 453
  DB2TraceManagerMXBean interface 456
  DB2XADataSource class 460
  DB2XAResource interface 462
  DB2XAResourceException interface 389
IBM Data Server Driver for JDBC and SQLJ-only properties
  DB2BaseDataSource class 390
  DB2ClientRerouteServerList class 404
  DB2ConnectionPoolDataSource class 422
  DB2SimpleDataSource class 445
  DB2sqlca class 446
  DB2Statement interface 447
  DB2SystemMonitor interface 450
  DB2TraceManager class 453
  DB2TraceManagerMXBean interface 456
  DB2XADataSource class 460
  DB2Xml interface 462
IDSSupportClass interface 389
IBM data server drivers
  automatic client reroute
    DB2 for z/OS 247
    IDS 232
IDS
  client configuration, high-availability support 228
  high availability
    application programming 236
    cluster support 227
    workload balancing 235
  IDS high availability support, example of enabling
  IBM Data Server Driver for JDBC and SQLJ 231
implements clause
  SQLJ 355
installation
  IBM Data Server Driver for JDBC and SQLJ 5
isolation levels  
JDBC 99  
SQLJ 162
iterator conversion clause  
SQLJ 364
iterator declaration clause  
SQLJ 357
iterators  
obtaining JDBC result sets from 147  
positioned DELETE 124  
positioned UPDATE 124

J
Java
applets  
building (JDBC) 191  
building (SQLJ) 193  
using 194
applications  
accessing z/OS servers 17  
building 191  
building (JDBC) 191  
building (overview) 191  
building (SQLJ) 193  
overview 1
distributed transactions 256
Enterprise Java Beans 257
environment
customization 16
routines  
building (JDBC) 192  
building (SQLJ) 196
Java 2 Platform, Enterprise Edition
application support 249
containers 250
database requirements 251
Enterprise Java Beans 257
overview 249
requirements 251
server 250
transaction management 251
Java Naming and Directory Interface (JNDI)
details 251
Java Transaction API (JTA) 251
Java Transaction Service (JTS) 251
JDBC
4.0  
getColumnName change 477  
getColumnName change 477
accessing packages 35
APIs 326
applets  
building 191  
using 194
applications  
24 as hour value 267  
building 191
data retrieval 45
element 21
invalid Gregorian date 267
programming overview 21
transaction control 99
variables 37
batch errors 107
batch queries 47
batch updates 41

JDBC (continued)
configuring 16
collections 33
data type mappings 261
DB2 for z/OS servers 17
drivers  
details 2
 differences 464, 474, 479
environment variables 16
executeUpdate methods 40
executing SQL 37
extensions 388
IBM Data Server Driver for JDBC and SQLJ installation 5
isolation levels 99
named parameter markers 80, 82
named parameters 60
objects  
creating 38
modifying 38
optimistic locking 87
problem diagnosis 199
ResultSet holdability 51
resultSets  
delete holes 56
holdability 50
inserting row 57, 58
routines  
building (procedure) 192
scrollable ResultSet 50, 51
SQLWarning 106
transactions  
committing 99
default auto-commit modes 100
rolling back 99
updatable ResultSet 50, 51
JNDI (Java Naming and Directory Interface)
details 251
JTA (Java Transaction API) 251
JTS (Java Transaction Service) 251

K
Kerberos authentication protocol  
IBM Data Server Driver for JDBC and SQLJ 173

L
large objects (LOBs)  
compatible Java data types  
JDBC applications 70
SQLJ applications 146
IBM Data Server Driver for JDBC and SQLJ 66, 68, 145
locators  
IBM Data Server Driver for JDBC and SQLJ 67, 68
SQLJ 145

M
monitoring  
system  
IBM Data Server Driver for JDBC and SQLJ 209
multi-row operations 54
multiple result sets  
retrieving from stored procedure in JDBC application  
keeping result sets open 64
known number 62
multiple result sets (continued)
  retrieving from stored procedure in JDBC application
  (continued)  
  overview 62
  unknown number 63
retrieving from stored procedure in SQLJ application 144

N
named iterators
  passed as variables 128
result set iterator 134
named parameter markers
  CallableStatement objects 82
  JDBC 80
  PreparedStatement objects 80
named parameters
  CALL statement
  JDBC 60
notices 523

O
optimistic locking
  JDBC applications 87
ordering DB2 books 516

P
packages
  JDBC 35
  SQLJ 121
ParameterMetaData methods 44
positioned deletes
  SQLJ 124
positioned iterators
  passed as variables 128
  result set iterators 136
positioned updates
  SQLJ 124
PreparedStatement methods
  SQL statements with no parameter markers 39
  SQL statements with parameter markers 39, 46
problem determination
  information available 520
  JDBC 199
  SQLJ 199
tutorials 520
progressive streaming
  IBM Data Server Driver for JDBC and SQLJ 66, 68
  JDBC 145
properties
  IBM Data Server Driver for JDBC and SQLJ
    customizing 16
    for all database products 271
    for DB2 Database for Linux, UNIX, and Windows 298, 300
    for DB2 for z/OS 302
    for DB2 servers 290
    for IDS 298, 300, 306
    overview 270

R
reference information
  Java 261
registerDB2XMLSchema method 97
remote trace controller
  accessing 212
  enabling 211
  overview 211
resources
  releasing
    closing connections 110, 164
restrictions
  SQLJ variable names 122
result set iterator
  public declaration in separate file 148
result set iterators
  details 133
  generating JDBC ResultSets from SQLJ iterators 147
  named 134
  positioned 136
  retrieving data from JDBC result sets using SQLJ
  iterators 147
ResultSet
  holdability 50
  inserting row 57
  testing for delete hole 56
  testing for inserted row 58
ResultSet holdability
  JDBC 51
ResultSetMetaData methods
  ResultSetMetaData.getColumnLabel change in value 477
  ResultSetMetaData.getColumnName change in value 477
retrieving result set information 49
retrieving data
  JDBC
    data source information 35
    PreparedStatement.executeQuery method 46
    result set information 49
    tables 45
  SQLJ 133, 138, 139
retrieving parameter information
  JDBC 44
retrieving parameter names
  JDBC 65
retrieving SQLCA
  DB2Diagnosable class 163
return codes
  IBM Data Server Driver for JDBC and SQLJ errors 481
rollbacks
  JDBC transactions 99
  SQLJ transactions 163
routines
  invoking
    XML parameters in Java applications 95
ROWID 151

S
savepoints
  JDBC applications 75
  SQLJ applications 154
scrollable iterators
  SQLJ 140
scrollable ResultSet
  JDBC 51

Index 531
scrollable ResultSet
JDBC 50
SDKs
   differences 481
   version 1.5 160
security
   DB2 JDBC Type 2 Driver 189
   IBM Data Server Driver for JDBC and SQLJ
      encrypted security-sensitive data 171
      encrypted user ID or encrypted password 171
      Kerberos 173
      security mechanisms 167
      user ID and password 169
      user ID only 170
plug-ins
   JDBC support 176
SQLJ program preparation 186
SET TRANSACTION clause 362
setTransactionTimeout method 256
SQL statements
   displaying help 517
   error handling
      SQLJ applications 163
   executing
      JDBC interfaces 37
   SQLJ applications 124, 151
SQLException
   DB2 JDBC Type 2 Driver 109
   IBM Data Server Driver for JDBC and SQLJ 103
SQLJ
   accessing packages for 121
   applets
      building 193
      using 194
   applications
      24 as hour value 267
      building 193
      compiler options (UNIX) 195
      compiler options (Windows) 195
      examples 113
      invalid Gregorian date 267
      programming 113
   transaction control 162
assignment clause 363
batch updates 129
building routines 196
calling stored procedures 143
classes 354
collecting trace data 199
comments 123
connecting to data source 115
connecting using default context 121
connection declaration clause 357
context clause 359, 360
DataSource interface 118, 120
DB2 tables
   creating 124
   modifying 124
DriverManager interface 115, 117
drivers 2
environment variables 16
error handling 163
executable clauses 359
executing SQL 124
execution context 151
execution control 151
existing connections 121

SQLJ (continued)
host expressions 122, 354
   implements clause 355
installing runtime environment 16
   isolation levels 162
iterator conversion clause 364
iterator declaration clause 357
multiple instances of iterator 139
multiple iterators on table 138
problem diagnosis 199
Profile Binder command 506
Profile Printer command 512
result set iterator 133
retrieving SQLCA 163
routines
   compiler options (UNIX) 196
   compiler options (Windows) 197
scrollable iterators 140
SDK for Java Version 5 functions 160
security 186
SET TRANSACTION clause 362
SQLWarning 164
statement reference 354
transactions 163
translator command 491
variable names 122
with-clause 355
sqlj command 491
SQLJ variable names
   restrictions 122
sqlj.runtime package 364
sqlj.runtime.AsciiStream 376, 386
sqlj.runtime.BinaryStream 376
sqlj.runtime.CharacterStream 377
sqlj.runtime.ConnectionContext 365
sqlj.runtime.ExecutionContext 378
sqlj.runtime.ForUpdate 370
sqlj.runtime.NamedIterator 370
sqlj.runtime.PositionedIterator 371
sqlj.runtime.ResultSetIterator 371
sqlj.runtime.Scrollable 374
sqlj.runtime.SQLNullException 386
sqlj.runtime.UnicodeStream 387
SQLSTATE
   IBM Data Server Driver for JDBC and SQLJ errors 488
SQLWarning
   DB2 JDBC Type 2 Driver 109
   IBM Data Server Driver for JDBC and SQLJ 106
SSL
   configuring
      Java Runtime Environment 183
      IBM Data Server Driver for JDBC and SQLJ 182
      sslConnection property 182
      sslConnection property 182
   Statement.executeQuery 45
stored procedures
   calling
      CallableStatement class 58
      JDBC applications 74
      SQLJ applications 143, 153
      DB2 for z/OS 58
   keeping result sets open in JDBC applications 64
<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>terms and conditions 521</td>
</tr>
<tr>
<td>time value adjustment</td>
</tr>
<tr>
<td>JDBC applications 267</td>
</tr>
<tr>
<td>SQLJ applications 267</td>
</tr>
<tr>
<td>trace controller 211</td>
</tr>
<tr>
<td>traces</td>
</tr>
<tr>
<td>IBM Data Server Driver for JDBC and SQLJ 199, 201</td>
</tr>
<tr>
<td>transaction control</td>
</tr>
<tr>
<td>JDBC 99</td>
</tr>
<tr>
<td>SQLJ 162</td>
</tr>
<tr>
<td>transaction-level load balancing</td>
</tr>
<tr>
<td>client application 215</td>
</tr>
<tr>
<td>troubleshooting</td>
</tr>
<tr>
<td>online information 520</td>
</tr>
<tr>
<td>tutorials 520</td>
</tr>
<tr>
<td>trusted contexts</td>
</tr>
<tr>
<td>JDBC support 180</td>
</tr>
<tr>
<td>tutorials</td>
</tr>
<tr>
<td>problem determination 520</td>
</tr>
<tr>
<td>troubleshooting 520</td>
</tr>
<tr>
<td>Visual Explain 520</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
</tr>
<tr>
<td>SQLJ applications 195</td>
</tr>
<tr>
<td>SQLJ routines 196</td>
</tr>
<tr>
<td>updatable ResultSet</td>
</tr>
<tr>
<td>inserting row 57</td>
</tr>
<tr>
<td>JDBC 50, 51</td>
</tr>
<tr>
<td>testing for delete hole 56</td>
</tr>
<tr>
<td>testing for inserted row 58</td>
</tr>
<tr>
<td>updates</td>
</tr>
<tr>
<td>PreparedStatement.executeUpdate method 39</td>
</tr>
<tr>
<td>DB2 Information Center 518</td>
</tr>
<tr>
<td>URL format</td>
</tr>
<tr>
<td>DB2BaseDataSource class 27, 29</td>
</tr>
<tr>
<td>user ID and password security</td>
</tr>
<tr>
<td>IBM Data Server Driver for JDBC and SQLJ 169</td>
</tr>
<tr>
<td>user ID-only security</td>
</tr>
<tr>
<td>IBM Data Server Driver for JDBC and SQLJ 170</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Explain</td>
</tr>
<tr>
<td>tutorial 520</td>
</tr>
</tbody>
</table>

W  
- warnings  
  - IBM Data Server Driver for JDBC and SQLJ 100
  - Windows  
    - SQLJ applications 195
    - SQLJ routines 197
  - with clause  
    - SQLJ 355
X  
- XML  
  - IBM Data Server Driver for JDBC and SQLJ 155
  - parameters  
    - invoking routines from Java programs 95
  - XML data  
    - Java applications 89
    - updating  
      - tables in Java applications 90, 156
  - XML data retrieval  
    - Java applications 92, 157
  - XML schemas  
    - registering 97
    - removing 97
  - XMLCAST  
    - SQLJ applications 159