DB2

DB2 Version 9
for Linux, UNIX, and Windows

Developing Java Applications

DB2 9 BETA

SC10-4233-00

DB2®

IBM

DB2 Version 9
for Linux, UNIX, and Windows

Developing Java Applications

DB2 9 BETA

SC10-4233-00

Before using this information and the product it supports, be sure to read the general information under *Notices*.

Edition Notice

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).

Contents

Chapter 1. Introduction

The following topics introduce Java application support for the DB2 database system and explain how to configure Java application support.

- "Introduction to Java application development for DB2"
- "Supported drivers for JDBC and SQLJ"
- "Supported Java application development software" on page 3
- "Setting up the DB2 JDBC and SQLJ development environment" on page 4

Introduction to Java application development for DB2

The DB2® database system provides driver support for client applications and applets that are written in Java™ using JDBC, and for embedded SQL for Java (SQLJ).

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

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.

In general, Java applications use JDBC for dynamic SQL and SQLJ for static SQL. However, because SQLJ can inter-operate with JDBC, an application program can use JDBC and SQLJ within the same unit of work.

Related concepts:

- "Supported drivers for JDBC and SQLJ" on page 1
- "Supported Java application development software" on page 3

Supported drivers for JDBC and SQLJ

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 support 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 server using a

database-independent protocol. The server then communicates the client's requests to the data source. The DB2 database system does not support 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 Version 9.1 supports a driver that combines type 2 and type 4 JDBC implementations. DB2 Version 9.1 also supports a .type 2 driver, although this support is deprecated. The DB2 Version 9.1 type 2 driver continues to use the DB2 CLI interface to communicate with DB2 database servers. The drivers that are supported in DB2 Version 9.1 are:

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 translated to Java native methods. The Java applications that use this driver must run on a DB2 client, through which JDBC requests flow to the DB2 server. DB2 Connect™ Version 9.1 must be installed before the DB2 JDBC application driver can be used to access DB2 UDB for iSeries data sources or data sources in the DB2 for OS/390® or 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. See Comparison of driver support for JDBC APIs.
- SQLJ statements that perform equivalent functions 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 of the DB2 database system. You should therefore consider moving to the IBM DB2 Driver for JDBC and SQLJ.

IBM DB2 Driver for JDBC and SQLJ (type 2 and type 4):

The IBM DB2 Driver for JDBC and SQLJ is a single driver that includes JDBC type 2 and JDBC type 4 behavior, as well as SQLJ support. When an application loads the IBM DB2 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 DB2 Driver for JDBC and SQLJ type 2 driver behavior is referred to as *IBM DB2 Driver for JDBC and SQLJ type 2 connectivity.* IBM DB2 Driver for JDBC and SQLJ type 4 driver behavior is referred to as *IBM DB2 Driver for JDBC and SQLJ type 4 connectivity.*

The IBM DB2 Driver for JDBC and SQLJ supports these JDBC and SQLJ functions:
- All of the methods that are described in the JDBC 3.0 specifications. See Comparison of driver support for JDBC APIs.
- SQLJ statements that perform equivalent functions to most JDBC methods.
- Connections that are enabled for connection pooling. WebSphere® Application Server or another application server does the connection pooling.

- Java user-defined functions and stored procedures (IBM DB2 Driver for JDBC and SQLJ type 2 connectivity only).
- Global transactions that run under WebSphere Application Server Version 5.0 and above.
- 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) .

Related concepts:
- "How JDBC applications connect to a data source" on page 24
- "Security under the IBM DB2 Driver for JDBC and SQLJ" on page 142

Related reference:
- "Driver support for JDBC APIs" on page 247
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335
- "SQLJ differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 342

Supported Java application development software

To develop and deploy Java applications that run against DB2 databases, you need to use supported development software and operating systems.

When you install the IBM DB2 Driver for JDBC and SQLJ, the driver installation process does not install a software development kit (SDK) for Java. If the installation process for the DB2 Database for Linux, UNIX, and Windows product installs an SDK for Java, that installation process installs the latest SDK for Java that is available.

The following table lists the supported levels of the SDK for Java. Only the listed levels and forward-compatible later versions of the same level are supported. For example, if only level 1.4.2 is listed for a particular operating system, level 1.4.2 SR*n* is also supported, but level 5 is not supported. If level 1.4.2 to level 5 is listed, level 1.4.2 SR*n* and level 5 SR*m* are also supported.

Because there are frequent SDK for Java fixes and updates, not all levels and versions have been tested. If your database application has problems that are related to the SDK for Java, try the next available version of your SDK for Java at the given level.

Non-IBM versions of the SDK for Java are supported only for building and running stand-alone Java applications. For building and running Java stored procedures and user-defined functions, only the IBM SDK for Java that is included with the DB2 Database for Linux, UNIX, and Windows product is supported.

Table 1. SDK for Java by DB2 Database for Linux, UNIX, and Windows

Operating system	31-bit, 32-bit, or 64-bit operating system	Supported levels of the SDK for Java
AIX® 5	32-bit/64-bit	1.4.2 to 5
HP-UX 11i	32-bit/64-bit	1.4.2[1]

Table 1. SDK for Java by DB2 Database for Linux, UNIX, and Windows (continued)

Operating system	31-bit, 32-bit, or 64-bit operating system	Supported levels of the SDK for Java
Linux on Intel® x86	32-bit	1.4.2 to 5[2]
Linux on IA64	64-bit	1.4.2[2,3]
Linux on AMD64/EM64T	32-bit/64-bit	1.4.2 to 5[2]
Linux on PowerPC®	32-bit/64-bit	1.4.2 to 5
Linux on zSeries®	31-bit/64-bit	1.4.2 to 5
Solaris	32-bit/64-bit	1.4.2[2]
Windows on Intel x86	32-bit	1.4.2 to 5[2]
Windows on IA64	64-bit	1.4.2[2,3]
Windows on x64	32-bit/64-bit	1.4.2 to 5[2]

Notes:

1. The same levels of the SDK for Java that are available from Hewlett-Packard are supported for building and running client applications with the IBM DB2 Driver for JDBC and SQLJ.
2. The same levels of the SDK for Java that are available from Sun Microsystems are supported for building and running client applications with the IBM DB2 Driver for JDBC and SQLJ.
3. A minimum level of SDK for Java 1.4.2 SR3 is required for Montecito processors.

Related tasks:

- "Installing the IBM DB2 Driver for JDBC and SQLJ" on page 4

Setting up the DB2 JDBC and SQLJ development environment

The following topics contain information on setting up the environment for Java application programming on DB2 Database for Linux, UNIX, and Windows.

- "Installing the IBM DB2 Driver for JDBC and SQLJ"
- "DB2Binder utility" on page 8
- "DB2LobTableCreator utility" on page 10
- "IBM DB2 Driver for JDBC and SQLJ configuration properties customization" on page 11
- "Special setup for accessing DB2 for z/OS servers from Java programs" on page 15
- "DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers" on page 16
- "Special setup for running Java routines in the HP-UX environment" on page 19

Installing the IBM DB2 Driver for JDBC and SQLJ

Follow these steps to install the IBM DB2 Driver for JDBC and SQLJ.

Prerequisites:

- An SDK for Java, 1.4.2 or later.

  For all DB2 products except the DB2 Runtime Client, the installation process automatically or optionally installs an SDK for Java.
- 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.

- Support for accessing DB2 for z/OS database servers

  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.
- Unicode support for iSeries™ servers

  If any SQLJ or JDBC programs will use IBM DB2 Driver for JDBC and SQLJ type 4 connectivity to connect to a DB2 UDB for iSeries server, the OS/400® operating system must support the Unicode UTF-8 encoding scheme. The following table lists the OS/400 PTFs that you need for Unicode UTF-8 support:

Table 2. OS/400 PTFs for Unicode UTF-8 support

OS/400 version	PTF numbers
V5R3 or later	None (support is included)
V5R2	SI06541, SI06796, SI07557, SI07564, SI07565, SI07566, and SI07567
V5R1	SI06308, SI06300, SI06301, SI06302, SI06305, SI06307, and SI05872

- Java support for HP-UX clients and servers

  HP-UX servers: The IBM DB2 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 DB2 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 DB2 Driver for JDBC and SQLJ.

  See Special setup for running Java routines in the HP-UX environment for details.

Procedure:

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:
   - Install the IBM DB2 Driver for JDBC and SQLJ class files, and to modify 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 files names are db2jcc.jar and sqlj.zip. You need only db2jcc.jar for preparing and executing JDBC programs. You need db2jcc.jar and sqlj.zip for preparing and executing SQLJ programs.
   - Install IBM DB2 Driver for JDBC and SQLJ license files, and modify 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 3. IBM DB2 Driver for JDBC and SQLJ license files

License file	Server to which license file permits a connection	Product that includes license file
db2jcc_license_c.jar	Cloudscape™	Cloudscape Network Server
db2jcc_license_cu.jar	Cloudscape All DB2 Database for Linux, UNIX, and Windows servers	All DB2 Database for Linux, UNIX, and Windows products
db2jcc_license_cisuz.jar	Cloudscape All DB2 Database for Linux, UNIX, and Windows servers DB2 for z/OS DB2 UDB for iSeries	All DB2 Connect products

- Install IBM DB2 Driver for JDBC and SQLJ native libraries for support of IBM DB2 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:

  libdb2jcct2.so
  For AIX, HP-UX on IPF, Linux, and Solaris

  libdb2jcct2.sl
  For HP-UX on PA-RISC

  db2jcct2.dll
  For Windows

2. Customize the driver-wide configuration properties, if any of the defaults are inappropriate. See IBM DB2 Driver for JDBC and SQLJ configuration properties customization for details.
3. Configure TCP/IP

   Servers must be configured for TCP/IP communication in the following cases:
   - JDBC or SQLJ applications that use IBM DB2 Driver for JDBC and SQLJ type 4 connectivity.
   - JDBC or SQLJ applications that use IBM DB2 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
   ```
   You must execute the db2stop and db2start commands for this setting to take effect.

   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.
4. 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/jdk142
```

/home/db2inst/jdk142 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\jdk142
```

c:\Program Files\jdk142 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.

5. 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.
6. If you plan to run Java stored procedures that work with XML data on DB2 Database for Linux, UNIX, and Windows servers, you need to set the IBM DB2 Driver for JDBC and SQLJ as the default JDBC driver for running stored procedures. To do that, set the DB2_USE_DB2JCCT2_JROUTINE environment value to YES, yes, ON, on, TRUE, true, or 1. For example:

   To set the IBM DB2 Driver for JDBC and SQLJ as the default driver at the instance level:
   ```
   db2set DB2_USE_DB2JCCT2_JROUTINE=YES -i instance-name
   ```
   To set the IBM DB2 Driver for JDBC and SQLJ as the default driver at the global level:
   ```
   db2set DB2_USE_DB2JCCT2_JROUTINE=YES -g
   ```
7. If you plan to use Kerberos security, put the following files in the Java application CLASSPATH:
   - ibmjceprovider.jar
   - ibmjcefw.jar
   - ibmjlog.jar
   - US_export_policy.jar
   - Local_policy.jar
   - ibmjgssprovider.jar
   - jaas.jar
   - ibmjceprovider.jar
   - ibmjcefw.jar
   - ibmjlog.jar
   - US_export_policy.jar
   - Local_policy.jar

8. If you intend to connect to a DB2 for z/OS server, run the com.ibm.db2.jcc.DB2Binder utility to bind the DB2 packages that are used at the server by the IBM DB2 Driver for JDBC and SQLJ. See DB2Binder utility for details.
9. Determine whether you need to use LOB locators to access the following types of data on DB2 for z/OS servers:
   - Data in DBCLOB columns
   - Data in CLOB columns

   If so, you need to create tables on the database servers that are needed for fetching data from DBCLOB or CLOB columns using LOB locators. Create the tables in one of the following ways:
   - On the DB2 for z/OS servers, customize and run job DSNTIJMS. That job is located in data set *prefix*.SDSNSAMP.
   - On the client, run the com.ibm.db2.jcc.DB2LobTableCreator utility against each of the DB2 for z/OS servers. See DB2LobTableCreator utility for details.
10. If you plan to use IBM DB2 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. See DB2T4XAIndoubtUtil utility for details.

Related concepts:
- "IBM DB2 Driver for JDBC and SQLJ configuration properties customization" on page 11
- "Supported Java application development software" on page 3

Related tasks:
- "Special setup for running Java routines in the HP-UX environment" on page 19
- "Configuring TCP/IP communications for a DB2 instance" in *Installation and Configuration Supplement*
- "Updating the database manager configuration file on the server for TCP/IP communications" in *Installation and Configuration Supplement*
- "Updating the services file on the server for TCP/IP communications" in *Installation and Configuration Supplement*
- "Special setup for accessing DB2 for z/OS servers from Java programs" on page 15

Related reference:
- "DB2Binder utility" on page 8
- "DB2LobTableCreator utility" on page 10
- "IBM Software Development Kit for Java levels for DB2 products" in *Quick Beginnings for DB2 Servers*
- "DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers" on page 16
- "DB2 Connect product offerings" in *DB2 Connect User's Guide*

DB2Binder utility

The DB2Binder utility binds the DB2 packages that are used at the database server by the IBM DB2 Driver for JDBC and SQLJ, and grants EXECUTE authority on the packages to PUBLIC.

DB2Binder syntax:

```
>>--java--com.ibm.db2.jcc.DB2Binder---url jdbc:db2://server--+----------+--/database---user user-ID------->
                                                             '-:--port--'

>--password password--+--------------+--+-------------------------------+-------------------------------->
                      '--size integer-'  '--collection collection-name--'

                                              .-action--add-------.
>--+-------------------------------------+--+-------------------+--+-------+--------------------------><
   |                  .-,--------------. |  '-action--replace---'  '--help-'
   |                  V                | |
   '--tracelevel -----trace-option-----+-'
```

DB2Binder option descriptions:

-url
: Specifies the data source at which the JCC packages are to be bound. The variable parts of the `-url` value are:

  server
  : The domain name or IP address of the MVS™ 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.

  database
  : The location name for the DB2 subsystem, as defined in the SYSIBM.LOCATIONS catalog table.

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

-size
: Specifies the number of DB2 packages that DB2binder binds for each of the four DB2 isolation levels and each of the two holdability values. The IBM DB2 Driver for JDBC and SQLJ uses these packages to process dynamic SQL. In addition, the DB2binder binds a single package that the IBM DB2 Driver for JDBC and SQLJ uses for static SQL. Therefore, the total number of packages that DB2binder binds is:

  `4*2*`*`integer`*`+1`

  The default value for *integer* is 3.

-collection
: Specifies the collection ID for the packages that are used by an instance of the IBM DB2 Driver for JDBC and SQLJ. The default is NULLID. DB2binder translates this value to uppercase.

  You can create multiple instances of the JCC package set at a single location 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 DB2 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.

-action
Specifies whether the IBM DB2 Driver for JDBC and SQLJ packages can be replaced.

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.

Reference Text

Related tasks:
- "Installing the IBM DB2 Driver for JDBC and SQLJ" on page 4

DB2LobTableCreator utility

The DB2LobTableCreator utility creates tables on a DB2 for z/OS database server that are required by JDBC or SQLJ applications that access the following types of data using LOB locators:
- Data in DBCLOB columns
- Data in CLOB columns

DB2LobTableCreator syntax:

```
►►──java──com.ibm.db2.jcc.DB2LobTableCreator──-url──jdbc:db2:─//server──┬────────┬──/database──────────►
                                                                        └─:port──┘

►──-user──user-ID──-password──password──┬───────┬────────────────────────────────────────────────────►◄
                                        └─-help─┘
```

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:

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

-user
: Specifes the user ID under which DB2LobTableCreator is to run. This user must have authority to create tables in the DSNATPDB database.

-password
: Specifes 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.

Related tasks:

- "Installing the IBM DB2 Driver for JDBC and SQLJ" on page 4

IBM DB2 Driver for JDBC and SQLJ configuration properties customization

The IBM DB2 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 DB2 Driver for JDBC and SQLJ configuration property setting is of this form:

```
property=value
```

property can have one or more of the following forms:

- db2.jcc.override.*property-name*
- db2.jcc.*property-name*
- db2.jcc.default.*property-name*

If the configuration property begins with db2.jcc.override, the configuration property is applicable to all connections and overrides any `Connection` or `DataSource` property with the same *property-name*. If the configuration property begins with db2.jcc or db2.jcc.default, the configuration property value is a default. `Connection` or `DataSource` property settings override that value.

You can set configuration properties in the following ways:

- Set the configuration properties as Java system properties. Those settings override any other settings.

  For stand-alone Java applications, you can set the configuration properties as Java system properties by specifying -D*property=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 DB2 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 (*udddd*) characters, you do not need to do character conversion before the IBM DB2 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.

You can set any of the following IBM DB2 Driver for JDBC and SQLJ configuration properties. All properties are optional.

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 a DB2 server.

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 property applies only to IBM DB2 Driver for JDBC and SQLJ type 4 connectivity to a DB2 for z/OS server.

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.

The data type of db2.jcc.dumpPool is int. 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.

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.

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.

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.

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.

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.

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.

db2.jcc.traceDirectory or db2.jcc.override.traceDirectory

Enables the IBM DB2 Driver for JDBC and SQLJ trace for Java driver code, and specifies a directory into which trace information is written. When db2.jcc.override.traceDirectory is specified, trace information for multiple connections on the same `DataSource` is written to multiple files.

When db2.jcc.override.traceDirectory is specified, a connection is traced to a file named *file-name_origin_n*.

n is the *n*th 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 DB2 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.sqljUncustomizedWarningOrException

Specifies the action that the IBM DB2 Driver for JDBC and SQLJ takes when an uncustomized SQLJ application runs. db2.jcc.sqljUncustomizedWarningOrException can have the following values:

0
The IBM DB2 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 DB2 Driver for JDBC and SQLJ throws a `Warning` when an uncustomized SQLJ application is run.

2 The IBM DB2 Driver for JDBC and SQLJ throws an `Exception` when an uncustomized SQLJ application is run.

db2.jcc.traceFile or db2.jcc.override.traceFile

Enables the IBM DB2 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 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 DB2 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. 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.

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.

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

Special setup for accessing DB2 for z/OS servers from Java programs

Follow these steps if you plan to write JDBC or SQLJ applications that access DB2 for z/OS database servers.

Procedure:

1. Install DB2 for z/OS stored procedures.

   If any JDBC or SQLJ applications will connect to a DB2 for z/OS server, a number of stored procedures need to be installed on that server to support retrieval of DB2 catalog information, tracing, and error message formatting. The stored procedures are:
   - SQLCOLPRIVILEGES
   - SQLCOLUMNS
   - SQLFOREIGNKEYS
   - SQLGETTYPEINFO
   - SQLPRIMARYKEYS

- SQLPROCEDURECOLS
- SQLPROCEDURES
- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS
- SQLCAMESSAGE

The following DB2 for z/OS PTFs provide the latest versions of the stored procedures:

Table 4. PTFs for DB2 for z/OS stored procedures

DB2 for z/OS	PTF numbers
Version 7	UQ72083, UQ93889
Version 8	UQ93890

Ask your DB2 for z/OS system administrator whether these stored procedures are installed.

2. Create DB2 for z/OS tables.

   If any JDBC or SQLJ applications will connect to a DB2 for z/OS server, the following tables need to be installed on that server to support efficient storing of data in CLOB or DBCLOB columns:
   - SYSIBM.SYSDUMMYU
   - SYSIBM.SYSDUMMYA
   - SYSIBM.SYSDUMMYE

   Jobs that define the tables are shipped in the following PTFs:

Table 5. PTFs for DB2 for z/OS

DB2 for z/OS Version	PTF number
Version 7	UQ86843
Version 8	UQ86844

   Ask your DB2 for z/OS system administrator whether these tables are defined.

3. Enable Unicode support for OS/390 and z/OS servers.

   If any SQLJ or JDBC programs will use IBM DB2 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.

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 DB2 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. This utility 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" on page 18 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──com.ibm.db2.jcc.DB2T4XAIndoubtUtil──-url──jdbc:db2:─//server──┬────────┬──/database──────────►
                                                                          └─:port──┘

►─-user──user-ID──-password──password──┬──────────────────────┬──┬───────┬──┬─────────┬──┬───────────┬──►
                                       └─-owner──owner-ID─────┘  └─-help─┘  └─-delete─┘  └─-bindonly─┘

                    ┌─-jdbcCollection──NULLID─────────┐
►──┬──────────┬─────┼─────────────────────────────────┼──────────────────────────────────────────────────►◄
   └─-showSQL─┘     └─-jdbcCollection──collection-ID──┘
```

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
: Specifes 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
Specifes 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.

-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 *collection-name* | **NULLID**
Specifies the value of the -collection parameter that was used when the IBM DB2 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:

```
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
```

Related tasks:

- "Installing the IBM DB2 Driver for JDBC and SQLJ" on page 4

Special setup for running Java routines in the HP-UX environment

For the HP-UX operating system on PA-RISC processors, you have extra prerequisites for running Java stored procedures and user-defined functions. *In addition to* the prerequisites in "Installing the IBM DB2 Driver for JDBC and SQLJ" on page 4, you need to perform the following prerequisite steps:

1. Enable the db2hpjv tool by issuing the following commands on the command line:

   ```
   db2hpjv -e
   db2stop
   db2start
   ```

   If you need to disable db2hpjv, issue these commands:

   ```
   db2hpjv -d
   db2stop
   db2start
   ```

   Java **must** be installed on the operating system before you issue `db2hpjv -e`. DB2 Database for Linux, UNIX, and Windows cannot run on HP-UX if Java routine support is enabled, but Java is not on the operating system.
2. Give the HP-UX run-time linker access to Java shared libraries.

   To run Java stored procedures or user-defined functions, the HP-UX run-time linker must be able to access certain Java shared libraries, and the DB2 system must be able to load these libraries and the JVM. Because the program that does this loading runs with setuid privileges, it looks for the dependent libraries **only** in `/usr/lib/pa20_64`. To create access to the Java shared libraries, choose one of the following methods:

   - Create symbolic links to the Java shared libraries. To do that, log in as root, and issue the following commands to create symbolic links to the Java shared libraries:

     ```
     ln -s /opt/java1.4/jre/lib/PA_RISC2.0W/*.sl /usr/lib/pa20_64
     ln -s /opt/java1.4/jre/lib/PA_RISC2.0W/hotspot/*.sl /usr/lib/pa20_64
     ```

     These commands create symbolic links to the following libraries:

     ```
     /opt/java1.4/jre/lib/PA_RISC2.0W/libnet.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libzip.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/librmi.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libnio.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libverify.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libmlib_image.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libhprof.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libjaas_unix.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libawt.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libcmm.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libdcpr.sl
     /opt/java1.4/jre/lib/PA_RISC2.0W/libdt_socket.sl
     ```

```
/opt/java1.4/jre/lib/PA_RISC2.0W/libfontmanager.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libioser12.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libmawt.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libjsound.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libjava.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libjawt.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libjcov.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libjcpm.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libjdwp.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/libjpeg.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/hotspot/libjsig.sl
/opt/java1.4/jre/lib/PA_RISC2.0W/hotspot/libjvm.sl
```

- Add the /opt/java1.4/jre/lib/PA_RISC2.0W and opt/java1.4/jre/lib/PA_RISC2.0W/hotspot directories to the /etc/dld.sl.conf file, **and** to the SHLIB_PATH environment.

If the DB2 server cannot find the shared Java libraries when it executes a Java routine, it generates a -4300 error.

Related concepts:

- "Java sample programs" in *Samples Topics*
- "Java applet considerations" on page 163
- "The DB2 database application development environment" in *Getting Started with Database Application Development*

Related tasks:

- "Installing the IBM DB2 Driver for JDBC and SQLJ" on page 4

Chapter 2. Programming JDBC applications

The topics that follow contain information about writing JDBC applications.

Basic steps in writing a JDBC application

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.

Figure 1 on page 22 is a simple program that demonstrates each task. This program runs on the IBM DB2 Driver for JDBC and SQLJ.

```
import java.sql.*;                                                        1

public class EzJava
{
  public static void main(String[] args)
  {
    String urlPrefix = "jdbc:db2:";
    String url;
    String empNo;                                                         2
    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 DB2 Driver for JDBC and SQLJ topic.
    // For example, for IBM DB2 Driver for JDBC and SQLJ type 2 connectivity,
    // args[0] might be MVS1DB2M. For
    // type 4 connectivity, args[0] might
    // be //stlmvs1:10110/MVS1DB2M.
    if (args.length==0)
    {
      System.err.println ("Invalid value. First argument appended to "+
       "jdbc:db2: must specify a valid URL.");
      System.exit(1);
    }
    url = urlPrefix + args[0];

    try
    {
      // Load the IBM DB2 Driver for JDBC and SQLJ
      Class.forName("com.ibm.db2.jcc.DB2Driver");                         3a
      System.out.println("**** Loaded the JDBC driver");

      // Create the connection using the IBM DB2 Driver for JDBC and SQLJ
      con = DriverManager.getConnection (url);                            3b
      // Commit changes manually
      con.setAutoCommit(false);
      System.out.println("**** Created a JDBC connection to the data source");

      // Create the Statement
      stmt = con.createStatement();                                       4a
      System.out.println("**** Created JDBC Statement object");

      // Execute a query and generate a ResultSet instance
      rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");               4b
      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");
```

Figure 1. Simple JDBC application (Part 1 of 2)

```
      // 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();                                                  6
      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)                                          5
    {
      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
      }
    }
  }  // End main
}    // End EzJava
```

Figure 1. Simple JDBC application (Part 2 of 2)

Notes® to Figure 1 on page 22:

Note	Description
1	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 Access Java packages for JDBC support.
2	`String` variable empNo performs the function of a host variable. That is, it is used to hold data retrieved from an SQL query. See Declare variables in JDBC applications for more information.
3a and 3b	These two sets of statements demonstrate how to connect to a data source using one of two available interfaces. See Connect to a data source using JDBC for more details.
4a and 4b	These two sets of statements demonstrate how to perform a SELECT in JDBC. For information on how to perform other SQL operations, see Execute SQL in a JDBC application.
5	This `try/catch` block demonstrates the use of the `SQLException` class for SQL error handling. For more information on handling SQL errors, see Handle an SQLException under the IBM DB2 Driver for JDBC and SQLJ. For information on handling SQL warnings, see Handle SQL warnings in a JDBC application.
6	This statement disconnects the application from the data source. See Close the connection to the data source.

Related concepts:

- "Java packages for JDBC support" on page 35

- "How JDBC applications connect to a data source" on page 24
- "Variables in JDBC applications" on page 37
- "JDBC interfaces for executing SQL" on page 38

Related tasks:

- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77
- "Handling an SQLWarning under the IBM DB2 Driver for JDBC and SQLJ" on page 81

Connecting to database servers in JDBC applications

The following topics contain information on connection to DB2 Database for Linux, UNIX, and Windows database servers.

How JDBC applications connect to a data source

Before you can execute SQL statements in any SQL program, you must connect to a database server. In JDBC, a database server is known as a *data source*.

Figure 2 shows how a Java application connects to a data source for a type 2 driver or IBM DB2 Driver for JDBC and SQLJ type 2 connectivity.

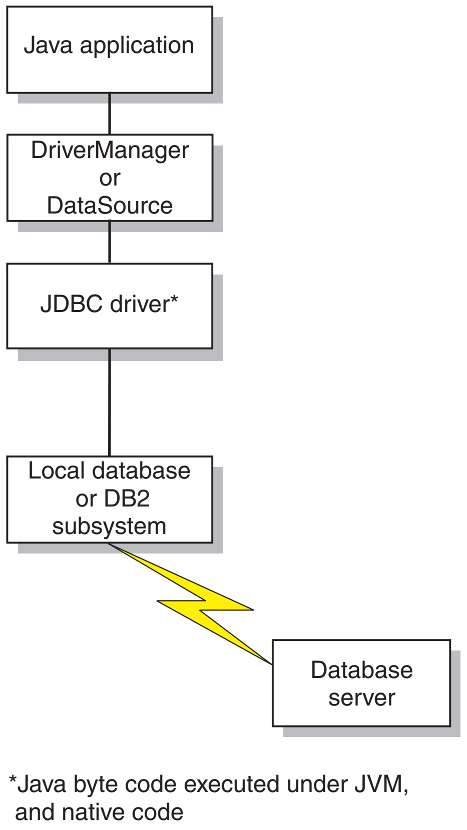

Java application
DriverManager or DataSource
JDBC driver*
Local database or DB2 subsystem
Database server
*Java byte code executed under JVM, and native code

Figure 2. Java application flow for a type 2 driver or IBM DB2 Driver for JDBC and SQLJ type 2 connectivity

Figure 3 shows how a Java application connects to a data source for IBM DB2 Driver for JDBC and SQLJ type 4 connectivity.

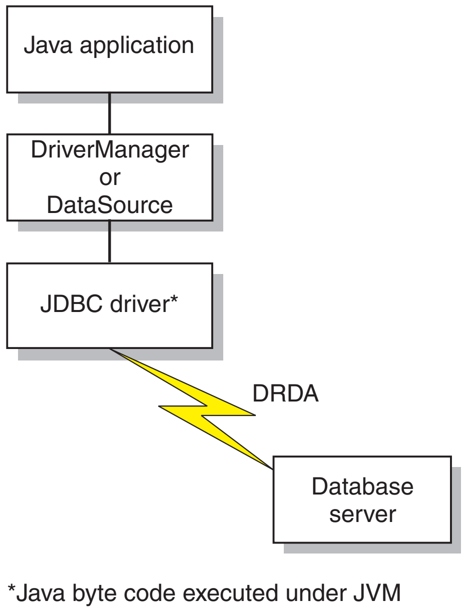

Java application
DriverManager or DataSource
JDBC driver*
DRDA
Database server
*Java byte code executed under JVM

Figure 3. Java application flow for IBM DB2 Driver for JDBC and SQLJ type 4 connectivity

Related concepts:

- "How DB2 applications connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver" on page 25

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 30
- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27

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 database server 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:

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

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

```
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`:

```
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:

```
►►──jdbc:db2:database──────────────────────────────────────────────►◄
```

The parts of the URL have the following meanings:

jdbc:db2:
: jdbc:db2: indicates that the connection is to a DB2 database server.

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*.
- 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:

```
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:

```
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
```

Related concepts:
- "Security under the DB2 JDBC Type 2 Driver" on page 141

Connecting to a data source using the DriverManager interface with the IBM DB2 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 Java application first loads the JDBC driver by invoking the `Class.forName` method. After the application loads the driver, it connects to a database server by invoking the `DriverManager.getConnection` method.

For the IBM DB2 Driver for JDBC and SQLJ, you load the driver by invoking the `Class.forName` method with the following argument:

```
com.ibm.db2.jcc.DB2Driver
```

For compatibility with previous JDBC drivers, you can use the following argument instead:

```
COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
```

The following code demonstrates loading the IBM DB2 Driver for JDBC and SQLJ:

```
try {
  // Load the IBM DB2 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.

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`:

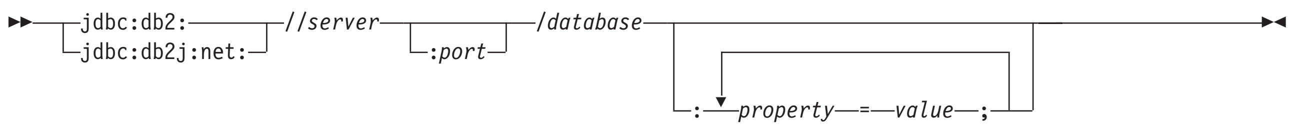
```
getConnection(String url);
getConnection(String url, user, password);
getConnection(String url, java.util.Properties info);
```

For IBM DB2 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.

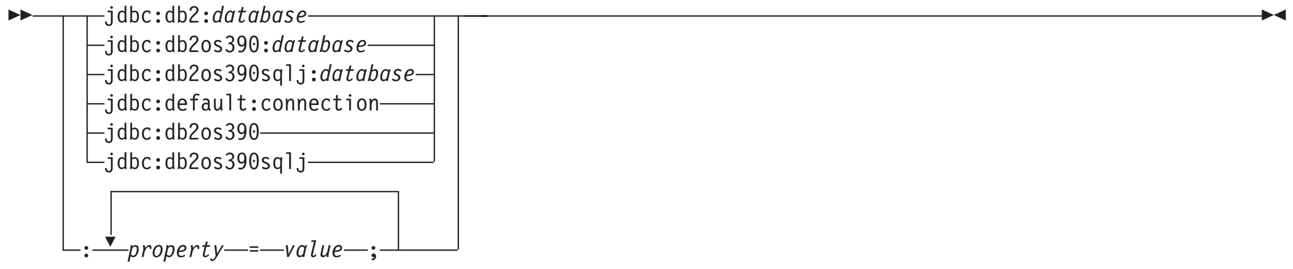
For IBM DB2 Driver for JDBC and SQLJ type 4 connectivity, specify a URL of the following form:

Syntax for a URL for IBM DB2 Driver for JDBC and SQLJ type 4 connectivity:



For IBM DB2 Driver for JDBC and SQLJ type 2 connectivity, specify a URL of one of the following forms:

Syntax for a URL for IBM DB2 Driver for JDBC and SQLJ type 2 connectivity:



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 or DB2 Database for Linux, UNIX, and Windows server.

jdbc:db2j:net:
Indicates that the connection is to a remote IBM Cloudscape server.

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. This name depends on whether IBM DB2 Driver for JDBC and SQLJ type 4 connectivity or IBM DB2 Driver for JDBC and SQLJ type 2 connectivity is used.

For IBM DB2 Driver for JDBC and SQLJ type 4 connectivity:

- 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 DB2 Driver for JDBC and SQLJ does not convert lowercase characters in the database value to uppercase for IBM DB2 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, 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 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"
  ```

For IBM DB2 Driver for JDBC and SQLJ type 2 connectivity:

- *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, *database* is a database alias.
- If the connection is to a DB2 for z/OS server or a DB2 UDB for iSeries server, all characters in *database* must be uppercase characters.

property=value;
  A property for the JDBC connection. For the definitions of these properties, see Properties for the IBM DB2 Driver for JDBC and SQLJ.

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 DB2 Driver for JDBC and SQLJ for the properties that you can specify.

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 *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: Setting the user ID and password in a URL:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose:" +
  "user=db2adm;password=db2adm;";
                                           // Set URL for data source
Connection con = DriverManager.getConnection(url);
                                           // Create connection
```

Example: Setting the user ID and password in user and password parameters:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose";
                                           // 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:

```
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://sysmvs1.stl.ibm.com:5021/san_jose";
                                           // Set URL for data source
Connection con = DriverManager.getConnection(url, properties);
                                           // Create connection
```

Related concepts:

- "Security under the IBM DB2 Driver for JDBC and SQLJ" on page 142

Related reference:
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

Connecting to a data source using 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. If your applications need to be portable among data sources, you should use the `DataSource` interface.

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. Figure 4 shows an example of creating and using a `DataSource` object in the same application.

Figure 4. Creating and using a DataSource object in the same application

```
import java.sql.*;        // JDBC base
import javax.sql.*;       // Methods for producing server-side
                          // applications using Java
import com.ibm.db2.jcc.*; // IBM DB2 Driver for JDBC and SQLJ  1
                          // interfaces
DB2SimpleDataSource db2ds=new DB2SimpleDataSource();     2
db2ds.setDatabaseName("db2loc1");                        3
                          // Assign the location name
db2ds.setDescription("Our Sample Database");
                          // Description for documentation
db2ds.setUser("john");
                          // Assign the user ID
db2ds.setPassword("db2");
                          // Assign the password
Connection con=db2ds.getConnection();                    4
                          // Create a Connection object
```

Note	Description
1	Import the package that contains the implementation of the DataSource interface.
2	Creates a `DB2SimpleDataSource` object. `DB2SimpleDataSource` is one of the DB2 implementations of the `DataSource` interface. See Create and deploy DataSource objects for information on DB2's `DataSource` implementations.
3	The `setDatabaseName`, `setDescription`, `setUser`, and `setPassword` methods assign attributes to the DB2SimpleDataSource object. See Properties for the IBM DB2 Driver for JDBC and SQLJ for information about the attributes that you can set for a `DB2SimpleDataSource` object under the IBM DB2 Driver for JDBC and SQLJ.
4	Establishes a connection to the data source that `DB2SimpleDataSource` object db2ds represents.

The best way to use a `DataSource` object is for your system administrator to create and manage it separately, using WebSphere 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 Create and deploy 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.

The remainder of this topic explains how to create a connection using a `DataSource` object, given that the system administrator has already created the object and assigned a logical name to it.

To obtain a connection using a `DataSource` object, you need 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:

   ```
   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.

Figure 5 shows an example of the code that you need in your application program to obtain a connection using a `DataSource` object, given that 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.

Figure 5. Obtaining a connection using a DataSource object

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
Context ctx=new InitialContext();                      2
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb"); 3
Connection con=ds.getConnection();                     4
```

Related tasks:

- "Creating and deploying DataSource objects" on page 33

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

How to determine which type of IBM DB2 Driver for JDBC and SQLJ connectivity to use

The IBM DB2 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 6. Comparison of IBM DB2 Driver for JDBC and SQLJ type 2 connectivity and IBM DB2 Driver for JDBC and SQLJ type 4 connectivity

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

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

Use IBM DB2 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 procedure. 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 DB2 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 DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing support.

Related tasks:

- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

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 Execute SQL in a JDBC application.
- Gather information about the data source to which you are connected. This process is discussed in Use DatabaseMetaData to learn about a data source.
- 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 Close a connection to a JDBC data source.

Related concepts:

- "JDBC interfaces for executing SQL" on page 38

Related tasks:

- "Disconnecting from database servers in JDBC applications" on page 88
- "Committing or rolling back JDBC transactions" on page 76
- "Learning about a data source using DatabaseMetaData methods" on page 35

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 DB2 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 DB2 Driver for JDBC and SQLJ type 2 connectivity or IBM DB2 Driver for JDBC and SQLJ type 4 connectivity.

- `com.ibm.db2.jcc.DB2ConnectionPoolDataSource`, which supports connection pooling. You can use this implementation with IBM DB2 Driver for JDBC and SQLJ type 2 connectivity or IBM DB2 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 DB2 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 example in Figure 6 shows how to perform these tasks.

```
import java.sql.*;        // JDBC base
import javax.naming.*;    // JNDI Naming Services
import javax.sql.*;       // Methods for producing server-side
                          // applications using Java
import com.ibm.db2.jcc.*; // DB2 implementation of JDBC
                          // standard extension APIs

DB2SimpleDataSource db2ds = new com.ibm.db2.jcc.DB2SimpleDataSource();  1

db2ds.setDatabaseName("db2loc1");                                       2
db2ds.setDescription("Our Sample Database");
db2ds.setUser("john");
db2ds.setPassword("db2");
⋮
Context ctx=new InitialContext();                                       3
Ctx.bind("jdbc/sampledb",db2ds);                                        4
```

Figure 6. Example of creating and deploying a DataSource object

Note	Description
1	Creates an instance of the DB2SimpleDataSource class.
2	This statement and the next three statements set values for properties of this `DB2SimpleDataSource` object.
3	Creates a context for use by JNDI.
4	Associates `DBSimple2DataSource` object db2ds with the logical name jdbc/sampledb. An application that uses this object can refer to it by the name jdbc/sampledb.

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

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 DB2-specific implementation of JDBC for the IBM DB2 Driver for JDBC and SQLJ.

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

Related concepts:
- "Basic steps in writing a JDBC application" on page 21

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 these types of applications, 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 data source, 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

If your application connects to a DB2 for z/OS server, a number of stored procedures need to be installed on that server before you can invoke some `DatabaseMetaData` methods that require DB2 catalog information. The stored procedures are:

- SQLCOLPRIVILEGES
- SQLCOLUMNS
- SQLFOREIGNKEYS
- SQLGETTYPEINFO
- SQLPRIMARYKEYS
- SQLPROCEDURECOLS
- SQLPROCEDURES
- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS

For DB2 UDB for OS/390 and z/OS, Version 7, the stored procedures are shipped in a PTFs. The PTF is orderable through normal service channels using the following PTF number:

Table 7. PTFs for DB2 for z/OS

DB2 for z/OS Version	PTF number
Version 7	UQ72083

Ask your DB2 for z/OS system administrator whether these stored procedures are installed.

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.

For 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  1
mtadtaint = dmtadta.getDriverVersion();                                 2
                                  // Check the driver version
System.out.println("Driver version: " + mtadtaint);
rs = dbmtadta.getProcedures(null, null, "%");
                                  // Get information for all procedures
while (rs.next()) {               // Position the cursor                 3a
 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                 3b
```

Figure 7. Using DatabaseMetaData methods to get information about a data source

Related reference:

- “Driver support for JDBC APIs” on page 247
- “JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers” on page 335

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 DB2 tables. Identifier `empNo` in the sample program in Basic steps in writing a JDBC application is an example of a Java `String` identifier that holds data that you retrieve from a CHAR column of a DB2 table.

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. Java, JDBC, and SQL data types shows the recommended mappings of Java data types and JDBC data types to SQL data types.

Related concepts:

- “Basic steps in writing a JDBC application” on page 21

Related reference:

- “Data types that map to SQL data types in JDBC applications” on page 227

Executing SQL statements in JDBC applications

The topics that follow contain information about executing SQL statements JDBC applications.
- "JDBC interfaces for executing SQL"
- "Updating DB2 tables in JDBC applications" on page 39
- "Retrieving data from DB2 tables in JDBC applications" on page 44
- "Calling stored procedures in JDBC applications" on page 52
- "Working with LOBs in JDBC applications" on page 57
- "ROWIDs in JDBC with the IBM DB2 Driver for JDBC and SQLJ" on page 61
- "Distinct types in JDBC applications" on page 61
- "Savepoints in JDBC applications" on page 62
- "Retrieving identity column values in JDBC applications" on page 63
- "Providing extended client information to the DB2 server with the IBM DB2 Driver for JDBC and SQLJ" on page 66

JDBC interfaces for executing SQL

You execute SQL statements in a traditional SQL program to insert, update, delete, or merge data in tables, retrieve data from the tables, or call stored procedures. To perform the same functions in a JDBC program, you invoke methods that 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 DB2 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.
  - 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 DB2 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.

For a complete list of DB2 support for JDBC interfaces, see Comparison of driver support for JDBC APIs.

Related tasks:
- "Retrieving data from DB2 using the PreparedStatement.executeQuery method" on page 47
- "Updating data in DB2 tables using the PreparedStatement.executeUpdate method" on page 40
- "Retrieving data from DB2 tables using the Statement.executeQuery method" on page 46

- “Creating and modifying DB2 objects using the Statement.executeUpdate method” on page 39

Related reference:

- “Driver support for JDBC APIs” on page 247

Updating DB2 tables in JDBC applications

The topics that follow contain information about creating and modifying DB2 tables in JDBC applications.

- “Creating and modifying DB2 objects using the Statement.executeUpdate method”
- “Updating data in DB2 tables using the PreparedStatement.executeUpdate method” on page 40
- “Learning about parameters in a PreparedStatement using ParameterMetaData methods” on page 41
- “Making batch updates in JDBC applications” on page 42

Creating and modifying DB2 objects using the Statement.executeUpdate method

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 DB2 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.

For example, 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.

```
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

Related reference:

- “Driver support for JDBC APIs” on page 247
- “JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers” on page 335

Updating data in DB2 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 DB2 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.

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 variables.
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.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
                                 // Create a PreparedStatement object       1
pstmt.setString(1,"4657");       // Assign first value to first parameter   2
pstmt.setString(2,"000010");     // Assign first value to second parameter
numUpd = pstmt.executeUpdate();  // Perform first update                    3
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      4
```

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. The following example demonstrates these steps.

```
Connection con;
PreparedStatement pstmt;
int numUpd;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
                                  // Create a PreparedStatement object   1
numUpd = pstmt.executeUpdate();   // Perform the update                  3
pstmt.close();                    // Close the PreparedStatement object  4
```

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

Related reference:
- "Driver support for JDBC APIs" on page 247
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

Learning about parameters in a PreparedStatement using ParameterMetaData methods

The IBM DB2 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.

For example, 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.

```
Connection con;
ParameterMetaData pmtadta;
int mtadtacnt;
String sqlType;
...
pstmt = con.prepareStatement(
  "UPDATE EMPLOYEE SET PHONENO=? WHERE EMPNO=?");
                                   // Create a PreparedStatement object   1
pmtadta = pstmt.getParameterMetaData();                                   2
                                   // Create a ParameterMetaData object
mtadtacnt = pmtadta.getParameterCount();                                  3
                                   // Determine the number of parameters
System.out.println("Number of statement parameters: " + mtadtacnt);
for (int i = 1; i <= mtadtacnt; i++) {
  sqlType = pmtadta.getParameterTypeName(i);                              4
                                   // 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

Related reference:
- "Driver support for JDBC APIs" on page 247

Making batch updates in JDBC applications

With batch updates, instead of updating rows of a DB2 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`

To make batch updates using several statements with no input parameters, follow these basic steps:
1. Invoke the `createStatement` method to create a `Statement` object.
2. For each SQL statement that you want to execute in the batch, invoke the `addBatch` method.
3. Invoke the `executeBatch` method to execute the batch of statements.
4. 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.

      If `AutoCommit` is enabled for the `Connection` object, the IBM DB2 Driver for JDBC and SQLJ adds a `commit` method at the end of the batch.

To make batch updates 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 `setXXX` methods to assign values to the input parameters.
   b. Invoke the `addBatch` method to add the set of input parameters to the batch.
3. Invoke the `executeBatch` method to execute the statements with all sets of parameters.
4. Check for errors. 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.
   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 DB2 Driver for JDBC and SQLJ adds a `commit` method at the end of the batch.

Example of a batch update: 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.

```
try {
...
  PreparedStatement prepStmt = con.prepareStatement(
    "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");             1
  prepStmt.setString(1,mgrnum1);                           2a
  prepStmt.setString(2,deptnum1);
  prepStmt.addBatch();                                     2b

  prepStmt.setString(1,mgrnum2);
  prepStmt.setString(2,deptnum2);
  prepStmt.addBatch();
  int [] numUpdates=prepStmt.executeBatch();               3
  for (int i=0; i < numUpdates.length; i++) {              4a
    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");
  }
  con.commit();                                            4b
} catch(BatchUpdateException b) {
  // process BatchUpdateException
}
```

Figure 12. Performing a batch update

Related tasks:
- "Committing or rolling back JDBC transactions" on page 76

Related reference:
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

Retrieving data from DB2 tables in JDBC applications

The topics that follow contain information about retrieving data from DB2 tables in JDBC applications.
- "Characteristics of a JDBC ResultSet under the IBM DB2 Driver for JDBC and SQLJ"
- "Learning about a ResultSet using ResultSetMetaData methods" on page 45
- "Retrieving data from DB2 tables using the Statement.executeQuery method" on page 46
- "Retrieving data from DB2 using the PreparedStatement.executeQuery method" on page 47
- "Making batch queries in JDBC applications" on page 48
- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 49

Characteristics of a JDBC ResultSet under the IBM DB2 Driver for JDBC and SQLJ

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 or delete rows of a `ResultSet`
- Leave the `ResultSet` open after a COMMIT

The following terms describe characteristics of a `ResultSet`:

scrollability
Whether the cursor can move forward, backward, or to a specific row.

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.

A scrollable `ResultSet` in JDBC is equivalent to the result table of a DB2 cursor that is declared as SCROLL. A scrollable cursor can be *insensitive*, *sensitive*, or *asensitive*. Insensitive means that changes to the underlying table after the cursor is opened are not visible to the cursor. Asensitive means that a cursor can behave as a sensitive or insensitive cursor, depending on whether it is used as a read-only cursor. Insensitive cursors are read-only. Sensitive means the following things:
- Changes that the cursor makes to the underlying table are always visible to the cursor.
- Changes that are made by other means to the underlying table *can* be visible to the cursor. In DB2, if the rows are fetched with FETCH INSENSITIVE, changes that are made by other means are not visible to the cursor. If the rows are fetched with FETCH SENSITIVE, changes that are made by other means are

visible to the cursor. In JDBC, calling the refreshRow method before calling getXXX methods has the same effect as FETCH SENSITIVE.

A JDBC ResultSet can also be *static* or *dynamic*, if the database server supports both attributes. You determine whether scrollable cursors in a program are static or dynamic by setting the cursorSensitivity property. See Properties for the IBM DB2 Driver for JDBC and SQLJ for more information about the cursorSensitivity property.

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 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 can test whether the current row is a delete hole by using the rowDeleted method. See Comparison of driver support for JDBC APIs for a complete list of the methods that are supported for ResultSets.

Related tasks:

- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 49

Learning about a ResultSet using ResultSetMetaData methods

Previous discussions of retrieving data from a table or stored procedure result set assumed that you know the number of columns and data types of the columns in the table or result set. This is not always the case, 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 for the same table definition might differ, depending on the data source. However, the returned information correctly reflects the column name information that is stored in the DB2 catalog for that data source.

For example, 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.

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
ResultSetMetaData rsmtadta;
int colCount
int mtadtaint;
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  1
colCount = rsmtadta.getColumnCount();                                   2
                                  // Find number of columns in EMP
for (i=1; i<= colCount; i++) {                                          3
 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 13. Using ResultSetMetaData methods to get information about a ResultSet

Related tasks:

- "Calling stored procedures using CallableStatement methods" on page 53
- "Retrieving data from DB2 tables using the Statement.executeQuery method" on page 46

Retrieving data from DB2 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 DB2 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. See Retrieve multiple result sets from a stored procedure in a JDBC application for more information.

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 DB2 Driver for JDBC and SQLJ also supports updatable and scrollable `ResultSets`. These are discussed in Specify updatability, scrollability, and holdability for ResultSets in JDBC applications.

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. See Comparison of driver support for JDBC APIs for a list of supported getXXX and setXXX methods.
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.

For example, 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.

```
String empNo;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();     // Create a Statement object          1
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE");                    2
                                  // Get the result table from the query
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 14. Using Statement.executeQuery

Related tasks:
- "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 54
- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 49

Related reference:
- "Driver support for JDBC APIs" on page 247

Retrieving data from DB2 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 DB2 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 Statement.execute method. See Retrieve multiple result sets from a stored procedure in a JDBC application for more information.

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.

For example, 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.

```
String empnum, phonenum;
Connection con;
PreparedStatement pstmt;
ResultSet rs;
...
pstmt = con.prepareStatement(
  "SELECT EMPNO, PHONENO FROM EMPLOYEE WHERE EMPNO=?");
                                  // Create a PreparedStatement object    1
pstmt.setString(1,"000010");      // Assign value to input parameter      2

rs = pstmt.executeQuery();        // Get the result table from the query  3
while (rs.next()) {               // Position the cursor                  4
 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                  5
pstmt.close();                    // Close the PreparedStatement          6
```

Figure 15. Using PreparedStatement.executeQuery

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`.

Related tasks:

- "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 54

Related reference:

- "Driver support for JDBC APIs" on page 247

Making batch queries in JDBC applications

The IBM DB2 Driver for JDBC and SQLJ provides a DB2-only interface that lets you perform batch queries on a homogeneous batch.

With the `DB2PreparedStatement` interface, you can execute a single SQL statement with multiple sets of input parameters.

Use the following PreparedStatement 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

Use the following DB2PreparedStatement method for executing the batch query.

- executeDB2QueryBatch

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.
4. Invoke the DB2PreparedStatement.executeBatch method to execute the statement with all sets of parameters.
5. Check for errors.

Example of a batch query: 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.

```
try {
...
  PreparedStatement prepStmt = con.prepareStatement(
    "SELECT EMPNO FROM EMPLOYEE WHERE EMPNO=?");          1
  prepStmt.setString(1,empnum1);                          2a
  prepStmt.addBatch();                                    2b

  prepStmt.setString(1,empnum2);
  prepStmt.addBatch();
  ((com.ibm.db2.jcc.DB2PreparedStatement)prepStmt).executeDB2QueryBatch();
                                                          3,4
} catch(BatchUpdateException b) {                         5
  // process BatchUpdateException
}
```

Figure 16. Performing a batch query

Related tasks:

- "Making batch updates in JDBC applications" on page 42

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

To specify scrollability, updatability, and holdability for a ResultSet, you need to 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:

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

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

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

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

```
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);
```

See Table 8 for a list of valid values for *resultSetType* and *resultSetConcurrency*.

Table 8. Valid combinations of resultSetType and resultSetConcurrency for scrollable ResultSets

resultSetType **value**	*resultSetConcurrency* **value**
TYPE_FORWARD_ONLY	CONCUR_READ_ONLY
TYPE_FORWARD_ONLY	CONCUR_UPDATABLE
TYPE_SCROLL_INSENSITIVE	CONCUR_READ_ONLY
TYPE_SCROLL_SENSITIVE	CONCUR_READ_ONLY
TYPE_SCROLL_SENSITIVE	CONCUR_UPDATABLE

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 statement that defines the `ResultSet` cannot select columns with the following data types:
- LONG VARCHAR
- LONG VARGRAPHIC
- DATALINK
- BLOB
- CLOB
- 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 Table 9.

Table 9. ResultSet methods for positioning a scrollable cursor

Method	Positions the cursor
`first()`	On the first row of the `ResultSet`
`last()`	On the last row of the `ResultSet`
`next()`[1]	On the next row of the `ResultSet`
`previous()`[2]	On the previous row of the `ResultSet`
`absolute(int n)`[3]	If $n>0$, on row n of the `ResultSet`. If $n<0$, and m is the number of rows in the `ResultSet`, on row $m+n+1$ of the `ResultSet`.
`relative(int n)`[4,5]	If $n>0$, on the row that is n rows after the current row. If $n<0$, on the row that is n rows before the current row. If $n=0$, on the current row.
`afterLast()`	After the last row in the `ResultSet`
`beforeFirst()`	Before the first row in the `ResultSet`

Notes:

1. If the cursor is before the first row of the `ResultSet`, this method positions the cursor on the first row.
2. If the cursor is after the last row of the `ResultSet`, 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 set, this method positions the cursor after the last row if n is positive, or before the first row if n is negative.
4. 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`.
5. 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`, `isBeforeFirst`, or `isAfterLast` method to obtain this information.

c. If you specified a *resultSetType* value of `TYPE_SCROLL_SENSITIVE` in step 1 on page 49, 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.

For example, 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.

```
String s;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                           ResultSet.CONCUR_UPDATABLE);            1
                                        // Create a Statement object
                                        // for a scrollable, updatable
                                        // ResultSet
rs = stmt.executeQuery("SELECT EMPNO FROM EMPLOYEE FOR UPDATE OF PHONENO");
                                       // Create the ResultSet          3
rs.afterLast();                        // Position the cursor at the end of
                                       // the ResultSet                 4a
while (rs.previous()) {                // Position the cursor backward
 s = rs.getString("EMPNO");            // Retrieve the employee number  4d
                                       // (column 1 in the result
                                       // table)
 System.out.println("Employee number = " + s);
                                       // Print the column value
 if (s.compareTo("000010") == 0) {     // Look for employee 000010
  rs.updateString("PHONENO","4657");   // Update their phone number
  rs.updateRow();                      // Update the row
 }
}
rs.close();                            // Close the ResultSet           5
stmt.close();                          // Close the Statement           6
```

Figure 17. Using a scrollable cursor

Important: The method of performing positioned UPDATE operations that is described previously in this topic follows the JDBC 2.0 standard. The IBM DB2 Driver for JDBC and SQLJ supports an alternative method that follows the JDBC 1.0 standard. That method is not recommended. The JDBC 1.0 method involves using the `ResultSet.getCursorName` method to obtain the name of the cursor for the `ResultSet`, and defining a positioned UPDATE statement of the following form:

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

If you use the JDBC 1.0 method to update data on a database server that supports multiple-row FETCH, the positioned UPDATE statement might update multiple rows, when you expect it to update a single row. To avoid unexpected updates, you should modify your applications to use the JDBC 2.0 method.

Calling stored procedures in JDBC applications

The topics that follow contain information about calling stored procedures in JDBC applications.

- “Calling stored procedures using CallableStatement methods” on page 53

- "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 54

Calling stored procedures using CallableStatement methods

To call stored procedures, you invoke methods in the `CallableStatement` class. The basic steps are:

1. Invoke the `Connection.prepareCall` method with the CALL statement as its argument to create a `CallableStatement` object.

   The CALL statement cannot contain literal arguments unless the DB2 server on which the statement runs supports dynamic execution of the CALL statement.
2. Invoke the `CallableStatement.setXXX` methods to pass values to the input (IN) parameters.

   If the database server 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 indicate which parameters are output-only (OUT) parameters, or input and output (INOUT) parameters.

   If the database server does not support dynamic execution of the CALL statement, you must specify the data types for CALL statement output-only or input and output 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.

   `CallableStatement.execute`
   Invoke this method if the stored procedure returns multiple result sets, or an unknown number of result sets.
5. If the stored procedure returns result sets, retrieve the result sets. See Retrieve multiple result sets from a stored procedure in a JDBC application.
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.

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.

```
int ifcaret;
int ifcareas;
int xsbytes;
String errbuff;
Connection con;
CallableStatement cstmt;
ResultSet rs;
...
cstmt = con.prepareCall("CALL DSN8.DSN8ED2(?,?,?,?,?)");                  1
                                 // Create a CallableStatement object
cstmt.setString (1, "DISPLAY THREAD(*)");                                 2
                                 // Set input parameter (DB2 command)
cstmt.registerOutParameter (2, Types.INTEGER);                            3
                                 // Register output parameters
cstmt.registerOutParameter (3, Types.INTEGER);
cstmt.registerOutParameter (4, Types.INTEGER);
cstmt.registerOutParameter (5, Types.VARCHAR);
cstmt.executeUpdate();           // Call the stored procedure             4
ifcaret = cstmt.getInt(2);       // Get the output parameter values       6
ifcareas = cstmt.getInt(3);
xsbytes = cstmt.getInt(4);
errbuff = cstmt.getString(5);
cstmt.close();                                                            7
```

Figure 18. Using CallableStatement methods for a stored procedure call with parameter markers

Related tasks:

- "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 54

Related reference:

- "Driver support for JDBC APIs" on page 247
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

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:

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

1. Invoke the `Statement.execute` method, the `PreparedStatement.execute` method, or the `CallableStatement.execute` method to call the stored procedure. Use `PreparedStatement.execute` 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.

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.

```
CallableStatement cstmt;
ResultSet rs;
int i;
String s;
...
cstmt.execute();                        // Call the stored procedure      1
rs = cstmt.getResultSet();              // Get the first result set       2
while (rs.next()) {                     // Position the cursor            3
 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 4a
                                        // and close the first result set
rs = cstmt.getResultSet();              // Get the second result set      4b
while (rs.next()) {                     // Position the cursor            4c
 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
cstmt.close();                          // Close the statement
```

Figure 19. Retrieving known result sets from a stored procedure

Retrieving an unknown 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. See Use ResultSetMetaData to learn about a ResultSet for more information on determining the contents of a ResultSet.

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. Process the ResultSet, as shown in Use ResultSetMetaData to learn about a ResultSet.
   c. Invoke the getMoreResults method to determine whether there is another result set. If getMoreResults returns true, go to step 2a to get the next result set.

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.

```
CallableStatement cstmt;
ResultSet rs;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
while (resultsAvailable) {                  // Test for result sets       1
 ResultSet rs = cstmt.getResultSet();       // Get a result set          2a
 ...                                    // process ResultSet
 resultsAvailable = cstmt.getMoreResults(); // Check for next result set 2c
                                            // (Also closes the
                                            // previous result set)
}
```

Figure 20. Retrieving unknown result sets from a stored procedure

Keeping result sets open:

In Figure 20, invocation of getMoreResults() closes the ResultSet object that is returned by the previous invocation of getResultSet. However, with the IBM DB2 Driver for JDBC and SQLJ, you can invoke the JDBC 3 form of getMoreResults, which has a parameter that determines whether the current ResultSet or previously-opened ResultSets are closed.

You can specify one of these constants:

Statement.KEEP_CURRENT_RESULT
Checks for the next ResultSet, but does not close the current ResultSet.

Statement.CLOSE_CURRENT_RESULT
Checks for the next ResultSet, and closes the current ResultSet.

Statement.CLOSE_ALL_RESULTS
Closes all ResultSets for the Statement object that were previously kept open.

For example, the code in Figure 21 keeps all ResultSets open until the final ResultSet has been retrieved, and then closes all ResultSets.

```
CallableStatement cstmt;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
if (resultsAvailable==true) {               // Test for result set
 ResultSet rs1 = cstmt.getResultSet();      // Get a result set
 ...                                          // Process ResultSet
 resultsAvailable = cstmt.getMoreResults(Statement.KEEP_CURRENT_RESULT);
                                            // Check for next result set
                                            // but do not close
                                            // previous result set
 if (resultsAvailable==true) {              // Test for another result set
  ResultSet rs2 = cstmt.getResultSet();     // Get next result set
  ...                                         // Process either ResultSet
 }
}
resultsAvailable = cstmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);
                                            // Close the result sets
```

Figure 21. Keeping retrieved stored procedure result sets open

Related tasks:

- "Learning about a ResultSet using ResultSetMetaData methods" on page 45

Working with LOBs in JDBC applications

The topics that follow contain information about updating and retrieving LOB data in JDBC applications.

- "LOBs in JDBC applications with the IBM DB2 Driver for JDBC and SQLJ"
- "Java data types for retrieving or updating LOB column data in JDBC applications" on page 59

LOBs in JDBC applications with the IBM DB2 Driver for JDBC and SQLJ

The IBM DB2 Driver for JDBC and SQLJ includes all of the LOB support in the JDBC 3.0 and earlier specifications. This driver also includes support for LOBs in additional methods and for additional data types.

Progressive streaming support: If the database server supports progressive streaming, the IBM DB2 Driver for JDBC and SQLJ can use progressive streaming to retrieve data in LOB or XML columns. With progressive streaming, the database server dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects. To cause JDBC to use progressive streaming to retrieve data, you need to set the `progressiveStreaming` property to `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`, and to be connected to a database server that supports progressive streaming, DB2 Version 9.1 for z/OS or later. When you enable progressive streaming, 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.

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:

```
...
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 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(2); // Put the CLOB from 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 DB2 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 locator support: The IBM DB2 Driver for JDBC and SQLJ can use LOB locators to retrieve data in LOB columns. You should use LOB locators only if the database server does not support progressive streaming. 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 `DB2BaseDataSource.NO`. If you do not set `progressiveStreaming` to `DB2BaseDataSource.NO`, and the database server supports progressive streaming, the JDBC driver ignores the `fullyMaterializeLobData` value.

`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 database. You cannot use a single LOB locator to move data between two different databases. To move LOB data between two databases, you need to materialize the LOB data when you retrieve it from a table in the first database and then insert that data into the table in the second database.

Additional methods supported by the IBM DB2 Driver for JDBC and SQLJ: In addition to the methods in the JDBC specification, the IBM DB2 Driver for JDBC and SQLJ includes LOB support in the following methods:

- You can specify a BLOB column as an argument of the following `ResultSet` methods to retrieve data from a BLOB column:
  - `getBinaryStream`
  - `getBytes`
- You can specify a CLOB column as an argument of the following `ResultSet` methods to retrieve data from a CLOB column:
  - `getAsciiStream`
  - `getCharacterStream`
  - `getString`
  - `getUnicodeStream`
- You can use the following `PreparedStatement` methods to set the values for parameters that correspond to BLOB columns:
  - `setBytes`
  - `setBinaryStream`
- You can use the following `PreparedStatement` methods to set the values for parameters that correspond to CLOB columns:
  - `setString`
  - `setAsciiStream`
  - `setUnicodeStream`
  - `setCharacterStream`
- You can retrieve the value of a JDBC `CLOB` parameter using the following `CallableStatement` method:
  - `getString`

Restriction on using LOBs with the IBM DB2 Driver for JDBC and SQLJ: If you are using IBM DB2 Driver for JDBC and SQLJ type 2 connectivity, you cannot call a stored procedure that has DBCLOB OUT or INOUT parameters.

Related reference:

- "Driver support for JDBC APIs" on page 247
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

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

When the deferPrepares property is set to true, and the IBM DB2 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.

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, or input/output 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:
  ```
  cstmt.setBlob(parmIndex, blobData);
  ```
- Use a `CallableStatement.setObject` call that specifies that the target data type is BLOB:
  ```
  byte[] byteData = {(byte)0x1a, (byte)0x2b, (byte)0x3c};
  cstmt.setObject(parmInd, byteData, java.sql.Types.BLOB);
  ```
- Use an input parameter of type of `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.io.ByteArrayInputStream byteStream =
    new java.io.ByteArrayInputStream(byteData);
  int numBytes = byteData.length;
  cstmt.setBinaryStream(parmIndex, byteStream, numBytes);
  ```

Output parameters for BLOB columns:

For output parameters for BLOB columns, or input/output 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:
  ```
  cstmt.registerOutParameter(parmIndex, java.sql.Types.BLOB);
  cstmt.execute();
  byte[] byteData = cstmt.getBytes(parmIndex);
  ```

Input parameters for CLOB columns:

For input parameters for CLOB columns, or input/output 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:

```
cstmt.setClob(parmIndex, clobData);
```

- Use a `CallableStatement.setObject` call that specifies that the target data type is CLOB:

  ```
  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.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:

    ```
    byte[] charDataBytes = charData.getBytes("US-ASCII");
    java.io.ByteArrayInputStream byteStream =
      new java.io.ByteArrayInputStream (charDataBytes);
    cstmt.setAsciiStream(parmIndex, byteStream, charDataBytes.length);
    ```

  For these calls, you need to specify the exact length of the input data.
- Use a String input parameter with a `cstmt.setString` call:

  ```
  cstmt.setString(charData);
  ```

  If the length of the data is greater than 32KB, the JDBC driver assigns the CLOB data type to the input data.
- Use a String input parameter with a `cstmt.setObject` call, and specify the target data type as VARCHAR or LONGVARCHAR:

  ```
  cstmt.setObject(parmIndex, charData, java.sql.Types.VARCHAR);
  ```

  If the length of the data is greater than 32KB, the JDBC driver assigns the CLOB data type to the input data.

Output parameters for CLOB columns:

For output parameters for CLOB columns, or input/output parameters that are used for output from CLOB columns, you can use one of the following techniques:

- Use the `CallableStatement.registerOutParameter` call to specify that an output parameter is of type CLOB. Then you can retrieve the parameter value into any variable that has a data type that is compatible with a CLOB data type. For example, the following code lets you retrieve a CLOB value into a String variable:

  ```
  cstmt.registerOutParameter(parmIndex, java.sql.Types.CLOB);
  cstmt.execute();
  String charData = cstmt.getString(parmIndex);
  ```

- Use the `CallableStatement.registerOutParameter` call to specify that an output parameter is of type VARCHAR or LONGVARCHAR:

  ```
  cstmt.registerOutParameter(parmIndex, java.sql.Types.VARCHAR);
  cstmt.execute();
  String charData = cstmt.getString(parmIndex);
  ```

  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.

Related concepts:

- "LOBs in JDBC applications with the IBM DB2 Driver for JDBC and SQLJ" on page 57

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

ROWIDs in JDBC with the IBM DB2 Driver for JDBC and SQLJ

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

You can use the following `ResultSet` methods to retrieve data from a ROWID column:

- `getBytes`
- `getObject`

For `getObject`, the IBM DB2 Driver for JDBC and SQLJ returns an instance of the DB2-only class `com.ibm.db2.jcc.DB2RowID`.

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

- `setBytes`
- `setObject`

For `setObject`, use the DB2-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.

Example: Using PreparedStatement.setObject with a com.ibm.db2.jcc.DB2Types.ROWID target type: To set parameter 1, use this form of the `SetObject` method:

```
ps.setObject(1, bytes[], com.ibm.db2.jcc.DB2Types.ROWID);
```

Example: Using PreparedStatement.setObject with a com.ibm.db2.jcc.DB2RowID target type: Suppose that rwid is an instance of `com.ibm.db2.jcc.DB2RowID`. To set parameter 1, use this form of the `SetObject` method:

```
ps.setObject (1, rwid);
```

To call a stored procedure that is defined with a ROWID output parameter, register that parameter to be of the `com.ibm.db2.jcc.DB2Types.ROWID` type.

Example: Using CallableStatement.registerOutParameter with a com.ibm.db2.jcc.DB2Types.ROWID parameter type: To register parameter 1 of a CALL statement as a `com.ibm.db2.jcc.DB2Types.ROWID` data type, use this form of the `registerOutParameter` method:

```
cs.registerOutParameter(1, com.ibm.db2.jcc.DB2Types.ROWID)
```

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

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:

```
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 22. Creating and using a distinct type

Related reference:

- "CREATE DISTINCT TYPE statement" in *SQL Reference, Volume 2*

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. SQL statements exist to set a savepoint, release a savepoint, and restore data and schemas to the state that the savepoint represents.

The IBM DB2 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.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.

```
Connection con;
Statement stmt;
ResultSet rs;
String empNumVar;
int shoeSizeVar;
...
con.setAutoCommit(false);                  // set autocommit OFF
stmt = con.createStatement();              // Create a Statement object
...                                          // Perform some SQL
con.commit();                              // Commit the transaction
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
  "VALUES ('000010', 6)");                 // Insert a row
Savepoint savept = con.setSavepoint();     // Create a savepoint
...
stmt.executeUpdate("INSERT INTO EMP_SHOE " +
  "VALUES ('000020', 10)");                // Insert another row
conn.rollback(savept);                     // Roll back work to the point
                                           //  after the first insert
...
con.releaseSavepoint(savept);              // Release the savepoint
stmt.close();                              // Close the Statement
```

Figure 23. Setting, rolling back to, and releasing a savepoint in a JDBC application

Related tasks:
- "Committing or rolling back JDBC transactions" on page 76

Related reference:
- "Driver support for JDBC APIs" on page 247

Retrieving identity column values in JDBC applications

An identity column is a DB2 table column that provides a way for DB2 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).

With the IBM DB2 Driver for JDBC and SQLJ, you can retrieve identity columns from a DB2 table using JDBC 3.0 methods. In a JDBC program, identity columns are known as automatically generated keys. To enable retrieval of automatically generated keys from a table, you need to indicate when you insert rows that you will want to retrieve automatically generated key values. You do that by setting a flag in a `Connection.prepareStatement`, `Statement.executeUpdate`, or `Statement.execute` method call. The statement that is executed must be an INSERT statement or an INSERT within SELECT statement. Otherwise, the JDBC driver ignores the parameter that sets the flag.

To retrieve automatically generated keys from a DB2 table, 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:

  Use the following form for a table on any database server that supports identity columns. Use this form for single-row INSERT statements only.

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

  Use one of the following forms only for a table on any database server that supports identity columns and INSERT within SELECT. Use one of these forms for single-row INSERT statements or multiple-row INSERT statements. 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.

  ```
  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:

  Use the following form for a table on any database server that supports identity columns. Use this form for single-row INSERT statements only.

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

  Use one of the following forms only for a table on any database server that supports identity columns and INSERT within SELECT. Use one of these forms for single-row INSERT statements or multiple-row INSERT statements.

  ```
  Statement.executeUpdate(sql-statement, String [] columnNames);
  Statement.executeUpdate(sql-statement, int [] columnIndexes);
  ```
- If you use the `Statement.execute` method to insert rows, invoke one of these forms of the `Statement.execute` method:

  Use the following form for a table on any database server that supports identity columns. Use this form for single-row INSERT statements only.

  ```
  Statement.execute(sql-statement, Statement.RETURN_GENERATED_KEYS);
  ```

  Use one of the following forms only for a table on any database server that supports identity columns and INSERT within SELECT. Use one of these forms for single-row INSERT statements or multiple-row INSERT statements.

  ```
  Statement.execute(sql-statement, String [] columnNames);
  Statement.execute(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.

   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.

```
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) " +          1
  "VALUES ('000010', "5555")",        // Insert a row
  Statement.RETURN_GENERATED_KEYS);   // Indicate you want automatically
                                      // generated keys
rs = stmt.getGeneratedKeys();         // Retrieve the automatically      2
                                      // generated key value in a ResultSet.
                                      // Only one row is returned.
                                      // Create ResultSet for query
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
```

Figure 24. Retrieving automatically generated keys

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.

```
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 =                                                           1
  stmt.executeUpdate("INSERT INTO EMP_PHONE (EMPNO, PHONENO)" +
  "VALUES ('000010', '5555'), ('000020', '5556')", id_col);
                                       // Insert two rows
                                       // Indicate you want automatically
                                       // generated keys
rs = stmt.getGeneratedKeys();          // Retrieve the automatically        2
                                       // generated key values in a ResultSet.
                                       // Two rows are returned.
                                       // Create ResultSet for query
while (rs.next()) {
 java.math.BigDecimal idColVar = rs.getBigDecimal(1);
                                       // Get automatically generated key
                                       // values
 System.out.println("automatically generated key value = " + idColVar);
}
stmt.close();
con.close();
```

Figure 25. Retrieving automatically generated keys after a multiple-row INSERT

Related concepts:

- "Identity Columns" in *SQL Guide*

Related tasks:

- "Updating data in DB2 tables using the PreparedStatement.executeUpdate method" on page 40
- "Creating and modifying DB2 objects using the Statement.executeUpdate method" on page 39

Related reference:

- "Driver support for JDBC APIs" on page 247

Providing extended client information to the DB2 server with the IBM DB2 Driver for JDBC and SQLJ

The IBM DB2 Driver for JDBC and SQLJ provides DB2-only methods that you can use to provide extra information about the client to the server. This information can be used for accounting, workload management, or debugging. The information is sent to the DB2 server when the application performs an action that accesses the server, such as executing SQL.

The methods are listed in Table 10.

Table 10. Methods that provide client information to the DB2 server

Method	Information provided
setDB2ClientAccountingInformation	Accounting information
setDB2ClientApplicationInformation	Name of the application that is working with a connection
setDB2ClientDebugInfo	The CLIENT DEBUGINFO connection attribute for the Unified debugger
setDB2ClientProgramId	
setDB2ClientUser	User name for a connection
setDB2ClientWorkstation	Client workstation name for a connection

To set the extended information:
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 10.
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.

```
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);
      if (conn instanceof DB2Connection) {
        DB2Connection db2conn = (DB2Connection) conn;          2
        db2conn.setDB2ClientUser("Michael L Thompson");        3
        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();                     4
      }
    } catch (Throwable e) {
        e.printStackTrace();
      }
  }
}
```

Figure 26. Example of passing extended client information to a DB2 server

Related reference:
- "Summary of IBM DB2 Driver for JDBC and SQLJ extensions to JDBC" on page 301

Working with XML data in JDBC applications

The topics that follow contain information about updating and retrieving XML data in JDBC applications.

- "XML data in JDBC applications"
- "XML column updates in JDBC applications"
- "XML data retrieval in JDBC applications" on page 70
- "Invocation of routines with XML parameters in Java applications" on page 72
- "Java support for XML schema registration and removal" on page 74

XML data in JDBC applications

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.

In applications, XML data is in the serialized string format.

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 XML string 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.

Java has no XML data type, and invocations of metadata methods, such as `ResultSetMetaData.getColumnTypeName` return a type of java.sql.Types.OTHER for an XML column type.

Related concepts:

- "XML column updates in JDBC applications" on page 68
- "XML data retrieval in JDBC applications" on page 70

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

XML column updates in JDBC applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the serialized string format. Table 11 on page 69 lists the methods and corresponding input data types that you can use to put data in XML columns.

Table 11. Methods and data types for updating XML columns

Method	Input data type
PreparedStatement.setAsciiStream	InputStream
PreparedStatement.setBinaryStream	InputStream
PreparedStatement.setBlob	Blob
PreparedStatement.setBytes	byte[]
PreparedStatement.setCharacterStream	Reader
PreparedStatement.setClob	Clob
PreparedStatement.setObject	byte[], Blob, Clob, DB2Xml, InputStream, Reader, String
PreparedStatement.setString	String

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 database server 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 database server as character data, but the data contains encoding information. The database server handles incompatibilities between internal and external encoding as follows:

- If the database server is DB2 Database for Linux, UNIX, and Windows, the database server 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 server ignores the internal encoding.
- If the database server is DB2 for z/OS, the database server ignores the internal encoding.

Data in XML columns is stored in UTF-8 encoding. The database server handles conversion of the data from its internal or external encoding to UTF-8.

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.

```
public void insertBinStream()
{
  PreparedStatement insertStmt = null;
  String sqls = null;
  int cid = 0;
  ResultSet rs=null;
  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.");
    }
  }
  catch (IOException ioe) {
   ioe.printStackTrace();
  }
  catch (SQLException sqle) {
    System.out.println("insertBinStream: SQL Exception: " +
      sqle.getMessage());
    System.out.println("insertBinStream: SQL State: " +
      sqle.getSQLState());
    System.out.println("insertBinStream: SQL Error Code: " +
      sqle.getErrorCode());

  }
}
```

Related concepts:

- "Encoding considerations for XML data in JDBC, SQLJ, and .NET applications" in *XML Guide*

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

XML data retrieval in JDBC applications

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 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.

  `DB2Xml.getDB2XmlXXX` methods add XML declarations with encoding specifications to the output data. `DB2Xml.getDB2XXX` methods do not add XML declarations with encoding specifications to the output data.

Table 12 lists the `ResultSet` methods and corresponding output data types for retrieving XML data.

Table 12. ResultSet methods and data types for retrieving XML data

Method	Output data type
ResultSet.getAsciiStream	InputStream
ResultSet.getBinaryStream	InputStream
ResultSet.getBytes	byte[]
ResultSet.getCharacterStream	Reader
ResultSet.getObject	DB2Xml
ResultSet.getString	String

Table 13 on page 71 lists the methods and corresponding output data types for retrieving data from a `DB2Xml` object, as well as the type of encoding in the XML declaration that the driver adds to the output data.

Table 13. DB2Xml methods, data types, and added encoding specifications

Method	Output data type	Type of XML internal encoding declaration added
DB2Xml.getDB2AsciiStream	InputStream	None
DB2Xml.getDB2BinaryStream	InputStream	None
DB2Xml.getDB2Bytes	byte[]	None
DB2Xml.getDB2CharacterStream	Reader	None
DB2Xml.getDB2String	String	None
DB2Xml.getDB2XmlAsciiStream	InputStream	US-ASCII
DB2Xml.getDB2XmlBinaryStream	InputStream	Specified by `getDB2XmlBinaryStream` *targetEncoding* parameter
DB2Xml.getDB2XmlBytes	byte[]	Specified by DB2Xml.getDB2XmlBytes *targetEncoding* parameter
DB2Xml.getDB2XmlCharacterStream	Reader	ISO-10646-UCS-2
DB2Xml.getDB2XmlString	String	ISO-10646-UCS-2

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 a String variable.

```
public void fetchToString()
  {
    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 java.sql.Types.OTHER
      ResultSetMetaData meta = rs.getMetaData();
      String colType = meta.getColumnTypeName(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()
  {
    System.out.println(">> fetchToDB2Xml: Get XML data as a DB2XML object " +
    "using getObject");
    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 java.sql.Types.OTHER
      ResultSetMetaData meta = rs.getMetaData();
      String colType = meta.getColumnTypeName(1);
      System.out.println("fetchToObject: Column type = " + colType);
      if (rs.next() == false) {
        System.out.println("fetchToObject: "
          "Cannot read document with cid " + cid);
      }
      else {
      // Retrieve the XML data with getObject, and cast the object
      // as a DB2Xml object. Then write it to a string with
      // explicit internal ISO-10646-UCS-2 encoding.
        com.ibm.db2.jcc.DB2Xml xml =
          (com.ibm.db2.jcc.DB2Xml) rs.getObject(1);
        System.out.println (xml.getDB2XmlString());
      }
      rs.close();
    }
    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());
    }
  }
```

Related concepts:

- "Encoding considerations for XML data in JDBC, SQLJ, and .NET applications" in *XML Guide*

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

Invocation of routines with XML parameters in Java applications

SQL or external stored procedures and external user-defined functions can include XML parameters. For SQL procedures, those parameters in the stored procedure definition have the XML type. For external stored procedures and user-defined functions, 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 `com.ibm.db2.jcc.DB2Xml` type. To register XML output parameters, register the parameters as the `com.ibm.db2.jcc.DB2Types.XML` type.

Example: JDBC program that calls a stored procedure that takes three XML parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```
com.ibm.db2.jcc.DB2Xml in_xml = xmlvar;
com.ibm.db2.jcc.DB2Xml out_xml = null;
com.ibm.db2.jcc.DB2Xml inout_xml = xmlvar;
                                  // Declare an input, output, and
                                  // input/output XML parameter
Connection con;
CallableStatement cstmt;
ResultSet rs;
...
stmt = con.prepareCall("CALL SP_xml(?,?,?)");
                                  // Create a CallableStatement object
cstmt.setObject (1, in_xml);      // Set input parameter
cstmt.registerOutParameter (2, com.ibm.db2.jcc.DB2Types.XML);
                                  // Register out and input parameters
cstmt.registerOutParameter (3, com.ibm.db2.jcc.DB2Types.XML);
cstmt.executeUpdate();            // Call the stored procedure
System.out.println("Parameter values from SP_xml call: ");
System.out.println("Output parameter value ");
printBytes(out_xml.getDB2String());
                               // Use the DB2-only method getBytes to
                               // convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_xml.getDB2String());
```

To call a routine with XML parameters from an SQLJ program, use parameters of the `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.

```
com.ibm.db2.jcc.DB2Xml in_xml = xmlvar;
com.ibm.db2.jcc.DB2Xml out_xml = null;
com.ibm.db2.jcc.DB2Xml inout_xml = xmlvar;
                                  // Declare an input, output, and
                                  // input/output 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 ");
printBytes(out_xml.getDB2String());
                               // Use the DB2-only method getBytes to
                               // convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_xml.getDB2String());
```

Related tasks:

- "Calling stored procedures in an SQLJ application" on page 122
- "Calling stored procedures using CallableStatement methods" on page 53

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227
- "DB2Xml interface" on page 333

Java support for XML schema registration and removal

DB2 provides the SYSPROC.XSR_REGISTER, SYSPROC.XSR_ADDSCHEMADOC, SYSPROC.XSR_COMPLETE, and SYSPROC.XSR_REMOVE stored procedures that let you register and remove XML schemas and their components.

The IBM DB2 Driver for JDBC and SQLJ provides methods that let you perform the same functions from a Java application program. Those 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 an `InputStream` objects, and one form for XML schema documents that are in a `String`s.

DB2Connection.deregisterDB2XMLObject
Removes an XML schema definition from DB2.

Before you can invoke these methods, the underlying stored procedures 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. The *xmlSchemaLocations* value is null, so DB2 will not find this XML schema on an invocation of DSN_XMLVALIDATE that supplies a non-null XML schema location value. No additional properties are registered.

```
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,
    false);
}
```

Figure 27. Example of registration of an XML schema with DB2 using an XML document from an input stream

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.

```
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);
}
```

Figure 28. Example of removal of an XML schema from DB2

Related concepts:

- "XML schema, DTD, and external entity management using the XML schema repository (XSR)" in *XML Guide*
- "XSR object registration" in *XML Guide*

Transaction control in JDBC applications

The topics that follow discuss JDBC methods for controlling DB2 transactions.

- "Setting the isolation level for a JDBC transaction"
- "Committing or rolling back JDBC transactions"

Setting the isolation level for a JDBC transaction

To set the isolation level for a unit of work within a JDBC program, use the `Connection.setTransactionIsolation(int level)` method. Table 14 shows the values of *level* that you can specify in the `Connection.setTransactionIsolation` method and their DB2 equivalents.

Table 14. Equivalent JDBC and DB2 isolation levels

JDBC value	DB2 isolation level
TRANSACTION_SERIALIZABLE	Repeatable read
TRANSACTION_REPEATABLE_READ	Read stability
TRANSACTION_READ_COMMITTED	Cursor stability
TRANSACTION_READ_UNCOMMITTED	Uncommitted read

Related concepts:

- "JDBC connection objects" on page 33

Committing or rolling back JDBC transactions

In JDBC, to commit or roll back transactions explicitly, use the `commit` or `rollback` methods. For example:

```
 Connection con;
 ...
con.commit();
```

If autocommit mode is on, the DB2 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.

When autocommit mode is on, you cannot execute the `commit` and `rollback` methods.

Connections that participate in distributed transactions cannot invoke the `setAutoCommit(true)` method.

When you change the autocommit state, the DB2 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.

Related concepts:
- "Savepoints in JDBC applications" on page 62

Related tasks:
- "Disconnecting from database servers in JDBC applications" on page 88
- "Making batch updates in JDBC applications" on page 42

Handling errors and warnings in JDBC applications

The topics that follow explain how to handle SQL errors and warnings in JDBC applications.
- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ"
- "Handling an SQLWarning under the IBM DB2 Driver for JDBC and SQLJ" on page 81
- "Retrieving information from a BatchUpdateException" on page 82
- "Handling an SQLException under the DB2 JDBC Type 2 Driver" on page 84
- "Handling an SQLWarning under the DB2 JDBC Type 2 Driver" on page 85

Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ

As in all Java programs, error handling is done using `try/catch` blocks. Methods throw exceptions when an error occurs, and the code in the `catch` block handles those exceptions.

JDBC provides 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 IBM DB2 Driver for JDBC and SQLJ provides a `com.ibm.db2.jcc.DB2Diagnosable` interface that extends the `SQLException` class. The `DB2Diagnosable` interface gives you more information about errors that occur when DB2 is accessed. If the JDBC driver detects an error, `DB2Diagnosable` gives you the same information as the standard SQLException class. However, if DB2 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.

The basic steps for handling an SQLException in a JDBC program that runs under the IBM DB2 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:
```
import com.ibm.db2.jcc.DB2Diagnosable;
import com.ibm.db2.jcc.DB2Sqlca;
```
2. Put code that can generate an SQLException in a try block.
3. 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. Check whether any DB2-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 database server.

c. Invoke the `SQLException.getNextException` method to retrieve the next `SQLException`.

The following code demonstrates how to obtain information from the DB2 version of an `SQLException` that is provided with the IBM DB2 Driver for JDBC and SQLJ. The numbers to the right of selected statements correspond to the previously-described steps.

```
import java.sql.*;                       // Import JDBC API package
import com.ibm.db2.jcc.DB2Diagnosable;  // Import packages for DB2      1
import com.ibm.db2.jcc.DB2Sqlca;        // SQLException support
java.io.PrintWriter printWriter;        // For dumping all SQLException
                                        // information
...
try {                                                                  2
  // Code that could generate SQLExceptions
 ...
} catch(SQLException sqle) {
    while(sqle != null) {             // Check whether there are more  3a
                                      // SQLExceptions to process
      //=====> Optional DB2-only error processing
      if (sqle instanceof DB2Diagnosable) {                            3b
                                      // Check if DB2-only information exists
          com.ibm.db2.jcc.DB2Diagnosable diagnosable =
            (com.ibm.db2.jcc.DB2Diagnosable)sqle;                      3b1
          diagnosable.printTrace (printWriter, "");                    3b2
          java.lang.Throwable throwable =
            diagnosable.getThrowable();                                3b3
          if (throwable != null) {
            // Extract java.lang.Throwable information
            // such as message or stack trace.
            ...
          }
          DB2Sqlca sqlca = diagnosable.getSqlca();                     3b4
                                            // Get DB2Sqlca object
          if (sqlca != null) {              // Check that DB2Sqlca is not null
            int sqlCode = sqlca.getSqlCode(); // Get the SQL error code  3b5
            String sqlErrmc = sqlca.getSqlErrmc();                     3b6
                                            // Get the entire SQLERRMC
            String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
                                            // You can also retrieve the
                                            // individual SQLERRMC tokens
            String sqlErrp = sqlca.getSqlErrp();                       3b7
                                            // Get the SQLERRP
            int[] sqlErrd = sqlca.getSqlErrd();                        3b8
                                            // Get SQLERRD fields
            char[] sqlWarn = sqlca.getSqlWarn();                       3b9
                                            // Get SQLWARN fields
            String sqlState = sqlca.getSqlState();                     3b10
                                            // Get SQLSTATE
            String errMessage = sqlca.getMessage();                    3b11
                                            // Get error message

            System.err.println ("Server error message: " + errMessage);

            System.err.println ("--------------- SQLCA ---------------");
            System.err.println ("Error code: " + sqlCode);
            System.err.println ("SQLERRMC: " + sqlErrmc);
            for (int i=0; i< sqlErrmcTokens.length; i++) {
              System.err.println ("  token " + i + ": " + sqlErrmcTokens[i]);
            }
```

Figure 29. Processing an SQLException under the IBM DB2 Driver for JDBC and SQLJ (Part 1 of 2)

```
              System.err.println ( "SQLERRP: " + sqlErrp );
              System.err.println (
                "SQLERRD(1): " + sqlErrd[0] + "\n" +
                "SQLERRD(2): " + sqlErrd[1] + "\n" +
                "SQLERRD(3): " + sqlErrd[2] + "\n" +
                "SQLERRD(4): " + sqlErrd[3] + "\n" +
                "SQLERRD(5): " + sqlErrd[4] + "\n" +
                "SQLERRD(6): " + sqlErrd[5] );
              System.err.println (
                "SQLWARN1: " + sqlWarn[0] + "\n" +
                "SQLWARN2: " + sqlWarn[1] + "\n" +
                "SQLWARN3: " + sqlWarn[2] + "\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" +
                "SQLWARNA: " + sqlWarn[9] );
              System.err.println ("SQLSTATE: " + sqlState);
                                               // portion of SQLException
            }
        sqle=sqle.getNextException();     // Retrieve next SQLException      3c
      }
    }
```

Figure 29. Processing an SQLException under the IBM DB2 Driver for JDBC and SQLJ (Part 2 of 2)

Related reference:
- "Error codes issued by the IBM DB2 Driver for JDBC and SQLJ" on page 344

Handling an SQLWarning under the IBM DB2 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. Calling `getWarnings` retrieves an `SQLWarning` object.

Important:

When a call to `Statement.executeUpdate` or `PreparedStatement.executeUpdate` affects no rows, the IBM DB2 Driver for JDBC and SQLJ generates an `SQLWarning` with error code +100.

When a call to `ResultSet.next` returns no rows, the IBM DB2 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 DB2 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.

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. 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.

```
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          1
while (sqlwarn != null) {         // While there are warnings, get and   2a
                                  // 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           2f
}
```

Figure 30. Processing an SQLWarning

Related tasks:

- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77

Retrieving information from a BatchUpdateException

When an error occurs during execution of a statement in a batch, processing continues. However, executeBatch throws a BatchUpdateException. A BatchUpdateException object contains the following items:

- 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.

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.

   Statement.EXECUTE_FAILED
   This value is returned if the statement did not execute successfully.
2. Use SQLException methods getMessage, getSQLState, and getErrorCode to retrieve the description of the error, the SQLSTATE, and the error code for the first error.
3. Use the BatchUpdateException.getNextException method to get a chained SQLException.
4. In a loop, execute the getMessage, getSQLState, getErrorCode, and getNextException method calls to obtain information about an SQLException and get the next SQLException.

Example of obtaining information from a BatchUpdateException: The following code fragment demonstrates how to obtain the fields of a BatchUpdateException and the chained SQLException objects. The numbers to the right of selected statements correspond to the previously-described steps.

```
try {
  // Batch updates
} catch(BatchUpdateException buex) {
    System.err.println("Contents of BatchUpdateException:");
    System.err.println(" Update counts: ");
    int [] updateCounts = buex.getUpdateCounts();           1
    for (int i = 0; i < updateCounts.length; i++) {
      System.err.println("  Statement " + i + ":" + updateCounts[i]);
    }
    System.err.println(" Message: " + buex.getMessage());    2
    System.err.println(" SQLSTATE: " + buex.getSQLState());
    System.err.println(" Error code: " + buex.getErrorCode());
    SQLException ex = buex.getNextException();              3
    while (ex != null) {                                    4
      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();
    }
}
```

Figure 31. Retrieving a BatchUpdateException fields

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.

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.

Related tasks:

- "Making batch updates in JDBC applications" on page 42

Handling an SQLException under the DB2 JDBC Type 2 Driver

As in all Java programs, error handling is done using `try/catch` blocks. Methods throw exceptions when an error occurs, and the code in the `catch` block handles those exceptions.

JDBC provides 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.

```
import java.sql.*;                      // Import JDBC API package
...
try {
  // Code that could generate SQLExceptions
 ...
} catch(SQLException sqle) {
    while(sqle != null) {             // Check whether there are more    1
      System.out.println("Message:  " + sqle.getMessage());              2
      System.out.println("SQLSTATE: " + sqle.getSQLState());
      System.out.println("SQL error code: " + sqle.getErrorCode());
      sqle=sqle.getNextException();     // Retrieve next SQLException    3
    }
}
```

Figure 32. Processing an SQLException under the IBM DB2 Driver for JDBC and SQLJ

Related tasks:

- "Handling an SQLWarning under the IBM DB2 Driver for JDBC and SQLJ" on page 81

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.

```
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              1
while (sqlwarn != null) {         // While there are warnings, get and     2a
                                  // 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             2f
}
```

Figure 33. Processing an SQLWarning

Related tasks:

- "Handling an SQLException under the DB2 JDBC Type 2 Driver" on page 84

IBM DB2 Driver for JDBC and SQLJ client reroute support

The DB2 Database for Linux, UNIX, and Windows automatic client reroute feature allows client applications to recover from a loss of communication with the server so that they can continue to work with minimal interruption.

Whenever a server crashes, each client that is connected to that server receives a communication error, which terminates the connection and results in an application error. When availability is important, you should have a redundant setup or failover support. Failover is the ability of a server to take over operations when another server fails. In either case, the IBM DB2 Driver for JDBC and SQLJ client attempts to reestablish the connection to the original server or to a new server. When the connection is reestablished, the application receives an `SQLException` that informs it of the transaction failure, but the application can continue with the next transaction.

IBM DB2 Driver for JDBC and SQLJ client reroute support is available only for connections that are obtained using a `DataSource` interface. The `DriverManager` interface is not supported.

Before a client application can recover from a loss of communication, an alternate server location must be specified at the server. The database administrator specifies the alternate server with the UPDATE ALTERNATE SERVER FOR DATABASE command.

After the database administrator specifies the alternate server location on a particular database at the server instance, the primary and alternate server locations are returned to the client at connect time. The IBM DB2 Driver for JDBC and SQLJ creates an instance of the `DB2ClientRerouteServerList` class, which implements the `javax.naming.Referenceable` interface, and stores that instance in its transient memory. If communication is lost, the IBM DB2 Driver for JDBC and SQLJ tries to reestablish the connection using the server location information that is returned from the server.

`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.

The clientRerouteServerListJNDIName property of a `DataSource` provides additional client reroute support at the client. clientRerouteServerListJNDIName has two functions:
- Allows information about reroute servers to persist across JVMs
- Provides an alternate server location if the first connection to the database server fails

clientRerouteServerListJNDIName identifies a JNDI reference to a `DB2ClientRerouteServerList` instance in a JNDI repository of reroute server information. After a successful connection to the primary server, the alternate server information that is provided by clientRerouteServerListJNDIName is overwritten by the information from the server.

If the clientRerouteServerListJNDIName property is defined:
- The IBM DB2 Driver for JDBC and SQLJ attempts to propagate the updated information to the JNDI store after a failover.
- 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.

When a failover connection is established, it has the original `DataSource` properties, except for the server name and port number. In addition, any DB2 special registers that were modified during the original connection are reestablished in the failover connection by the IBM DB2 Driver for JDBC and SQLJ.

When a communication failure occurs, the IBM DB2 Driver for JDBC and SQLJ first attempts recovery to the original server. Reconnection to the original server is called failback. If failback fails, the driver attempts failover to the alternate location. After a connection is reestablished, the driver throws a `java.sql.SQLException` to the application with SQLCODE -4498, to indicate to the application that the connection to the alternate server was automatically reestablished and the transaction was implicitly rolled back. The application can then retry its transaction without doing an explicit rollback first.

To set up storage to make `DB2ClientRerouteServerList` persistent, follow these steps:

1. Create an instance of `DB2ClientRerouteServerList`, and bind that instance to the JNDI registry.

   Example:

   ```
   // 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("mvs1.sj.ibm.com");
   ```

```
// Set the  port number and server name for the alternate server
int[] port = {50002};
String[] server = {"mvs3.sj.ibm.com"};
address.setAlternatePortNumber(port);
address.setAlternateServerName(server);

registry.rebind("serverList", address);
```

2. Assign the JNDI name of the `DB2ClientRerouteServerList` object to the clientRerouteServerListJNDIName property.

   Example:

   ```
   datasource.setClientRerouteServerListJNDIName("serverList");
   ```

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232
- "Summary of IBM DB2 Driver for JDBC and SQLJ extensions to JDBC" on page 301

Disconnecting from database servers 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 DB2 and JDBC resources immediately. To close the connection to the data source, use the `close` method. For example:

```
 Connection con;
 ...
con.close();
```

If autocommit mode is not on, the connection needs to be on a unit-of-work boundary before you close the connection.

Related concepts:

- "How JDBC applications connect to a data source" on page 24

Chapter 3. Programming SQLJ applications

The topics that follow contain information about writing SQLJ applications.

Basic steps in writing an SQLJ application

Writing a 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.

Figure 34 on page 90 is a simple program that demonstrates each task.

```
import sqlj.runtime.*;                                                       1
import java.sql.*;

#sql context EzSqljCtx;                                                      3a
#sql iterator EzSqljNameIter (String LASTNAME);                              4a

public class EzSqlj {
  public static void main(String args[])
    throws SQLException
  {
    EzSqljCtx ctx = null;
    String URLprefix = "jdbc:db2:";
    String url;
    url = new String(URLprefix + args[0]);
                                                // Location name is an input parameter
    String hvmgr="000010";                                                   2
    String hvdeptno="A00";
    try {                                                                    3b
      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);                   3c
                                                // Create a JDBC Connection
       con0.setAutoCommit(false);               // set autocommit OFF
       ctx = new EzSqljCtx(con0);                                            3d

       try
       {
         EzSqljNameIter iter;
         int count=0;

         #sql [ctx] iter =
           {SELECT LASTNAME FROM EMPLOYEE};                                  4b
                                                // Create result table of the SELECT
         while (iter.next()) {                                               4c
            System.out.println(iter.LASTNAME()); // Retrieve rows from result table
            count++;
         }
         System.out.println("Retrieved " + count + " rows of data");
       }
```

Figure 34. Simple SQLJ application (Part 1 of 2)

```
      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]                                                      4d
          {UPDATE DEPARTMENT SET MGRNO=:hvmgr
             WHERE DEPTNO=:hvdeptno};
                                       // Update data for one department 6
        #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();
      }
      iter.close();                                     // Close the iterator
      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();
    }

  }
```

Figure 34. Simple SQLJ application (Part 2 of 2)

Notes to Figure 34 on page 90:

Note	Description
1	These statements import the `java.sql` package, which contains the JDBC core API, and the `sqlj.runtime` package, which contains the SQLJ API. For information on other packages or classes that you might need to access, see Access Java packages for SQLJ support.
2	`String` variables hvmgr and hvdeptno are *host identifiers*, which are equivalent to DB2 host variables. See Declare variables in SQLJ applications for more information.
3a , 3b , 3c , and 3d	These statements demonstrate how to connect to a data source using one of the three available techniques. See Connect to a data source using SQLJ for more details.

Note	Description
4a , 4b , 4c , and 4d	These statements demonstrate how to execute SQL statements in SQLJ. Statement 4a demonstrates the SQLJ equivalent of declaring an SQL cursor. Statements 4b and 4c show one way of doing the SQLJ equivalent of executing SQL FETCHes. Statement 4d shows how to do the SQLJ equivalent of performing an SQL UPDATE. For more information, see Execute SQL in an SQLJ application.
5	This `try/catch` block demonstrates the use of the `SQLException` class for SQL error handling. For more information on handling SQL errors, see Handle errors in an SQLJ application. For more information on handling SQL warnings, see Handle SQL warnings in an SQLJ application.
6	This is an example of a comment. For rules on including comments in SQLJ programs, see Include comments in an SQLJ application.
7	This statement closes the connection to the data source. See Close the connection to the data source in an SQLJ application.

Related concepts:

- "Java packages for SQLJ support" on page 97
- "Variables in SQLJ applications" on page 98
- "SQL statements in an SQLJ application" on page 100

Related tasks:

- "Connecting to a data source using SQLJ" on page 92

Connecting to a data source using SQLJ

In an SQLJ application, as in any other DB2 application, you must be connected to a database server before you can execute SQL statements. In SQLJ, as in JDBC, a database server is called a *data source*.

You can use one of the following techniques to connect to a data source.

Connection technique 1: This technique uses the JDBC `DriverManager` as the underlying means for creating the connection. Use it with any level of the JDBC driver.

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 DB2 Driver for JDBC and SQLJ, invoke `Class.forName` this way:

     ```
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```
   - For the DB2 JDBC Type 2 Driver, invoke `Class.forName` this way:

     ```
     Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
     ```
3. Invoke the constructor for the connection context class that you created in step 1.

   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 DB2 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 DB2 Driver for JDBC and SQLJ, you can specify any of the properties listed in Properties for the IBM DB2 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.

```
#sql context Ctx;            // Create connection context class Ctx        1
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");                              2
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Ctx myConnCtx=                                                             3
  new Ctx("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
  userid,password,false);    // 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 35. Using connection technique 1 to connect to a data source

Connection technique 2: This technique uses the JDBC `DriverManager` interface for creating the connection. Use it with any level of the JDBC driver.

1. Execute an SQLJ connection declaration clause.

   This is the same as step 1 on page 92 in connection technique 1.
2. Load the driver.

   This is the same as step 2 on page 92 in connection technique 1.
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 DB2 Driver for JDBC and SQLJ.

   The meanings of the *url*, *user*, and *password* parameters are the same as the meanings of the parameters in step 3 on page 92 of connection technique 1.
4. Invoke the constructor for the connection context class that you created in step 1.

   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 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       1
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");                            2
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
Connection jdbccon=                                                       3
  DriverManager.getConnection("jdbc:db2://sysmvs1.stl.ibm.com:5021/NEWYORK",
    userid,password);
                              // Create JDBC connection object jdbccon
jdbccon.setAutoCommit(false); // Do not autocommit                        4
Ctx myConnCtx=new Ctx(jdbccon);                                           5
                              // 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 36. Using connection technique 2 to connect to a data source

Connection technique 3: This technique uses the JDBC `DataSource` interface for creating the connection.

1. Execute an SQLJ connection declaration clause.

   This is the same as step 1 on page 92 in connection technique 1.
2. If your system administrator created a `DataSource` object in a different program:
   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.

Otherwise, create a `DataSource` object and assign properties to it, as shown in "Creating and using a DataSource object in the same application" in Connect to a data source using the DataSource interface.

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 *user* and *password* parameters are the same as the meanings of the parameters in step 3 on page 92 of connection technique 1.
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 94.

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

```
import java.sql.*;
import javax.naming.*;
import javax.sql.*;
...
#sql context CtxSqlj;       // Create connection context class CtxSqlj   1
Context ctx=new InitialContext();                                         2b
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");                    2c
Connection con=ds.getConnection();                                        3
String empname;             // Declare a host variable
...
con.setAutoCommit(false);   // Do not autocommit                          4
CtxSqlj myConnCtx=new CtxSqlj(con);                                       5
                            // Create connection context object myConnCtx
#sql [myConnCtx] {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};
                            // Use myConnCtx for executing an SQL statement
```

Figure 37. Using connection technique 3 to connect to a data source

Connection technique 4 (IBM DB2 Driver for JDBC and SQLJ only): This technique uses the JDBC `DataSource` interface for creating the connection. This technique **requires** that the `DataSource` is registered with JNDI.

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 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:

   ```
   connection-context-class connection-context-object=
     new connection-context-class();

   connection-context-class connection-context-object=
     new connection-context-class (String user,
       String password);
   ```

   The meanings of the *user* and *password* parameters are the same as the meanings of the parameters in step 3 on page 92 of connection technique 1.

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 public static context Ctx
  with (dataSource="jdbc/sampledb");                                2
                               // 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);                            3
                               // 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 38. Using connection technique 4 to connect to a data source

Connection technique 5: This technique uses a previously created connection 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.

Example: Program CtxGen.sqlj declares connection context Ctx and creates instance oldCtx:

```
#sql context Ctx;
...
// Create connection context object oldCtx
```

Program test.sqlj receives oldCtx as a parameter and uses oldCtx as the argument of its connection context constructor:

```
void useContext(sqlj.runtime.ConnectionContext oldCtx)
                                   // 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
...
}
```

Connection technique 6: This technique 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. For example:

```
#sql {SELECT LASTNAME INTO :empname FROM EMPLOYEE
   WHERE EMPNO='000010'};  // Use default connection for
                              // executing an SQL statement
```

To create a default connection context, SQLJ does a JNDI lookup for `jdbc/defaultDataSource`. If nothing is registered, a null context exception is issued when SQLJ attempts to access the context.

Related concepts:

- "How JDBC applications connect to a data source" on page 24

Related tasks:

- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27
- "Connecting to a data source using the DataSource interface" on page 30

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

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:

sqlj.runtime
Contains the SQLJ run-time API.

java.sql
Contains the core JDBC API.

com.ibm.db2.jcc
Contains the DB2-specific implementation of JDBC and SQLJ.

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.

Related concepts:

- "Basic steps in writing an SQLJ application" on page 89

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, host variables are known as *host expressions*. A host expression can be a simple Java identifier, or it can be a complex expression. Every host expression must start with a colon when it is used in an SQL statement. Host expressions are case sensitive.

A Java identifier can have any of the data types listed in the Java data type column of Java, JDBC, and SQLJ data types. Data types that are specified in an iterator can be any of the types in the Java data type column of Java, JDBC, and SQLJ data types.

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`.

```
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 value `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.

```
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 [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.

Related concepts:

- "Basic steps in writing an SQLJ application" on page 89

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

Comments in an SQLJ application

To document your 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 database server supports a comment within the SQL statement (/* */ only).

only

Related concepts:

- "Basic steps in writing a JDBC application" on page 21

Executing SQL statements in SQLJ applications

The topics that follow contain information about executing SQL statements SQLJ applications.

- “SQL statements in an SQLJ application”
- “Updating DB2 tables in SQLJ applications” on page 101
- “Retrieving data from DB2 tables in SQLJ applications” on page 111
- “Calling stored procedures in SQLJ applications” on page 121
- “Working with LOBs in SQLJ applications” on page 124
- “Using SQLJ and JDBC in the same application” on page 127
- “Controlling the execution of SQL statements in SQLJ” on page 130
- “ROWIDs in SQLJ with the IBM DB2 Driver for JDBC and SQLJ” on page 130
- “Distinct types in SQLJ applications” on page 131
- “Savepoints in SQLJ applications” on page 132

SQL statements in an SQLJ application

You execute SQL statements in a traditional SQL program to create tables, insert, update, delete, or merge data in tables, retrieve data from the tables, call stored procedures, or commit or roll back 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 `DefaultContext`. This means that a context must already be defined as the `DefaultContext`. Use of a `DefaultContext` is not thread-safe, and is not recommended.

Related concepts:

- “Comments in an SQLJ application” on page 99
- “How an SQLJ application retrieves data from DB2 tables” on page 111
- “Retrieving multiple result sets from a stored procedure in an SQLJ application” on page 122
- “LOBs in SQLJ applications with the IBM DB2 Driver for JDBC and SQLJ” on page 124
- “Using SQLJ and JDBC in the same application” on page 127

Related tasks:

- “Calling stored procedures in an SQLJ application” on page 122
- “Committing or rolling back SQLJ transactions” on page 138
- “Controlling the execution of SQL statements in SQLJ” on page 130
- “Creating and modifying DB2 objects in an SQLJ application” on page 101
- “Handling SQL errors in an SQLJ application” on page 138

- "Handling SQL warnings in an SQLJ application" on page 139
- "Making batch updates in SQLJ applications" on page 107
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 101
- "Setting the isolation level for an SQLJ transaction" on page 138
- "Using a named iterator in an SQLJ application" on page 112
- "Using a positioned iterator in an SQLJ application" on page 114
- "Using scrollable iterators in an SQLJ application" on page 118

Related reference:
- "SQLJ executable-clause" on page 271

Updating DB2 tables in SQLJ applications

The topics that follow contain information about creating and modifying DB2 tables in SQLJ applications.
- "Creating and modifying DB2 objects in an SQLJ application"
- "Performing positioned UPDATE and DELETE operations in an SQLJ application"
- "Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application" on page 106
- "Making batch updates in SQLJ applications" on page 107

Creating and modifying DB2 objects in an SQLJ application

Use SQLJ executable clauses to do the following things:
- Execute data definition statements (CREATE, ALTER, DROP, GRANT, REVOKE)
- Execute INSERT, searched or positioned UPDATE, and searched or positioned DELETE statements

For example, 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"};
```

For information on positioned UPDATEs and DELETEs, see Perform positioned UPDATE and DELETE operations in an SQLJ application.

Related tasks:
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 101

Performing positioned UPDATE and DELETE operations in an SQLJ application

As in DB2 applications in other languages, performing positioned UPDATEs and DELETEs 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, the iterator must be declared as updatable. To do this, the declaration must include 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 DB2 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 DB2 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:

```
import java.math.*;    // Import this class for BigDecimal data type
#sql public iterator UpdByPos implements sqlj.runtime.ForUpdate      1
  with(updateColumns="SALARY") (String, BigDecimal);
```

Figure 39. 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:

```
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 connnection 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      2
                     // 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    3
  String enum;       // Declares host variable to receive EMPNO
  BigDecimal sal;    // and SALARY column values
  #sql [myConnCtx]
    upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE                    4
      WHERE WORKDEPT='D11'};
                     // Assign result table to iterator object
  #sql {FETCH :upditer INTO :enum,:sal};                             5a
                             // Move cursor to next row
  while (!upditer.endFetch())                                        5b
                             // Check if on a row
  {
    #sql [myConnCtx] {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
      WHERE CURRENT OF :upditer};                                    5c
                             // Perform positioned update
    System.out.println("Updating row for " + enum);
    #sql {FETCH :upditer INTO :enum,:sal};
                             // Move cursor to next row
  }
  upditer.close();           // Close the iterator                   6
  #sql [myConnCtx] {COMMIT};
                             // Commit the changes
  myConnCtx.close();         // Close the connection context
}
```

Figure 40. 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:

```
import java.math.*;          // Import this class for BigDecimal data type
#sql public iterator UpdByName implements sqlj.runtime.ForUpdate     1
  with(updateColumns="SALARY") (String EmpNo, BigDecimal Salary);
```

Figure 41. 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:

```
import sqlj.runtime.*;       // Import files for SQLJ and JDBC APIs
import java.sql.*;
import java.math.*;          // Import this class for BigDecimal data type
import UpdByName;            // Import the generated iterator class that
                             // was created by the iterator declaration clause
                             // for UpdByName in another file
#sql context HSCtx;          // Create a connnection 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  2
                         // do not destroy the cursor between updates
  HSCtx myConnCtx=new HSCtx(HSjdbccon);
                         // Create a connection context object
  UpdByName upditer;                                                  3
                         // Declare iterator object of UpdByName class
  String enum;               // Declare host variable to receive EmpNo
                             // column values
  #sql [myConnCtx]
    upditer = {SELECT EMPNO, SALARY FROM EMPLOYEE                     4
      WHERE WORKDEPT='D11'};
                             // Assign result table to iterator object
  while (upditer.next())                                              5a, 5b
                             // Move cursor to next row and
                             // check ifon a row
  {
    enum = upditer.EmpNo();  // Get employee number from current row
    #sql [myConnCtx]
      {UPDATE EMPLOYEE SET SALARY=SALARY*1.05
        WHERE CURRENT OF :upditer};                                   5c
                             // Perform positioned update
    System.out.println("Updating row for " + enum);
  }
  upditer.close();           // Close the iterator                    6
  #sql [myConnCtx] {COMMIT};
                             // Commit the changes
  myConnCtx.close();         // Close the connection context
}
```

Figure 42. Performing a positioned UPDATE with a named iterator

Related concepts:

- "Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application" on page 106
- "How an SQLJ application retrieves data from DB2 tables" on page 111

Related tasks:

- "Connecting to a data source using SQLJ" on page 92

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 43. Static positioned UPDATE that fails

In this example, only one iterator is defined. Two instances of that iterator are defined, 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 43 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 44 on page 107 shows an example of how to do this.

```
#sql iterator Table2Iter(String);
#sql iterator Table1Iter(String);
  public static void main ( String args[] )
  {
...
    Table2Iter iter2 = null;
    #sql [ctxt] iter2 = { SELECT CHAR_COL2 FROM TABLE2 };

    Table1Iter iter1 = null;
    #sql [ctxt] iter1 = { SELECT CHAR_COL1 FROM TABLE1 };
...

    doUpdate(iter1);

  }

 public static void doUpdate ( Table1Iter iter )
  {
    ...
    #sql [ctxt] { UPDATE TABLE1 ... WHERE CURRENT OF :iter };
    ...
  }
 public static void doUpdate ( Table2Iter iter )
  {
    ...
    #sql [ctxt] { UPDATE TABLE2 ... WHERE CURRENT OF :iter };
    ...
  }
```

Figure 44. Static positioned UPDATE that succeeds

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.

Related tasks:

- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 101

Related reference:

- "db2sqljcustomize - SQLJ profile customizer" on page 351

Making batch updates in SQLJ applications

The IBM DB2 Driver for JDBC and SQLJ supports batch updates in SQLJ. With batch updates, instead of updating rows of a DB2 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 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

The basic steps for creating, executing, and deleting a batch of statements are:

1. Disable `AutoCommit` for the connection.
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. SQLJ also executes the statement that is not compatible with the statements in the batch.
     – 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 of a batch update: In the following code fragment, raises are given to all managers by performing UPDATEs in a batch. The numbers to the right of selected statements correspond to the previously-described steps.

```
#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); 1
  ExecutionContext ec = new ExecutionContext();                      2
  ec.setBatching(true);                                              3

  #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]                                                    4
      {UPDATE EMPLOYEE SET SALARY=:(currentSalary+raise)
        WHERE EMPNO=:mgrnum};
    #sql {FETCH :deptiter INTO :mgrnum };
                                    // Fetch the next row
  }
  ec.executeBatch();                                                 5
  ec.setBatching(false);                                             6
  #sql [c1] {COMMIT};
  deptiter.close();                 // Close the iterator
  ec.close();                       // Close the execution context
  c1.close();                       // Close the connection
}
```

Figure 45. Performing a batch update

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. See Make batch updates in a JDBC application for information on how to process a `BatchUpdateException`.

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.

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.

Recommendation: Turn autocommit off when you do batch updates so that you can control whether to commit changes to already-executed statements when an error occurs during batch execution.

Related tasks:

- "Making batch updates in JDBC applications" on page 42
- "Connecting to a data source using SQLJ" on page 92
- "Controlling the execution of SQL statements in SQLJ" on page 130

Related reference:

- “sqlj.runtime.SQLNullException class” on page 298

Retrieving data from DB2 tables in SQLJ applications

The topics that follow contain information about retrieving data from DB2 tables in SQLJ applications.

- “How an SQLJ application retrieves data from DB2 tables”
- “Using a named iterator in an SQLJ application” on page 112
- “Using a positioned iterator in an SQLJ application” on page 114
- “Multiple open iterators for the same SQL statement in an SQLJ application” on page 116
- “Multiple open instances of an iterator in an SQLJ application” on page 117
- “Using scrollable iterators in an SQLJ application” on page 118

How an SQLJ application retrieves data from DB2 tables

Just as in DB2 applications in other languages, if you want to retrieve a single row from a DB2 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 [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.

The SQLJ equivalent of a cursor is a *result set iterator*. Like a cursor, a result set iterator can be non-scrollable or scrollable. 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*. Postitioned 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.

Related tasks:

- “Performing positioned UPDATE and DELETE operations in an SQLJ application” on page 101
- “Using a named iterator in an SQLJ application” on page 112

- "Using a positioned iterator in an SQLJ application" on page 114

Related reference:

- "SQLJ iterator-declaration-clause" on page 269

Using a named iterator in an SQLJ application

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` *`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.

```
#sql  iterator ByName(String LastName, Date HireDate);             1
                              // Declare named iterator ByName
{
  ByName nameiter;            // Declare object of ByName class       2
  #sql [ctxt]
 nameiter={SELECT LASTNAME, HIREDATE FROM EMPLOYEE};               3
                              // Assign the result table of the SELECT
                              // to iterator object nameiter
  while (nameiter.next())     // Move the iterator through the result 4
                              // 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                   5
}
```

Figure 46. Using a named iterator

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 127

Related tasks:

- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 101
- "Using a positioned iterator in an SQLJ application" on page 114

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227
- "SQLJ assignment-clause" on page 275
- "SQLJ context-clause" on page 272

Using a positioned iterator in an SQLJ application

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. SeeJava, 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.

```
#sql iterator ByPos(String,Date); // Declare positioned iterator ByPos  1
{
  ByPos positer;                    // Declare object of ByPos class    2
  String name = null;               // Declare host variables
  Date hrdate;
  #sql [ctxt] positer =
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                          3
                                    // Assign the result table of the SELECT
                                    // to iterator object positer
  #sql {FETCH :positer INTO :name, :hrdate };                           4
                                    // Retrieve the first row
  while (!positer.endFetch())       // Check whether the FETCH returned a row
  { System.out.println(name + " was hired in " +
      hrdate);
    #sql {FETCH :positer INTO :name, :hrdate };
                                    // Fetch the next row
  }
  positer.close();                  // Close the iterator               5
}
```

Figure 47. Using a positioned iterator

Related concepts:
- "Using SQLJ and JDBC in the same application" on page 127
- "How an SQLJ application retrieves data from DB2 tables" on page 111

Related tasks:
- "Using a named iterator in an SQLJ application" on page 112

Related reference:
- "Data types that map to SQL data types in JDBC applications" on page 227

Multiple open iterators for the same SQL statement in an SQLJ application

With the IBM DB2 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:

```
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 48 on page 117.

```
MultiIter iter1 = null;           // Iterator instance for retrieving
                                  // data for first employee
String EmpNo1 = "000100";         // Employee number for first employee
#sql [ctx] iter2 =
  {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 48. Example of concurrent table operations using iterators with different SQL statements

Figure 49 demonstrates how you can perform the same operations when you have the capability for multiple, concurrently open iterators for a single SQL statement.

```
...
MultiIter iter1 = openIter("000100"); // Invoke openIter to assign the result table
                                      // (for employee 100) to the first iterator
MultiIter iter2 = openIter("000200"); // Invoke openIter to assign the result
                                      // table to the second iterator
                                      // iter1 stays open when iter2 is opened
// 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 [ctxt] iter =
    {SELECT EMPNO, SALARY FROM EMPLOYEE WHERE EMPNO = :EmpNo};
  return iter;                        // Method returns an iterator instance
}
```

Figure 49. Example of concurrent table operations using iterators with the same SQL statement

Related concepts:
- “How an SQLJ application retrieves data from DB2 tables” on page 111

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.

```
...
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 50. Example of opening more than one instance of an iterator in a single application

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.

Related concepts:

- "How an SQLJ application retrieves data from DB2 tables" on page 111

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 DB2 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 DB2 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.

To create and use a scrollable iterator, you need to follow these steps:

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. For example, the following iterator declaration clause declares a positioned, sensitive, dynamic, scrollable iterator:

   ```
   #sql public iterator ByPos
     implements sqlj.runtime.Scrollable
     with (sensitivity=SENSITIVE, dynamic=true)  (String);
   ```

   The following iterator declaration clause declares a named, insensitive, scrollable iterator:

   ```
   #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
   - DATALINK
   - BLOB
   - CLOB
   - 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 Table 15.

Table 15. sqlj.runtime.Scrollable methods for positioning a scrollable cursor

Method	**Positions the cursor**
`first()`	On the first row of the result table
`last()`	On the last row of the result table
`previous()`[1]	On the previous row of the result table
`next()`	On the next row of the result table

Table 15. sqlj.runtime.Scrollable methods for positioning a scrollable cursor (continued)

Method	Positions the cursor
`absolute(int n)`[2]	If $n>0$, on row n of the result table. If $n<0$, and m is the number of rows in the result table, on row $m+n+1$ of the result table.
`relative(int n)`[3]	If $n>0$, on the row that is n rows after the current row. If $n<0$, on the row that is n rows before the current row. If $n=0$, on the current row.
`afterLast()`	After the last row in the result table
`beforeFirst()`	Before the first row in the result table

Notes:

1. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
2. 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.
3. 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 the `getRow`, `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 16 lists the clauses that you can use to position the cursor.

Table 16. FETCH clauses for positioning a scrollable cursor

Method	Positions the cursor
FIRST	On the first row of the result table
LAST	On the last row of the result table
PRIOR[1]	On the previous row of the result table
NEXT	On the next row of the result table
ABSOLUTE(*n*)[2]	If $n>0$, on row n of the result table. If $n<0$, and m is the number of rows in the result table, on row $m+n+1$ of the result table.
RELATIVE(*n*)[3]	If $n>0$, on the row that is n rows after the current row. If $n<0$, on the row that is n rows before the current row. If $n=0$, on the current row.
AFTER[4]	After the last row in the result table
BEFORE[4]	Before the first row in the result table

Table 16. FETCH clauses for positioning a scrollable cursor (continued)

Method	Positions the cursor

Notes:

1. If the cursor is after the last row of the result table, this method positions the cursor on the last row.
2. 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.
3. 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.
4. Values are not assigned to host expressions.

   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.

For example, 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.

```
#sql  iterator ScrollIter implements sqlj.runtime.Scrollable        1
  (String EmpNo, String LastName);
{
  ScrollIter scrliter;                                              2
  #sql [ctxt]
  scrliter={SELECT EMPNO, LASTNAME FROM EMPLOYEE};
  scrliter.afterLast();
  while (scrliter.previous())                                       4a
  {
    System.out.println(scrliter.EmpNo() + " "                       4c
      + scrliter.LastName());
  }
  scrliter.close();                                                 5
}
```

Figure 51. Using scrollable iterators

Related concepts:

- "How an SQLJ application retrieves data from DB2 tables" on page 111

Related tasks:

- "Using a named iterator in an SQLJ application" on page 112
- "Using a positioned iterator in an SQLJ application" on page 114

Calling stored procedures in SQLJ applications

The topics that follow contain information about calling stored procedures in SQLJ applications.

- "Calling stored procedures in an SQLJ application" on page 122
- "Retrieving multiple result sets from a stored procedure in an SQLJ application" on page 122

Calling stored procedures in an SQLJ application

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. See Retrieve multiple result sets from a stored procedure in an SQLJ application.

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.

```
String FirstName="TOM";           // Input parameters                  1
String LastName="NARISINST";
String Address="IBM";
int CustNo;                       // Output parameters
String Mark;
String MarkErrorText;
...
#sql [myConnCtx] {CALL ADD_CUSTOMER(:IN FirstName,                     2
                           :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); 3
System.out.println(Mark);
If (MarkErrorText != null)
  System.out.println(" Error messages:" + MarkErrorText);
```

Figure 52. Calling a stored procedure in an SQLJ application

Related concepts:

- "Retrieving multiple result sets from a stored procedure in an SQLJ application" on page 122

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. 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.

```
ExecutionContext execCtx=myConnCtx.getExecutionContext();      1
#sql [myConnCtx, execCtx] {CALL MULTRSSP()};                   2
            // MULTRSSP returns multiple result sets
ResultSet rs;
while ((rs = execCtx.getNextResultSet()) != null)              3a
{
  ResultSetMetaData rsmeta=rs.getMetaData();                   3b
  int numcols=rsmeta.getColumnCount();
  while (rs.next())                                            3c
  {
    for (int i=1; i<=numcols; i++)
    {
      String colval=rs.getString(i);
      System.out.println("Column " + i + "value is " + colval);
    }
  }
}
```

Figure 53. Retrieving result sets from a stored procedure

Working with LOBs in SQLJ applications

The topics that follow contain information about updating and retrieving LOB data in SQLJ applications.

- "LOBs in SQLJ applications with the IBM DB2 Driver for JDBC and SQLJ"
- "Java data types for retrieving or updating LOB column data in SQLJ applications" on page 125

LOBs in SQLJ applications with the IBM DB2 Driver for JDBC and SQLJ

With the IBM DB2 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 support: If the database server supports progressive streaming, the IBM DB2 Driver for JDBC and SQLJ can use progressive streaming to retrieve data in LOB or XML columns. With progressive streaming, the database server dynamically determines the most efficient mode in which to return LOB or XML data, based on the size of the LOBs or XML objects. To cause SQLJ to use progressive streaming to retrieve data, you need to set the `progressiveStreaming` property to `DB2DatabaseMetaData.YES`. When you use progressive streaming, 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.

Use of progressive streaming is the preferred method of LOB or XML data retrieval. To determine whether a database server supports progressive streaming, invoke the JDBC `DB2DatabaseMetaData.supportsDB2ProgressiveStreaming` method.

LOB locator support: The IBM DB2 Driver for JDBC and SQLJ can use LOB locators to retrieve data in LOB columns. You should use LOB locators only if the database server does not support progressive streaming. To cause SQLJ 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 `DB2BaseDataSource.NO`. If you do not set `progressiveStreaming` to `DB2BaseDataSource.NO`, and the database serve supports progressive streaming, the JDBC driver ignores the `fullyMaterializeLobData` value.

`fullyMaterializeLobData` has no effect on stored procedure output parameters or LOBs that are fetched using scrollable cursors. You cannot call a stored procedure that has LOB locator parameters. When you fetch from scrollable cursors, SQLJ always uses LOB locators to retrieve data from LOB columns.

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

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

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

When the deferPrepares property is set to true, and the IBM DB2 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.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. For example, if you are using the IBM DB2 Driver for JDBC and SQLJ, you can use the DB2-only method `com.ibm.db2.jcc.t2zos.DB2LobFactory.createBlob` to create a java.sql.Blob object and populate the object with byte[] data:

  ```
  byte[] byteArray = {0, 1, 2, 3};
  java.sql.Blob blobData =
    com.ibm.db2.jcc.t2zos.DB2LobFactory.createBlob(byteArray);
  ```
- Use an input parameter of type of `sqlj.runtime.BinaryStream`. A `sqlj.runtime.BinaryStream` object is compatible with a BLOB data type. For this call, you need to specify the exact length of the input data:

  ```
  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 input/output parameters.

Output parameters for BLOB columns:

For output or input/output parameters for BLOB columns, you can use the following technique:

- Declare the output parameter or input/output variable with a java.sql.Blob data type:

  ```
  java.sql.Blob blobData = null;
  #sql CALL STORPROC (:OUT blobData)};
  ```

```
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:

  ```
  #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. For example, if you are using the IBM DB2 Driver for JDBC and SQLJ, you can use the DB2-only method `com.ibm.db2.jcc.t2zos.DB2LobFactory.createClob` to create a java.sql.Clob object and populate the object with String data:

  ```
  String stringVal = "Some Data";
  java.sql.Clob clobData =
    com.ibm.db2.jcc.t2zos.DB2LobFactory.createClob(stringVal);
  ```
- Use one of the following types of stream input parameters:
  - A `sqlj.runtime.CharacterStream` input parameter:

    ```
    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:

    ```
    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:

    ```
    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 input/output parameters.
- Use a java.lang.String input parameter:

  ```
  java.lang.String charData;
  #sql {CALL STORPROC(:IN charData)};
  ```

Output parameters for CLOB columns:

For output or input/output 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.sql.Clob clobData = null;
  #sql CALL STORPROC(:OUT clobData)};
  ```
- Use a java.lang.String output variable:

  ```
  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 input/output parameters for stored procedures are not supported.

Related concepts:

- "LOBs in SQLJ applications with the IBM DB2 Driver for JDBC and SQLJ" on page 124

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

Using 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 JDBC `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 54 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);      1
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();                          2
  String query = "SELECT LASTNAME, HIREDATE FROM EMPLOYEE";
  query+=whereClause;   // Build the query
  ResultSet rs = stmt.executeQuery(query);                          3
  #sql [connCtx] nameiter = {CAST :rs};                             4
  while (nameiter.next())
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate());

  }
  nameiter.close();                                                 5
  stmt.close();
}
```

Figure 54. Converting a JDBC result set to an SQLJ iterator

Notes to Figure 54:

Note	Description
1	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.
2	This statement and the following two statements build and prepare a query for dynamic execution using JDBC.
3	This JDBC statement executes the SELECT statement and assigns the result table to result set rs.
4	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.
5	The `nameiter.close()` method closes the SQLJ iterator and JDBC `ResultSet` rs.

Generating JDBC ResultSets from SQLJ iterators: Use the getResultSet method to generate a JDBC `ResultSet` from an SQLJ iterator. Every SQLJ iterator has a getResultSet method. After you convert an iterator to a result set, you need to fetch rows using only the result set.

The code in Figure 55 generates a positioned iterator for a query, converts the iterator to a result set, and uses JDBC methods to fetch rows from the table.

```
#sql iterator EmpIter(String, java.sql.Date);
{
...
  EmpIter iter=null;
  #sql [connCtx] iter=
    {SELECT LASTNAME, HIREDATE FROM EMPLOYEE};                 1
  ResultSet rs=iter.getResultSet();                            2
  while (rs.next())                                            3
  { System.out.println(rs.getString(1) + " was hired in " +
      rs.getDate(2));
  }
  rs.close();                                                  4
}
```

Figure 55. Converting an SQLJ iterator to a JDBC ResultSet

Notes to Figure 55:

Note	Description
1	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.
2	The getResultSet() method converts iterator iter to ResultSet rs.
3	The JDBC getString() and getDate() methods retrieve values from the ResultSet. The next() method moves the cursor to the next row in the ResultSet.
4	The rs.close() method closes the SQLJ iterator as well as the ResultSet.

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 DB2 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 a generated ResultSet object or the underlying iterator at the end of the program.

  Closing the iterator object from which a ResultSet object is generated also closes the ResultSet object. Closing the generated ResultSet object also closes the iterator object. In general, it is best to close the object that is used last.
- For the IBM DB2 Driver for JDBC and SQLJ, which supports scrollable iterators and scrollable and updatable ResultSets, the following restrictions apply:
  - Scrollable iterators have the same restrictions as their underlying JDBC ResultSets. For example, because scrollable ResultSets do not support INSERTs, scrollable iterators do not support INSERTs.
  - You cannot cast a JDBC ResultSet that is not updatable to an SQLJ iterator that is updatable.

Related tasks:

- “Connecting to a data source using SQLJ” on page 92

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 an *execution context*.

   There are two ways to acquire an execution context:

   - Acquire the default execution context from the connection context. For example:

     ```
     ExecutionContext execCtx = connCtx.getExecutionContext();
     ```

   - Create a new execution context by invoking the constructor for `ExecutionContext`. For example:

     ```
     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. For example:

   ```
   #sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};
   ```

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:

   ```
   #sql [connCtx, execCtx] {DELETE FROM EMPLOYEE WHERE SALARY > 10000};
   System.out.println("Deleted " + execCtx.getUpdateCount() + " rows");
   ```

ROWIDs in SQLJ with the IBM DB2 Driver for JDBC and SQLJ

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

If you use ROWIDs in SQLJ programs, you need to customize those programs.

The IBM DB2 Driver for JDBC and SQLJ provides the DB2-only class `com.ibm.db2.jcc.DB2RowID` that you can use in iterators and in CALL statement parameters. For an iterator, you can also use the byte[] object type to retrieve ROWID values.

Figure 56 on page 131 shows an example of an iterator that is used to select values from a ROWID column:

```
#sql iterator PosIter(int,String,com.ibm.db2.jcc.DB2RowId);
                               // 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
  com.ibm.db2.jcc.DB2RowId 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 ");
     printBytes(rowid.getBytes());
                               // Use the DB2-only method getBytes to
                               // convert the value to bytes for printing
      #sql {FETCH :positrowid INTO :id, :i_fmt, :rowid};
                               // Retrieve the next row
    }
  positrowid.close();          // Close the iterator
}
```

Figure 56. Example of using an iterator to retrieve ROWID values

Figure 57 shows an example of calling a stored procedure that takes three ROWID parameters: an IN parameter, an OUT parameter, and an INOUT parameter.

```
com.ibm.db2.jcc.DB2RowId in_rowid = rowid;
com.ibm.db2.jcc.DB2RowId out_rowid = null;
com.ibm.db2.jcc.DB2RowId inout_rowid = rowid;
                                  // Declare an input, output, and
                                  // input/output 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("Output parameter value ");
printBytes(out_rowid.getBytes());
                               // Use the DB2-only method getBytes to
                               // convert the value to bytes for printing
System.out.println("Input/output parameter value ");
printBytes(inout_rowid.getBytes());
```

Figure 57. Example of calling a stored procedure with a ROWID parameter

Related reference:
- "Data types that map to SQL data types in JDBC applications" on page 227

Distinct types in SQLJ applications

In DB2, 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 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 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:

```
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 58. Defining and using a distinct type

Related reference:

- "CREATE DISTINCT TYPE statement" in *SQL Reference, Volume 2*

Savepoints in SQLJ applications

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.

Under the IBM DB2 Driver for JDBC and SQLJ, you can include any form of the SQL SAVEPOINT statement in your SQLJ program.

The following example demonstrates how to set a savepoint, roll back to the savepoint, and release the savepoint.

```
#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);
                              // Create connection context object myConnCtx
                              // for the connection to NEWYORK
...                             // Perform some SQL
#sql [ctxt] {COMMIT};         // Commit the transaction
                              // Commit the create
#sql [ctxt]
  {INSERT INTO EMP_SHOE VALUES ('000010', 6)};
                              // Insert a row
#sql [ctxt]
  {SAVEPOINT SVPT1 ON ROLLBACK RETAIN CURSORS};
                              // Create a savepoint
...
#sql [ctxt]
  {INSERT INTO EMP_SHOE VALUES ('000020', 10)};
                               // Insert another row
#sql [ctxt] {ROLLBACK TO SAVEPOINT SVPT1};
                               // Roll back work to the point
                               //  after the first insert
...
#sql [ctxt] {RELEASE SAVEPOINT SVPT1};
                               // Release the savepoint
ctx.close();                   // Close the connection context
```

Figure 59. Setting, rolling back to, and releasing a savepoint in an SQLJ application

Related tasks:
- “Committing or rolling back SQLJ transactions” on page 138

Related reference:
- “RELEASE SAVEPOINT statement” in *SQL Reference, Volume 2*
- “ROLLBACK statement” in *SQL Reference, Volume 2*
- “SAVEPOINT statement” in *SQL Reference, Volume 2*

Working with XML data in SQLJ applications

The topics that follow contain information about updating and retrieving XML data in SQLJ applications.
- “XML data in SQLJ applications”
- “XML column updates in SQLJ applications” on page 134
- “XML data retrieval in SQLJ applications” on page 136

XML data in SQLJ applications

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.

In applications, XML data is in the serialized string format.

In SQLJ applications, you can:
- Store an entire XML document in an XML column using INSERT or UPDATE 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 into a serialized XML string in the database, and then using using single-row SELECT statements or iterators 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 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.

Java has no XML data type, and invocations of metadata methods, such as `ResultSetMetaData.getColumnTypeName` return a type of java.sql.Types.OTHER for an XML column type.

Related concepts:
- "Encoding considerations for XML data in JDBC, SQLJ, and .NET applications" in *XML Guide*
- "XML column updates in SQLJ applications" on page 134
- "XML data retrieval in SQLJ applications" on page 136

XML column updates in SQLJ applications

When you update or insert data into XML columns of a DB2 table, the input data must be in the serialized string format. The host expression data types that you can use to update XML columns are:
- com.ibm.db2.jcc.DB2Xml
- String
- byte
- Blob
- Clob
- sqlj.runtime.AsciiStream
- sqlj.runtime.BinaryStream
- sqlj.runtime.CharacterStream

For stream types, you need to use an sqlj.runtime.*type*Stream host expression, rather than a java.io.*type*InputStream host expression so that you can pass the length of the stream to the JDBC driver.

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 database server 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 database server as character data, but the data contains encoding information. The database server handles incompatibilities between internal and external encoding as follows:

- If the database server is DB2 Database for Linux, UNIX, and Windows, the database server 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 server ignores the internal encoding.
- If the database server is DB2 for z/OS, the database server 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 on a DB2 database server. xmlString is a character type, so its external encoding is used, whether or not it has an internal encoding specification.

```
#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 on a DB2 database server.

```
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 DB2 database server.

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 DB2 database server.

```
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 database server.

Example: sqlj.runtime.CharacterStream host expression: Suppose that you construct an sqlj.runtime.CharacterStream host expression, and insert data from the sqlj.runtime.CharacterStream host expression into an XML column in a table on a DB2 database server.

```
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 com.ibm.db2.jcc.DB2Xml host expression, and insert the data into an XML column in a table on a DB2 database server.

```
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 DB2 table, the output data is in the serialized string format.

The host expression or iterator data types that you can use to retrieve data from XML columns are:
- com.ibm.db2.jcc.DB2Xml
- String
- byte[]
- sqlj.runtime.AsciiStream
- sqlj.runtime.BinaryStream
- sqlj.runtime.CharacterStream

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 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. Table 17 lists the methods that you can call to retrieve data from an com.ibm.db2.jcc.DB2Xml object, and the corresponding output data types and type of encoding in the XML declarations.

Table 17. DB2Xml methods, data types, and added encoding specifications

Method	Output data type	Type of XML internal encoding declaration added
DB2Xml.getDB2AsciiStream	InputStream	None
DB2Xml.getDB2BinaryStream	InputStream	None
DB2Xml.getDB2Bytes	byte[]	None
DB2Xml.getDB2CharacterStream	Reader	None
DB2Xml.getDB2String	String	None
DB2Xml.getDB2XmlAsciiStream	InputStream	US-ASCII

Table 17. DB2Xml methods, data types, and added encoding specifications (continued)

Method	Output data type	Type of XML internal encoding declaration added
DB2Xml.getDB2XmlBinaryStream	InputStream	Specified by `getDB2XmlBinaryStream` *targetEncoding* parameter
DB2Xml.getDB2XmlBytes	byte[]	Specified by DB2Xml.getDB2XmlBytes *targetEncoding* parameter
DB2Xml.getDB2XmlCharacterStream	Reader	ISO-10646-UCS-2
DB2Xml.getDB2XmlString	String	ISO-10646-UCS-2

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 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 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 the data is converted from UTF-8 to the internal encoding, and returned without any XML declaration.

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 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.

Transaction control in SQLJ applications

The topics that follow discuss control of DB2 transactions in SQLJ applications.
- "Setting the isolation level for an SQLJ transaction" on page 138
- "Committing or rolling back SQLJ transactions" on page 138

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. Table 18 shows the values that you can specify in the SET TRANSACTION ISOLATION LEVEL clause and their DB2 equivalents.

Table 18. Equivalent SQLJ and DB2 isolation levels

SET TRANSACTION value	DB2 isolation level
SERIALIZABLE	Repeatable read
REPEATABLE READ	Read stability
READ COMMITTED	Cursor stability
READ UNCOMMITTED	Uncommitted read

The isolation level affects the underlying JDBC connection as well as the SQLJ connection.

Related concepts:
- "Isolation levels" in *SQL Reference, Volume 1*

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:

```
#sql [myConnCtx] {COMMIT};
#sql [myConnCtx] {ROLLBACK};
```

Related concepts:
- "Savepoints in SQLJ applications" on page 132

Related tasks:
- "Connecting to a data source using SQLJ" on page 92

Handling errors and warnings in SQLJ applications

The topics that follow explain how to handle SQL errors and warnings in SQLJ applications.
- "Handling SQL errors in an SQLJ application"
- "Handling SQL warnings in an SQLJ application" on page 139

Handling SQL errors in an SQLJ application

SQLJ clauses use the JDBC class `java.sql.SQLException` for error handling. 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 can use the `getErrorCode` method to retrieve SQL error codes and the `getSQLState` method to retrieve SQLSTATEs.

To handle SQL errors in your SQLJ application, import the `java.sql.SQLException` class, and use the Java error handling `try/catch` blocks to modify program flow when an SQL error occurs. For example:

```
try {
  #sql [ctxt] {SELECT LASTNAME INTO :empname
    FROM EMPLOYEE WHERE EMPNO='000010'};
}
catch(SQLException e) {
  System.out.println("Error code returned: " + e.getErrorCode());
}
```

With the IBM DB2 Driver for JDBC and SQLJ, you can retrieve the SQLCA. For information on writing code to retrieve the SQLCA with the IBM DB2 Driver for JDBC and SQLJ, see Handle an SQLException under the IBM DB2 Driver for JDBC and SQLJ.

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.

Related tasks:

- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77

Handling SQL warnings in an SQLJ application

Other than a +100 SQL error code on a SELECT INTO statement, DB2 warnings do not throw SQLExceptions. To handle DB2 warnings, you need to give the program access to the `java.sql.SQLWarning` class. If you want to retrieve DB2-specific information about a warning, you also need to give the program access to the `com.ibm.db2.jcc.DB2Diagnosable` interface and the `com.ibm.db2.jcc.DB2Sqlca` class. To check for a DB2 warning, 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.

To retrieve DB2-specific information from the `SQLWarning` object with the IBM DB2 Driver for JDBC and SQLJ, follow the instructions in Handle an SQLException under the IBM DB2 Driver for JDBC and SQLJ.

Before you can execute `getWarnings` for an SQL clause, you need to 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. The following example demonstrates how to retrieve an `SQLWarning` object for an SQL clause with execution context execCtx:

```
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)
System.out.println("SQLWarning " + sqlWarn);
```

Related tasks:

- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77

- "Controlling the execution of SQL statements in SQLJ" on page 130
- "Handling SQL errors in an SQLJ application" on page 138

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. For example:

```
...
ctx = new EzSqljctx(con0);          // Create a connection context object
                                    // from JDBC connection con0
...                                   // Perform various SQL operations
 EzSqljctx.close(ConnectionContext.KEEP_CONNECTION);
                                    // Close the connection context but keep
                                    // the connection to the data source open
```

Related tasks:

- "Connecting to a data source using SQLJ" on page 92

Chapter 4. JDBC and SQLJ security

The sections that follow contain information on security mechanisms that are available under the JDBC drivers.

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:

```
import java.sql.*;        // JDBC base
...
String id = "db2adm";      // Set user ID
Sring 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:

```
import java.sql.*;                          // JDBC base
import COM.ibm.db2.jdbc.*;                  // DB2 implementation of JDBC
...
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:

```
import java.sql.*;                        // JDBC base
import COM.ibm.db2.jdbc.*;                // DB2 implementation of JDBC
...
Context ctx=new InitialContext();         // Create context for JNDI
DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");
                                          // Get DataSource object
String id = "db2adm";                     // Set user ID
Sring 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 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
```

Related concepts:

- "How DB2 applications connect to a data source using the DriverManager interface with the DB2 JDBC Type 2 Driver" on page 25

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 30
- "Creating and deploying DataSource objects" on page 33

Security under the IBM DB2 Driver for JDBC and SQLJ

When you use the IBM DB2 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.

Table 19 on page 143 lists the security mechanisms that the IBM DB2 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 DB2 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 19. Security mechanisms supported by the IBM DB2 Driver for JDBC and SQLJ

Security mechanism	`securityMechanism` property value
User ID and password	`DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY`
User ID only	`DB2BaseDataSource.USER_ONLY_SECURITY`
User ID and encrypted password	`DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY`
Encrypted user ID and encrypted password	`DB2BaseDataSource.ENCRYPTED_USER_AND_PASSWORD_SECURITY`
Encrypted user ID and encrypted security-sensitive data	`DB2BaseDataSource.ENCRYPTED_USER_AND_DATA_SECURITY`
Encrypted user ID, encrypted password, and encrypted security-sensitive data	`DB2BaseDataSource.ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY`
Kerberos[1]	`DB2BaseDataSource.KERBEROS_SECURITY`
Plugin[2]	`DB2BaseDataSource.PLUGIN_SECURITY`

Note:

1. Available for IBM DB2 Driver for JDBC and SQLJ type 4 connectivity only.
2. Available for IBM DB2 Driver for JDBC and SQLJ type 4 connectivity to DB2 Database for Linux, UNIX, and Windows servers only.

Table 20Tshows possible DB2 Database for Linux, UNIX, and Windows server authentication types and the compatible IBM DB2 Driver for JDBC and SQLJ securityMechanism property values.

Table 20. Compatible DB2 Database for Linux, UNIX, and Windows server authentication types and IBM DB2 Driver for JDBC and SQLJ securityMechanism values

DB2 Database for Linux, UNIX, and Windows server authentication type	securityMechanism setting
CLIENT	USER_ONLY_SECURITY
SERVER	CLEAR_TEXT_PASSWORD_SECURITY
SERVER_ENCRYPT	CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, or ENCRYPTED_USER_AND_PASSWORD_SECURITY
DATA_ENCRYPT	ENCRYPTED_USER_AND_DATA_SECURITY or ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY
KERBEROS	KERBEROS_SECURITY or PLUGIN_SECURITY[2]
KRB_SERVER_ENCRYPT	KERBEROS_SECURITY , PLUGIN_SECURITY[1], ENCRYPTED_PASSWORD_SECURITY, or ENCRYPTED_USER_AND_PASSWORD_SECURITY
GSSPLUGIN	PLUGIN_SECURITY[1] or KERBEROS_SECURITY
GSS_SERVER_ENCRYPT[3]	CLEAR_TEXT_PASSWORD_SECURITY, ENCRYPTED_PASSWORD_SECURITY, ENCRYPTED_USER_AND_PASSWORD_SECURITY, PLUGIN_SECURITY, or KERBEROS_SECURITY

Table 20. Compatible DB2 Database for Linux, UNIX, and Windows server authentication types and IBM DB2 Driver for JDBC and SQLJ securityMechanism values (continued)

DB2 Database for Linux, UNIX, and Windows server authentication type	securityMechanism setting
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 GSSPLUGIN and SERVER_ENCRYPT.	

Related concepts:

- "Encrypted password security or encrypted user ID and encrypted password security under the IBM DB2 Driver for JDBC and SQLJ" on page 146
- "Kerberos security under the IBM DB2 Driver for JDBC and SQLJ" on page 148
- "User ID and password security under the IBM DB2 Driver for JDBC and SQLJ" on page 144
- "User ID-only security under the IBM DB2 Driver for JDBC and SQLJ" on page 146

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

User ID and password security under the IBM DB2 Driver for JDBC and SQLJ

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:

```
import java.sql.*;         // JDBC base
...
String id = "db2adm";      // Set user ID
String pw = "db2adm";      // 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:

```
import java.sql.*;         // JDBC base
...
String url =
  "jdbc:db2://mvs1.sj.ibm.com:5021/san_jose:user=db2adm;password=db2adm;";
                           // 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:

```
import java.sql.*;                       // JDBC base
import com.ibm.db2.jcc.*;                // DB2 implementation of JDBC
...
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
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.*;                // DB2 implementation of JDBC
...
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. 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 db2ds =    // Create DB2SimpleDataSource object
  new com.ibm.db2.jcc.DB2SimpleDataSource();
db2ds.setDriverType(4);                        // Set driver type
db2ds.setDatabaseName("san_jose");             // Set location
db2ds.setServerName("mvs1.sj.ibm.com");        // Set server name
db2ds.setPortNumber(5021);                     // Set port number
db2ds.setUser("db2adm");                       // Set user ID
db2ds.setPassword("db2adm");                   // Set password
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.CLEAR_TEXT_PASSWORD_SECURITY);
                                               // Set security mechanism to
                                               // user ID and password
```

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 30
- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27
- "Creating and deploying DataSource objects" on page 33

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

User ID-only security under the IBM DB2 Driver for JDBC and SQLJ

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.*;                   // DB2 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.*;                 // DB2 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 security or encrypted user ID and encrypted password security under the IBM DB2 Driver for JDBC and SQLJ

If you use encrypted password security or encrypted user ID and encrypted password security, the IBM Java Cryptography Extension (ibmjceprovidere.jar) must be installed on your client.

You can also use encrypted security-sensitive data in addition to encrypted user ID security or encrypted password security when you access a DB2 for z/OS server. You specify encryption of security-sensitive data through the ENCRYPTED_USER_AND_DATA_SECURITY or ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY securityMechanism value. DB2 for z/OS encrypts the following data when you specify encryption of security-sensitive data:
- SQL statements that are being prepared, executed, or bound into a DB2 package

- Input and output parameter information
- Result sets
- LOB data
- Results of describe operations

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:

```
import java.sql.*;                        // JDBC base
import com.ibm.db2.jcc.*;                 // DB2 implementation of 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("securityMechanism",
  new String("" + com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY +
  ""));
                                          // Set security mechanism to
                                          // user ID and encrypted 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 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:

```
import java.sql.*;                        // JDBC base
import com.ibm.db2.jcc.*;                 // DB2 implementation of 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.setServerName("mvs1.sj.ibm.com");
                                          // Set the server name
db2ds.setPortNumber(5021);                // Set the port number
db2ds.setUser("db2adm");                  // Set the user ID
db2ds.setPassword("db2adm");              // Set the password
db2ds.setSecurityMechanism(
  com.ibm.db2.jcc.DB2BaseDataSource.ENCRYPTED_PASSWORD_SECURITY);
                                          // Set security mechanism to
                                          // User ID and encrypted password
```

Related tasks:
- "Connecting to a data source using the DataSource interface" on page 30
- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27
- "Creating and deploying DataSource objects" on page 33

Related reference:

- “Properties for the IBM DB2 Driver for JDBC and SQLJ” on page 232

Kerberos security under the IBM DB2 Driver for JDBC and SQLJ

JDBC support for Kerberos security is available for IBM DB2 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

Using 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 DB2 server.

You need to set the user, password, kerberosServerPrincipal, and securityMechanism properties. The kerberosServerPrincipal property specifies the principal name that the DB2 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:

```
import java.sql.*;                       // JDBC base
import com.ibm.db2.jcc.*;                // DB2 implementation of 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:

```
import java.sql.*;                       // JDBC base
import com.ibm.db2.jcc.*;                // DB2 implementation of 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
```

Using 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 DB2 server.

You need to set the kerberosServerPrincipal and securityMechanism properties.

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:

```
import java.sql.*;                       // JDBC base
import com.ibm.db2.jcc.*;                // DB2 implementation of 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:

```
import java.sql.*;                       // JDBC base
import com.ibm.db2.jcc.*;                // DB2 implementation of 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
```

Using Kerberos security with a delegated credential from another principal:

For this case, you authenticate to the DB2 server using a delegated credential that another principal passes to you.

You need to set the `kerberosServerPrincipal`, `gssCredential`, and `securityMechanism` properties.

*For the **`DriverManager`** interface:* Set the Kerberos server, delegated credential, and security mechanism by setting the `kerberosServerPrincipal`, 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:

```
import java.sql.*;                        // JDBC base
import com.ibm.db2.jcc.*;                 // DB2 implementation of 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:

```
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
```

Related tasks:
- "Connecting to a data source using the DataSource interface" on page 30
- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27
- "Creating and deploying DataSource objects" on page 33

Related reference:
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

IBM DB2 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 DB2 Driver for JDBC and SQLJ provides security plugin support. This support is available for IBM DB2 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 database 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 DB2 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 DB2 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 wheter 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 database server, which performs user ID and password checking. This sample is a Java version of the C language sample program gssapi_simple.c.

JCCKerberosPlugin.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 21. Connection or DataSource property settings for Java security plugin use

Property	Setting
com.ibm.db2.jcc.DB2BaseDataSource.user	The user ID under which the `Connection` is to be obtained
com.ibm.db2.jcc.DB2BaseDataSource.password	The password for the user ID
com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism	com.ibm.db2.jcc.DB2BaseDataSource.PLUGIN_SECURITY
com.ibm.db2.jcc.DB2BaseDataSource.pluginName	The name of the plugin module for a server-side security plugin
com.ibm.db2.jcc.DB2BaseDataSource.plugin	The plugin object for a client-side security plugin

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.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);
```

Related concepts:

- "Kerberos authentication details" in *Administration Guide: Implementation*
- "Security plug-ins" in *Administrative API Reference*

Related reference:

- "DB2JCCPlugin class" on page 321

- "Java plug-in samples" on page 178

IBM DB2 Driver for JDBC and SQLJ trusted context support

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 DB2 database server is made, the middleware server can use that database connection under a different user without reauthenticating the new user at the database server.

The IBM DB2 Driver for JDBC and SQLJ provides methods that allow you to establish and use trusted connections in Java programs.

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 `getDB2XAConnection` 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 DB2 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:

```
// The first item that was obtained from the previous getTrustedPooledConnection
// 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 getTrustedPooledConnection
// 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);
```

Related reference:

- "DB2PooledConnection class" on page 322
- "DB2XADataSource class" on page 331
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232
- "DB2Connection interface" on page 306
- "DB2ConnectionPoolDataSource class" on page 318

Security for preparing SQLJ applications with the IBM DB2 Driver for JDBC and SQLJ

This topic contains information about the following aspects of SQLJ security:
- Allowing users to customize only
- Limiting 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 DB2 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.

Related reference:
- "db2sqljbind - SQLJ profile binder" on page 361
- "db2sqljcustomize - SQLJ profile customizer" on page 351

Chapter 5. Building Java database applications

The following topics contain information on building JDBC and SQLJ applications.

Building JDBC applets

`Applt` demonstrates a dynamic SQL Java applet to access a DB2 database server.

Procedure:

To build and run the JDBC applet, `Applt`, by commands entered at the command line, either ensure that a web server is installed and running on your DB2 database (server or client), or use the applet viewer that comes with the software development kit for Java by entering the following command in the working directory of your client machine:

```
appletviewer Applt.html
```

Connecting with the IBM DB2 Driver for JDBC and SQLJ: To connect with the IBM DB2 Driver for JDBC and SQLJ, modify the `Applt.html` file according to the instructions in the file. For the TCP/IP port number, you can use the database port number, ″50000″.

Building the applet:

1. Compile `Applt.java` to produce the file `Applt.class` with this command:
   ```
   javac Applt.java
   ```
   Alternatively, you can use the Java `makefile` to build this program.
2. Ensure that your working directory is accessible by your web browser. If it is not, copy `Applt.class` and `Applt.html` into a directory that is accessible.
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`.
4. On your client machine, start your web browser and load `Applt.html`.

Related concepts:

- "Java applet considerations" on page 163

Related tasks:

- "Building JDBC applications" on page 158
- "Building JDBC routines" on page 158

- “Building SQLJ applets” on page 160

Related reference:

- “JDBC samples” on page 169

Related samples:

- “Applt.java -- A Java applet that use JDBC applet driver to access a database (JDBC)”

Building JDBC applications

`DbInfo` demonstrates a dynamic SQL Java application accessing a DB2 database server.

Procedure:

To build and run this application by commands entered at the command line:

1. Compile `DbInfo.java` to produce the file `DbInfo.class` with this command:
   ```
   javac DbInfo.java
   ```
2. Run the Java interpreter on the application with this command:
   ```
   java DbInfo
   ```

You can also use the Java `makefile` to build this program.

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
  ```

Related tasks:

- “Building JDBC applets” on page 157
- “Building JDBC routines” on page 158
- “Building SQLJ applications” on page 162

Related reference:

- “JDBC samples” on page 169

Related samples:

- “DbInfo.java -- How to get/set info in a database (JDBC)”

Building JDBC routines

DB2 provides sample programs demonstrating JDBC routines (stored procedures and user-defined functions) in the `samples/java/jdbc` directory on UNIX, and the `samples\java\jdbc` directory on Windows. Routines are compiled and stored on a server. When called by a client application, they access the server database and return information to the client application.

Procedure:

The following examples show you how to build routines comprising:
- stored procedures
- user-defined functions without SQL statements
- user-defined functions with SQL statements

Stored Procedures

`SpServer` demonstrates dynamic SQL PARAMETER STYLE JAVA stored procedures.

To build and run this program on the server from the command line:
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. Next, catalog the routines by running the `spcat` script on the server. Enter:
   ```
   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.
4. Then, 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.

User-defined functions without SQL statements

`UDFsrv` is a user-defined function library that does not contain SQL statements. DB2 provides both a JDBC client application, `UDFcli`, and an SQLJ client application, `UDFcli`, that can access the `UDFsrv` library.

To build and run the UDF program 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. To access the `UDFsrv` library, you can use either JDBC or SQLJ client applications. Both versions of the client program contain the `CREATE FUNCTION` SQL statement that you use to register the UDFs contained in `UDFsrv` with the database, and also contain SQL statements that make use of the UDFs, once they have been registered.

User-defined functions with SQL statements

`UDFsqlsv` is a user-defined function library that contains SQL statements. DB2 provides a JDBC client application, `UDFsqlcl`, to access the `UDFsqlsv` library.

To build and run the UDF program on the server from the command line:

1. Compile `UDFsqlsv.java` to produce the file `UDFsqlsv.class` with this command:
   ```
   javac UDFsqlsv.java
   ```
2. Copy `UDFsqlsv.class` to the `sqllib\function` directory on Windows operating systems, or to the `sqllib/function` directory on UNIX.
3. To access the `UDFsqlsv` library, use the client program, `UDFsqlcl`, which contains the `CREATE FUNCTION` SQL statement that you use to register the UDFs contained in `UDFsqlsv` with the database. The client program also contains SQL statements that make use of the UDFs, once they have been registered.

You can also use the Java makefile to build the above programs.

Related tasks:
- "Building JDBC applets" on page 157
- "Building JDBC applications" on page 158
- "Building SQLJ routines" on page 165

Related reference:
- "JDBC samples" on page 169

Related samples:
- "spcat -- To catalog SQLj stored procedures on UNIX"
- "SpClient.java -- Call a variety of types of stored procedures from SpServer.java (JDBC)"
- "SpCreate.db2 -- How to catalog the stored procedures contained in SpServer.java "
- "SpDrop.db2 -- How to uncatalog the stored procedures contained in SpServer.java"
- "SpServer.java -- Provide a variety of types of stored procedures to be called from (JDBC)"
- "UDFcli.java -- Call the UDFs in UDFsrv.java (JDBC)"
- "UDFCreate.db2 -- How to catalog the Java UDFs contained in UDFsrv.java "
- "UDFDrop.db2 -- How to uncatalog the Java UDFs contained in UDFsrv.java "
- "UDFsCreate.db2 -- How to catalog the UDFs contained in UDFsqlsv.java "
- "UDFsDrop.db2 -- How to uncatalog the UDFs contained in UDFsqlsv.java "
- "UDFsqlcl.java -- Call the UDFs in UDFsqlsv.java (JDBC)"
- "UDFsqlsv.java -- Provide UDFs to be called by UDFsqlcl.java (JDBC)"
- "UDFsrv.java -- Provide UDFs to be called by UDFcli.java (JDBC)"

Building SQLJ applets

The following steps show how to build the `Applt` sample that demonstrates an SQLJ applet accessing a DB2 database. 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.

Procedure:

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 Java Development Kit by entering the following command in the working directory of your client machine:

```
appletviewer Applt.html
```

Connecting with the IBM DB2 Driver for JDBC and SQLJ

To connect with the IBM DB2 Driver for JDBC and SQLJ, modify 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″.

Building the Applet

1. Build the applet with this command:
   ```
   bldsqlj Applt <userid> <password> <server_name> <port_number> <db_name>
   ```
   where all parameters except the program name can have default values, as explained in the build file.
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`.
4. On your client machine, start your web browser or Java applet viewer, and load `Applt.html`.

You can also use the Java `makefile` to build this program.

Related concepts:
- “Java applet considerations” on page 163

Related tasks:
- “Building JDBC applets” on page 157
- “Building SQLJ applications” on page 162
- “Building SQLJ routines” on page 165

Related reference:
- “SQLJ application and applet options for UNIX” on page 164
- “SQLJ application and applet options for Windows” on page 164
- “SQLJ samples” on page 174

Related samples:
- "bldsqlj.bat -- Builds a Java embedded SQL (SQLJ) application or applet on Windows"
- "Applt.sqlj -- An SQLJ applet that uses a JDBC applet driver to access a database (SQLj)"
- "bldsqlj -- Builds Java embedded SQL (SQLJ) applications and applets on UNIX"

Building SQLJ applications

The following steps show how to build the `TbMod` sample that demonstrates an SQLJ application accessing a DB2 database. 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.

Procedure:

To build `TbMod` with the build file, `bldsqlj` (UNIX) or `bldsqlj.bat` (Windows), enter this command:

```
bldsqlj TbMod <userid> <password> <server_name> <port_number> <db_name>
```

where all parameters except the program name can have default values, as explained in the build file.

Run the Java interpreter on the application with this command:

```
java TbMod
```

You can also use the Java `makefile` to build this program.

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
  ```

Related tasks:
- "Building JDBC applications" on page 158
- "Building SQLJ applets" on page 160
- "Building SQLJ routines" on page 165

Related reference:
- "SQLJ application and applet options for UNIX" on page 164

- “SQLJ application and applet options for Windows” on page 164
- “SQLJ samples” on page 174

Related samples:

- “bldsqlj.bat -- Builds a Java embedded SQL (SQLJ) application or applet on Windows”
- “TbMod.sqlj -- How to modify table data (SQLj)”
- “bldsqlj -- Builds Java embedded SQL (SQLJ) applications and applets on UNIX”

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 DB2 Driver for JDBC and SQLJ or for any SQLJ applet. This file is in the `sqllib\java` directory on Windows operating systems, and in the `sqllib/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.
- 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).

Related tasks:

- “Building JDBC applets” on page 157

- "Building SQLJ applets" on page 160

SQLJ application and applet options for UNIX

The SQLJ translator and customizer options are used in the `bldsqlj` build script on UNIX. These are the options DB2 recommends that you use to build SQLJ applications and applets on UNIX platforms.

sqlj The SQLJ translator (also compiles the program).

`"${progname}.sqlj"`
The SQLJ source file. The `progname=${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.

`"${progname}_SJProfile0"`
Specifies a serialized profile for the program.

Related tasks:

- "Building SQLJ applets" on page 160
- "Building SQLJ applications" on page 162

Related reference:

- "SQLJ routine options for UNIX" on page 166

Related samples:

- "bldsqlj -- Builds Java embedded SQL (SQLJ) applications and applets on UNIX"

SQLJ application and applet options for Windows

The following SQLJ translator and customizer options are used in the `bldsqlj.bat` batch file on Windows operating systems. These are the options DB2 recommends that you use to build SQLJ applications and applets on Windows.

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.

Related tasks:
- "Building SQLJ applets" on page 160
- "Building SQLJ applications" on page 162

Related reference:
- "SQLJ routine options for Windows" on page 167

Related samples:
- "bldsqlj.bat -- Builds a Java embedded SQL (SQLJ) application or applet on Windows"

Building SQLJ routines

DB2 provides sample programs demonstrating SQLJ routines (stored procedures and user-defined functions) in the `samples/java/sqlj` directory on UNIX, and the `samples\java\sqlj` directory on Windows. Routines are compiled and stored on a server. When called by a client application, they access the server database and return information to the client application.

In the same directory, DB2 also supplies the build file, `bldsqljs` (UNIX), or `bldsqljs.bat` (Windows), which contains commands to build routines.

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.

Procedure:

The following example shows you how to build a class file with stored procedures.

`SpServer` demonstrates PARAMETER STYLE JAVA stored procedures using the JDBC application driver to access a DB2 database.

To build this stored procedure class with the build file, `bldsqljs` (UNIX) or `bldsqljs.bat` (Windows):

1. Enter the following command:
   ```
   bldsqljs SpServer <userid> <password> <server_name> \
     <port_number> <db_name>
   ```
   where all parameters except the program name can have default values, as explained in the build file.
2. Next, catalog the routines by running the `spcat` script on the server. Enter:
   ```
   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. Then, 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. Build and run the `SpClient` client application to call the stored procedures. You can build `SpClient` with the application build file, `bldsqlj` (UNIX) or `bldsqlj.bat` (Windows).

You can also use the Java `makefile` to build the above programs.

Related tasks:

- "Building JDBC routines" on page 158
- "Building SQLJ applets" on page 160
- "Building SQLJ applications" on page 162

Related reference:

- "SQLJ samples" on page 174
- "SQLJ routine options for UNIX" on page 166
- "SQLJ routine options for Windows" on page 167

Related samples:

- "bldsqljs.bat -- Builds a Java embedded SQL (SQLJ) stored procedure on Windows"
- "SpClient.sqlj -- Call a variety of types of stored procedures from SpServer.sqlj (SQLj)"
- "SpCreate.db2 -- How to catalog the stored procedures contained in SpServer.sqlj "
- "SpDrop.db2 -- How to uncatalog the stored procedures contained in SpServer.sqlj"
- "SpServer.sqlj -- Provide a variety of types of stored procedures to be called from (SQLj)"
- "SpIterat.sqlj -- Iterator class file for SpServer.sqlj (SQLj)"
- "bldsqljs -- Builds Java embedded SQL (SQLJ) stored procedures on UNIX"
- "spcat -- To catalog SQLj stored procedures on UNIX"

SQLJ routine options for UNIX

The following SQLJ translator and customizer options are used in the `bldsqljs` build script on UNIX. These are the recommended options for building SQLJ routines (stored procedures and user-defined functions) on UNIX platforms.

Translator and customizer options for `bldsqljs`:

sqlj
The SQLJ translator (also compiles the program).

`"${progname}.sqlj"`
The SQLJ source file. The `progname=${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 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.

"${progname}_SJProfile0"
Specifies a serialized profile for the program.

Related tasks:

- "Building SQLJ routines" on page 165

Related reference:

- "SQLJ application and applet options for UNIX" on page 164

Related samples:

- "bldsqljs -- Builds Java embedded SQL (SQLJ) stored procedures on UNIX"

SQLJ routine options for Windows

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.

Related tasks:

- "Building SQLJ routines" on page 165

Related reference:

- "SQLJ application and applet options for Windows" on page 164

Related samples:

- "bldsqljs.bat -- Builds a Java embedded SQL (SQLJ) stored procedure on Windows"

Chapter 6. Java sample applications

The following topics contain information about the Java sample applications that are provided with DB2 Database for Linux, UNIX, and Windows.

- "JDBC samples"
- "SQLJ samples" on page 174
- "Java plug-in samples" on page 178
- "Java WebSphere samples" on page 179

JDBC samples

UNIX directory: `sqllib/samples/java/jdbc`.

Windows directory: `sqllib\samples\java\jdbc`.

JDBC samples include the following types of programs:

ADMIN_CMD routine samples
Samples that demonstrate the use of ADMIN_CMD stored procedure for administration tasks via SQL interface

Installation-image-level samples
Programs that deal with the installation image level of the database product.

Database-level samples
Programs that deal with database objects.

Table-level samples
Programs that deal with table objects.

Data type samples
Programs that deal with data types.

Applet samples
Samples that demonstrate Java applets.

Stored procedure samples
Samples that demonstrate stored procedures.

User-defined function samples
Samples that demonstrate user-defined functions.

Java bean samples
Samples that demonstrate Java bean classes.

Java GSS-API Plugin samples
Samples that demonstrate the JCC GCC_API plugin

Other samples
Samples that do not fall into any of the previous categories.

Table 22. JDBC sample program files

Type of sample	Sample program name	Program Description
ADMIN_CMD stored procedure samples	`AdmCmdAutoCfg.java`	How to autoconfigure a database using ADMIN_CMD routine.
	`AdmCmdContacts.java`	How to add, update and drop contacts and contactgroups using ADMIN_CMD routine.
	`AdmCmdOnlineBackup.java`	How to perform online backup using ADMIN_CMD routine.
	`AdmCmdUpdateCfg.java`	How to update and reset the Database and Database Manager configuration parameters using ADMIN_CMD routine.
	`AdmCmdExport.java`	How to export data using ADMIN_CMD routine.
	`AdmCmdImport.java`	How to import data using ADMIN_CMD routine.
	`AdmCmdQuiesce.java`	How to quiesce tablespace and database using ADMIN_CMD routine.
	`AdmCmdDescribe.java`	How to describe table and indices using ADMIN_CMD routine.
Installation-image-level samples	`IlInfo.java`	How to get and set installation level information.

Table 22. JDBC sample program files (continued)

Type of sample	Sample program name	Program Description
Database-level samples	`DbAuth.java`	How to grant/display/revoke authorities at the database level.
	`DbConn.java`	How to connect and disconnect from a database.
	`DbInfo.java`	How to get and set information at a database level.
	`DbMCon.java`	How to connect and disconnect from multiple databases.
	`DbNative.java`	How to translate a statement that contains an ODBC escape clause to a data source specific format.
	`DbRsHold.java`	How to use result set cursor holdability in the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows and the IBM DB2 Driver for JDBC and SQLJ. To compile this sample, you need Java Developer Kit 1.4 or above. To run this sample, you need Java Runtime Environment 1.4 or above.
	`DbSeq.java`	How to create, alter and drop a sequence in a database.
	`DbUse.java`	How to use database objects.
	`GetDBCfgParams.java`	How to get database configuration parameters.
	`GetDBMCfgParams.java`	How to get database manager configuration parameters.
	`GetLogs.java`	How to get customer view of diagnostic log file entries.

Table 22. JDBC sample program files (continued)

Type of sample	Sample program name	Program Description
Table-level samples	GetMessage.java	How to get error message in the required locale with token replacement.
	LargeRid.java	How to enable Large RIDs support on both new tables / tablespaces and existing tables/tablespaces.
	SetIntegrity.java	How to perform online SET INTEGRITY on a table.
	TbAST.java	How to use staging table for updating deferred AST.
	TbCompress.java	How to create tables with null and default value compression option.
	TbConstr.java	How to work with table constraints.
	TbCreate.java	How to create, alter and drop tables.
	TbGenCol.java	How to use generated columns.
	TbIdent.java	How to use Identity Columns.
	TbInfo.java	How to get and set information at a table level.
	TbInTrig.java	How to use an 'INSTEAD OF' trigger on a view.
	TbMerge.java	How to use the MERGE statement.
	TbMod.java	How to modify information in a table.
	TbOnlineInx.java	How to create and reorg indexes on a table.
	TbPriv.java	How to grant/display/revoke table level privileges.
	TbRead.java	How to read information in a table.
	TbRowcompress.java	How to perform row compression on a table
	TbSel.java	How to select from each of: insert, update, delete.
	TbTemp.java	How to use Declared Temporary Tables.
	TbTrig.java	How to use a trigger on a table.
	TbUMQT.java	How to use user materialzed query tables (summary tables).
	TbUnion.java	How to insert through a UNION ALL view.
Data types	DtInfo.java	How to get information about data types.
	DtLob.java	How to read and write LOB data.
	DtUdt.java	How to create, use, and drop user-defined distinct types.
Applet samples	Applt.java	How to use applets.

Table 22. JDBC sample program files (continued)

Type of sample	Sample program name	Program Description
Stored procedure samples	`spcat`	Stored procedure catalog script for the spserver program. This script calls SpDrop.db2 and SpCreate.db2.
	`SpCreate.db2`	CLP script to issue CREATE PROCEDURE statements.
	`SpDrop.db2`	CLP script to drop stored procedures from the catalog.
	`SpClient.java`	Client program used to call the server functions declared in SpServer.java.
	`SpServer.java`	Stored procedure functions built and run on the server.
User-defined function samples	`UDFcli.java`	Client application which calls the user-defined function library UDFsrv.
	`UDFsrv.java`	User-defined functions called by UDFcli.java.
	`udfcat`	UDF catalog script for the UDFsrv program. This script calls UDFDrop.db2 and UDFCreate.db2.
	`UDFDrop.db2`	CLP script to drop UDFs from the catalog.
	`UDFCreate.db2`	CLP script to issue CREATE PROCEDURE statements.
	`UDFjcli.java`	Client application which calls the user-defined function library UDFjsrv.
	`UDFjsrv.java`	User-defined functions called by UDFjcli.java.
	`udfjcat`	UDF catalog script for the UDFjsrv program. This script calls UDFjDrop.db2 and UDFjCreate.db2.
	`UDFjDrop.db2`	CLP script to drop UDFs from the catalog.
	`UDFjCreate.db2`	CLP script to issue CREATE PROCEDURE statements.
	`UDFsCreate.db2`	How to catalog the UDFs contained in UDFsqlsv.java
	`UDFsDrop.db2`	How to uncatalog the UDFs contained in UDFsqlsv.java
	`UDFsqlcl.java`	Call the UDFs in UDFsqlsv.java
	`UDFsqlsv.java`	User-Defined Functions with SQL statements called by UDFsqlcl.java
Java bean samples	`CreateEmployee.java`	How to create an employee record.
	`GeneratePayroll.java`	How to generate payroll reports by department.

Table 22. JDBC sample program files (continued)

Type of sample	Sample program name	Program Description
Java GSS-API Plugin samples	`JCCKerberosPlugin.java`	Implement a GSS-API Plugin that does Kerberos authentication using IBM DB2 Universal Driver.
	`JCCKerberosPluginTest.java`	Use JCCKerberosPlugin to get a DB2 Connection using IBM DB2 Universal Driver.
	`JCCSimpleGSSPlugin.java`	Implement a GSS-API Plugin that does userid and password checking using IBM DB2 Universal Driver.
	`JCCSimpleGSSContext.java`	Implement a GSSContext to be used by JCCSimpleGSSPlugin
	`JCCSimpleGSSCredential.java`	Implement a GSSCredential to be used by JCCSimpleGSSPlugin
	`JCCSimpleGSSException.java`	Implement a GSSException to be used by JCCSimpleGSSPlugin
	`JCCSimpleGSSName.java`	Implement a GSSName to be used by JCCSimpleGSSPlugin
	`JCCSimpleGSSPluginTest.java`	Use JCCSimpleGSSPlugin to get a DB2 Connection using IBM DB2 Universal Driver.
Other samples	`Util.java`	Utilities for JDBC sample programs.

Related concepts:
- "Java sample programs" in *Samples Topics*
- "Sample files" in *Samples Topics*

Related reference:
- "Java plug-in samples" on page 178
- "Java WebSphere samples" on page 179
- "SQLJ samples" on page 174

SQLJ samples

UNIX directory: `sqllib/samples/java/sqlj`.

Windows directory: `sqllib\samples\java\sqlj`.

SQLJ samples include the following types of programs:

Database-level samples
Programs that deal with database objects.

Table-level samples
Programs that deal with table objects.

Data type samples
Programs that deal with data types.

Applet samples
Samples that demonstrate Java applets.

Stored procedure samples
Samples that demonstrate stored procedures.

User-defined function samples
Samples that demonstrate user-defined functions.

Java bean samples
Samples that demonstrate Java bean classes.

Data source samples
Samples that demonstrate data sources.

Other samples
Samples that do not fall into any of the previous categories.

Table 23. SQLJ sample program files

Type of sample	Sample program name	Program Description
Database-level sample	`DbAuth.sqlj`	How to grant/display/revoke authorities at the database level.
	`DbConn.sqlj`	How to connect and disconnect from a database.
	`DbMCon.sqlj`	How to connect and disconnect from multiple databases.
	`DbUse.sqlj`	How to use database objects.

Table 23. SQLJ sample program files (continued)

Type of sample	Sample program name	Program Description
Table-level sample	`LargeRid.sqlj`	How to enable Large RIDs support on both new tables/ tablespaces and existing tables/tablespaces.
	`SetIntegrity.sqlj`	How to perform online SET INTEGRITY on a table.
	`TbAST.sqlj`	How to use staging table for updating deferred AST.
	`TbCompress.sqlj`	How to create tables with null and default value compression option.
	`TbConstr.sqlj`	How to work with table constraints.
	`TbCreate.sqlj`	How to create, alter and drop tables.
	`TbIdent.sqlj`	How to use identity columns.
	`TbInfo.sqlj`	How to get and set information at a table level.
	`TbMod.sqlj`	How to modify information in a table.
	`TbOnlineInx.sqlj`	How to create and reorg indexes on a table.
	`TbPriv.sqlj`	How to grant/display/revoke table level privileges.
	`TbRowcompress.sqlj`	How to perform row compression on a table
	`TbRunstats.sqlj`	How to perform runstats on a table.
	`TbRead.sqlj`	How to read information in a table.
	`TbSel.sqlj`	How to select from each of: insert, update, delete.
	`TbTrig.sqlj`	How to use a trigger on a table.
	`TbUMQT.sqlj`	How to use user materialzed query tables (summary tables).
Data type sample	`DtUdt.sqlj`	How to create, use, and drop user-defined distinct types.
Applet sample	`Applt.sqlj`	How to use applets.
Stored procedure sample	`spcat`	Stored procedure catalog script for the SpServer program. This script calls SpDrop.db2 and SpCreate.db2.
	`SpCreate.db2`	CLP script to issue CREATE PROCEDURE statements.
	`SpDrop.db2`	CLP script to drop stored procedures from the catalog.
	`SpClient.sqlj`	Client program used to call the server functions declared in SpServer.sqlj.
	`SpServer.sqlj`	Stored procedure functions built and run on the server.
	`SpIterat.sqlj`	Iterator class file for SpServer.sqlj.

Table 23. SQLJ sample program files (continued)

Type of sample	Sample program name	Program Description
User-defined function sample	`UDFcli.sqlj`	Client application which calls the user-defined function library UDFsrv.
	`UDFsrv.java`	User-defined functions called by UDFcli.
	`udfcat`	UDF catalog script for the UDFsrv program. This script calls UDFDrop.db2 and UDFCreate.db2.
	`UDFDrop.db2`	CLP script to drop UDFs from the catalog.
	`UDFCreate.db2`	CLP script to issue CREATE PROCEDURE statements.
	`UDFjcli.sqlj`	Client application which calls the user-defined function library UDFjsrv.
	`UDFjsrv.java`	User-defined functions called by UDFjcli.
	`udfjcat`	UDF catalog script for the UDFjsrv program. This script calls UDFjDrop.db2 and UDFjCreate.db2.
	`UDFjDrop.db2`	CLP script to drop UDFs from the catalog.
	`UDFjCreate.db2`	CLP script to issue CREATE PROCEDURE statements.
Java bean sample	`CreateEmployee.sqlj`	How to create an employee record.
	`GeneratePayroll.sqlj`	How to generate payroll reports by department.
Data source sample	`Batch1Demo.sqlj`	SQLJ batching -- How SQLJ batching works.
	`Batch2Demo.sqlj`	SQLJ batching - Association of ExecutionContext with BatchContext.
	`Batch3Demo.sqlj`	SQLJ Batching - When do we need to implicitly execute a batch.
	`BlobClobDemo.sqlj`	How to access Blob or Clob fields in DB2 tables.
	`createRegisterDS.java`	Create and Register DataSources as specified by the DataSource property files.
	`CreateDemoSchema.sqlj`	This program creates the schema for the DataSource Demo programs.
	`DbConnDataSource.sqlj`	How to connect to a database using DataSource with the IBM DB2 Driver for JDBC and SQLJ.
	`DbConMDataSources.sqlj`	How to connect to a database using Multiple DataSources with the IBM DB2 Driver for JDBC and SQLJ.
	`ScrollIterDemo.sqlj`	How to use Named and Positional Scrollable Iterators in SQLJ.
Other sample	`Util.sqlj`	Utilities for SQLJ sample programs.

Related concepts:

- "Java sample programs" in *Samples Topics*
- "Sample files" in *Samples Topics*

Related reference:

- "Java plug-in samples" on page 178
- "Java WebSphere samples" on page 179
- "JDBC samples" on page 169

Java plug-in samples

UNIX directory: `sqllib/samples/java/plugin`.

Windows directory: `sqllib\samples\java\plugin`.

Table 24. Java Control Center plug-in sample files

Sample program name	Program description
`Example1.java`	How to add a new toolbar button to the Control Center toolbar.
`Example2.java`	How to add new menu actions to Control Center Database objects.
`Example3.java`	How to add new objects under Database objects in the Control Center tree.
`Example3Child.java`	How to add plug-in objects under Database objects in the Control Center tree.
`Example3Folder.java`	How to add new objects under Database objects in the Control Center tree.

Related concepts:

- "Compiling and running the example plugins" in *Administration Guide: Implementation*
- "Introducing the plug-in architecture for the Control Center" in *Administration Guide: Implementation*
- "Writing plugins as Control Center extensions" in *Administration Guide: Implementation*
- "Java sample programs" in *Samples Topics*
- "Sample files" in *Samples Topics*

Related tasks:

- "Creating a plugin that adds a toolbar button" in *Administration Guide: Implementation*
- "Setting attributes for a plugin tree object" in *Administration Guide: Implementation*

Related reference:

- "Java WebSphere samples" on page 179
- "JDBC samples" on page 169
- "SQLJ samples" on page 174

Java WebSphere samples

UNIX directory: `sqllib/samples/java/Websphere`.

Windows directory: `sqllib\samples\java\Websphere`.

Table 25. Java WebSphere sample files

Sample program name	Program description
`AccessEmployee.ear`	This Enterprise ARchive (.EAR) file consists of four modules containing 32 different `.class`, `.JSP` and `.HTML` files. This EAR file, easily deployed using IBM WebSphere Application Server, demonstrates how Java clients can interact with Enterprise Java Beans (EJBs) to access data stored in DB2.

Related concepts:

- "Java sample programs" in *Samples Topics*
- "Sample files" in *Samples Topics*

Related reference:

- "Java plug-in samples" on page 178
- "JDBC samples" on page 169
- "SQLJ samples" on page 174

Chapter 7. Diagnosing JDBC and SQLJ problems

The topics that follow contain information on diagnosing JDBC and SQLJ problems.

- "Diagnosing JDBC and SQLJ problems under the IBM DB2 Driver for JDBC and SQLJ"
- "Diagnosing JDBC and SQLJ problems under the DB2 JDBC Type 2 Driver" on page 192

Diagnosing JDBC and SQLJ problems under the IBM DB2 Driver for JDBC and SQLJ

The topics that follow contain information on diagnosing JDBC and SQLJ problems under the IBM DB2 Driver for JDBC and SQLJ.

- "JDBC and SQLJ problem diagnosis with the IBM DB2 Driver for JDBC and SQLJ"
- "Example of using configuration properties to start a JDBC trace" on page 184
- "Example of a trace program under the IBM DB2 Driver for JDBC and SQLJ" on page 184
- "System monitoring for the IBM DB2 Driver for JDBC and SQLJ" on page 189

JDBC and SQLJ problem diagnosis with the IBM DB2 Driver for JDBC and SQLJ

To obtain data for diagnosing SQLJ or JDBC problems with the IBM DB2 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.

If your application connects to a DB2 for z/OS server, a number of stored procedures need to be installed on that server before you can collect trace data. Those stored procedures are also used for some `DatabaseMetaData` calls. The stored procedures are:

- SQLCOLPRIVILEGES
- SQLCOLUMNS
- SQLFOREIGNKEYS
- SQLGETTYPEINFO
- SQLPRIMARYKEYS
- SQLPROCEDURECOLS
- SQLPROCEDURES
- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS
- SQLCAMESSAGE

For DB2 UDB for OS/390 and z/OS, Version 7, the stored procedures are shipped in a PTF. The PTF is orderable through normal service channels using the following PTF numbers:

Table 26. PTFs for DB2 for z/OS

DB2 for z/OS Version	PTF number
Version 7	UQ72083

Ask your DB2 for z/OS system administrator whether these stored procedures are installed.

Collecting JDBC trace data:

Use one of the following procedures to start the trace:

Procedure 1: For IBM DB2 Driver for JDBC and SQLJ type 4 connectivity or IBM DB2 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 DB2 Driver for JDBC and SQLJ configuration properties file.

Procedure 2:

1. If you use the `DataSource` interface to connect to a data source, 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 DB2 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 DB2 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:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose" +
 ":traceFile=/u/db2p/jcctrace;" +
 "traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_DRDA_FLOWS + ";";
```

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 DB2 Driver for JDBC and SQLJ, see Example of a trace program under the IBM DB2 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 DB2 Driver for JDBC and SQLJ.

Run the `db2sqljprint` utility on the customized serialized profile for the connection in which the error occurs.

Related concepts:
- "Example of a trace program under the IBM DB2 Driver for JDBC and SQLJ" on page 184
- "Example of using configuration properties to start a JDBC trace" on page 184

Related reference:
- "db2sqljbind - SQLJ profile binder" on page 361
- "db2sqljcustomize - SQLJ profile customizer" on page 351
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

Example of using configuration properties to start a JDBC trace

Suppose that you want to collect trace data for a program named Test.java, which uses IBM DB2 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.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

Related concepts:
- "IBM DB2 Driver for JDBC and SQLJ configuration properties customization" on page 11
- "JDBC and SQLJ problem diagnosis with the IBM DB2 Driver for JDBC and SQLJ" on page 181

Example of a trace program under the IBM DB2 Driver for JDBC and SQLJ

The following example shows a class for establishing a connection using IBM DB2 Driver for JDBC and SQLJ type 4 connectivity and gathering and displaying trace data under the IBM DB2 Driver for JDBC and SQLJ. The class includes a method for the `DriverManager` interface and a method for the `DataSource` interface.

```
public class TraceExample
{

  public static void main(String[] args)
  {
    sampleConnectUsingSimpleDataSource();
    sampleConnectWithURLUsingDriverManager();
  }

  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);
```

Figure 60. Example of tracing under the IBM DB2 Driver for JDBC and SQLJ (Part 1 of 5)

```
      // 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();
      }
```

Figure 60. Example of tracing under the IBM DB2 Driver for JDBC and SQLJ (Part 2 of 5)

```
     catch(java.sql.SQLException e) {
       printWriter.println("[TraceExample] " +
        "Trapped the following java.sql.SQLException while trying to roll back:");
       com.ibm.db2.jcc.DB2ExceptionFormatter.printTrace(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.jcc.DB2ExceptionFormatter.printTrace(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.jcc.DB2ExceptionFormatter.printTrace(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.jcc.DB2ExceptionFormatter.printTrace(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.jcc.DB2ExceptionFormatter.printTrace(e, printWriter,
       "[TraceExample]");
    }
  }
```

Figure 60. Example of tracing under the IBM DB2 Driver for JDBC and SQLJ (Part 3 of 5)

```
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 DB2 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"
  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");
```

Figure 60. Example of tracing under the IBM DB2 Driver for JDBC and SQLJ (Part 4 of 5)

```
    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();
      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();
    }
  }
}
```

Figure 60. Example of tracing under the IBM DB2 Driver for JDBC and SQLJ (Part 5 of 5)

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 30
- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

System monitoring for the IBM DB2 Driver for JDBC and SQLJ

To assist you in monitoring the performance of your applications with the IBM DB2 Driver for JDBC and SQLJ, the driver provides two methods to collect the following information about a connection:

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 DB2 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 DB2 server elapsed times that were collected while system monitoring was enabled, in microseconds.

  Currently, DB2 Database for Linux, UNIX, and Windows servers do not support this function.

Application time
: The sum of the application, JDBC driver, network I/O, and DB2 server elapsed times, in milliseconds.

The two methods are:

- The `DB2SystemMonitor` interface
- The TRACE_SYSTEM_MONITOR trace level

DB2SystemMonitor method:

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.

```
import java.sql.*;
import com.ibm.db2.jcc.*;
public class TestSystemMonitor
{
  public static void main(String[] args)
  {
    String url = "jdbc:db2://sysmvs1.svl.ibm.com:5021/san_jose";
    String user="db2adm";
    String password="db2adm";
    try
    {
      // Load the IBM DB2 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 DB2 Driver for JDBC and SQLJ
      Connection conn = DriverManager.getConnection (url,user,password);
      // Commit changes manually
      conn.setAutoCommit(false);
      System.out.println("**** Created a JDBC connection to the data source");
      DB2SystemMonitor systemMonitor =                                1
        ((DB2Connection)conn).getDB2SystemMonitor();
      systemMonitor.enable(true);                                     2
      systemMonitor.start(DB2SystemMonitor.RESET_TIMES);              3
      Statement stmt = conn.createStatement();
      int numUpd = stmt.executeUpdate(
        "UPDATE EMPLOYEE SET PHONENO='4657' WHERE EMPNO='000010'");
      systemMonitor.stop();                                           4
      System.out.println("Server elapsed time (microseconds)="
        + systemMonitor.getServerTimeMicros());                       5
      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();
    }
    // Handle errors
    catch(ClassNotFoundException e)
    {
      System.err.println("Unable to load IBM DB2 Driver for JDBC and SQLJ, " + e);
    }
    catch(SQLException e)
    {
      System.out.println("SQLException: " + e);
      e.printStackTrace();
    }
  }
}
```

Figure 61. Example of using DB2SystemMonitor methods to collect system monitoring data

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:

```
String url = "jdbc:db2://sysmvs1.stl.ibm.com:5021/san_jose" +
 ":traceFile=/u/db2p/jcctrace;" +
 "traceLevel=" + com.ibm.db2.jcc.DB2BaseDataSource.TRACE_SYSTEM_MONITOR + ";";
```

The trace records with system monitor information look similar to this:

```
[ibm][db2][jcc][SystemMonitor:start]
...
[ibm][db2][jcc][SystemMonitor:stop] core: 565.67ms | network: 211.695ms | server: 207.771ms
```

Related concepts:

- "Example of a trace program under the IBM DB2 Driver for JDBC and SQLJ" on page 184
- "Example of using configuration properties to start a JDBC trace" on page 184

Related reference:

- "Summary of IBM DB2 Driver for JDBC and SQLJ extensions to JDBC" on page 301

Diagnosing JDBC and SQLJ problems under the DB2 JDBC Type 2 Driver

The topics that follow contain information on diagnosing JDBC and SQLJ problems under the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (DB2 JDBC Type 2 Driver).

- "CLI/ODBC/JDBC trace facility"
- "CLI and JDBC trace files" on page 197

CLI/ODBC/JDBC trace facility

This topic discusses the following subjects:

- "DB2 CLI and DB2 JDBC trace configuration" on page 193
- "DB2 CLI trace options and the db2cli.ini file" on page 193
- "DB2 JDBC trace options and the db2cli.ini file" on page 194
- "DB2 CLI driver trace versus ODBC driver manager trace" on page 196
- "DB2 CLI driver, DB2 JDBC Type 2 Driver, and DB2 traces" on page 196
- "DB2 CLI and DB2 JDBC traces and CLI or Java stored procedures" on page 196

The DB2 CLI and the DB2 JDBC Type 2 Driver for Linux, UNIX, and Windows offer comprehensive tracing facilities. By default, these facilities are disabled and use no additional computing resources. When enabled, the trace facilities generate one or more text log files whenever an application accesses the appropriate driver (DB2 CLI or DB2 JDBC Type 2 Driver). These log files provide detailed information about:

- the order in which CLI or JDBC functions were called by the application
- the contents of input and output parameters passed to and received from CLI or JDBC functions
- the return codes and any error or warning messages generated by CLI or JDBC functions

Note: This trace facility does not apply to the DB2 Universal JDBC Driver.

DB2 CLI and DB2 JDBC trace file analysis can benefit application developers in a number of ways. First, subtle program logic and parameter initialization errors are often evident in the traces. Second, DB2 CLI and DB2 JDBC traces might suggest ways of better tuning an application or the databases it accesses. For example, if a DB2 CLI trace shows a table being queried many times on a particular set of attributes, an index corresponding to those attributes might be created on the table to improve application performance. Finally, analysis of DB2 CLI and DB2 JDBC trace files can help application developers understand how a third party application or interface is behaving.

DB2 CLI and DB2 JDBC trace configuration:

The configuration parameters for both DB2 CLI and DB2 JDBC traces facilities are read from the DB2 CLI configuration file `db2cli.ini`. By default, this file is located in the `\sqllib` path on the Windows platform and the `/sqllib/cfg` path on UNIX platforms. You can override the default path by setting the DB2CLIINIPATH environment variable. On the Windows platform, an additional `db2cli.ini` file can be found in the user's profile (or home) directory if there are any user-defined data sources defined using the ODBC Driver Manager. This `db2cli.ini` file will override the default file.

To view the current `db2cli.ini` trace configuration parameters from the command line processor, issue the following command:

```
db2 GET CLI CFG FOR SECTION COMMON
```

There are three ways to modify the `db2cli.ini` file to configure the DB2 CLI and DB2 JDBC trace facilities:

- use the DB2 Configuration Assistant if it is available
- manually edit the `db2cli.ini` file using a text editor
- issue the UPDATE CLI CFG command from the command line processor

For example, the following command issued from the command line processor updates the `db2cli.ini` file and enables the JDBC tracing facility:

```
db2 UPDATE CLI CFG FOR SECTION COMMON USING jdbctrace 1
```

Notes:

1. Typically the DB2 CLI and DB2 JDBC trace configuration options are only read from the `db2cli.ini` configuration file at the time an application is initialized. However, a special `db2cli.ini` trace option, TraceRefreshInterval, can be used to indicate an interval at which specific DB2 CLI trace options are reread from the `db2cli.ini` file.
2. The DB2 CLI tracing facility can also be configured programmatically by setting the SQL_ATTR_TRACE environment attribute. This setting will override the settings contained in the `db2cli.ini` file.

Important: Disable the DB2 CLI and DB2 JDBC trace facilities when they are not needed. Unnecessary tracing can reduce application performance and generate unwanted trace log files. DB2 does not delete any generated trace files and will append new trace information to any existing trace files.

DB2 CLI Trace options and the db2cli.ini file:

When an application using the DB2 CLI driver begins execution, the driver checks for trace facility options in the [COMMON] section of the db2cli.ini file. These trace options are specific trace keywords that are set to certain values in the `db2cli.ini` file under the [COMMON] section.

Note: Because DB2 CLI trace keywords appear in the [COMMON] section of the `db2cli.ini` file, their values apply to all database connections through the DB2 CLI driver.

The DB2 CLI trace keywords that can be defined are:
- Trace
- TraceComm
- TraceErrImmediateTraceErrImmediate
- TraceFileName
- TraceFlush
- TraceFlushOnErrorTraceFlushOnError
- TraceLocks
- TracePathName
- TracePIDList
- TracePIDTID
- TraceRefreshInterval
- TraceStmtOnly
- TraceTime
- TraceTimeStamp

Note: DB2 CLI trace keywords are only read from the `db2cli.ini` file once at application initialization time unless the TraceRefreshInterval keyword is set. If this keyword is set, the Trace and TracePIDList keywords are reread from the `db2cli.ini` file at the specified interval and applied, as appropriate, to the currently executing application.

An example `db2cli.ini` file trace configuration using these DB2 CLI keywords and values is:

```
[COMMON]
trace=1
TraceFileName=\temp\clitrace.txt
TraceFlush=1
```

Notes:
1. CLI trace keywords are NOT case sensitive. However, path and file name keyword values might be case-sensitive on some operating systems (such as UNIX).
2. If either a DB2 CLI trace keyword or its associated value in the `db2cli.ini` file is invalid, the DB2 CLI trace facility will ignore it and use the default value for that trace keyword instead.

DB2 JDBC Trace options and the db2cli.ini file:

When an application using the DB2 JDBC Type 2 Driver begins execution, the driver also checks for trace facility options in the db2cli.ini file. As with the DB2 CLI trace options, DB2 JDBC trace options are specified as keyword/value pairs located under the [COMMON] section of the `db2cli.ini` file.

Note: Because DB2 JDBC trace keywords appear in the [COMMON] section of the `db2cli.ini` file, their values apply to all database connections through the DB2 JDBC Type 2 Driver.

The DB2 JDBC trace keywords that can be defined are:
- JDBCTrace
- JDBCTracePathName
- JDBCTraceFlush

JDBCTrace = 0 | 1

The JDBCTrace keyword controls whether or not other DB2 JDBC tracing keywords have any effect on program execution. Setting JDBCTrace to its default value of 0 disables the DB2 JDBC trace facility. Setting JDBCTrace to 1 enables it.

By itself, the JDBCTrace keyword has little effect and produces no trace output unless the JDBCTracePathName keyword is also specified.

JDBCTracePathName = <fully_qualified_trace_path_name>

The value of JDBCTracePathName is the fully qualified path of the directory to which all DB2 JDBC trace information is written. The DB2 JDBC trace facility attempts to generate a new trace log file each time a JDBC application is executed using the DB2 JDBC Type 2 Driver. If the application is multithreaded, a separate trace log file will be generated for each thread. A concatenation of the application process ID, the thread sequence number, and a thread-identifying string are automatically used to name trace log files. There is no default path name to which DB2 JDBC trace output log files are written.

JDBCTraceFlush = 0 | 1

The JDBCTraceFlush keyword specifies how often trace information is written to the DB2 JDBC trace log file. By default, JDBCTraceFlush is set to 0 and each DB2 JDBC trace log file is kept open until the traced application or thread terminates normally. If the application terminates abnormally, some trace information that was not written to the trace log file might be lost.

To ensure the integrity and completeness of the trace information written to the DB2 JDBC trace log file, the JDBCTraceFlush keyword can be set to 1. After each trace entry has been written to the trace log file, the DB2 JDBC driver closes the file and then reopens it, appending new trace entries to the end of the file. This guarantees that no trace information will be lost.

Note: *Each DB2 JDBC log file close and reopen operation incurs significant input/output overhead and can reduce application performance considerably.*

An example `db2cli.ini` file trace configuration using these DB2 JDBC keywords and values is:

```
[COMMON]
jdbctrace=1
JdbcTracePathName=\temp\jdbctrace\
JDBCTraceFlush=1
```

Notes:

1. JDBC trace keywords are NOT case sensitive. However, path and file name keyword values might be case-sensitive on some operating systems (such as UNIX).
2. If either a DB2 JDBC trace keyword or its associated value in the `db2cli.ini` file is invalid, the DB2 JDBC trace facility will ignore it and use the default value for that trace keyword instead.
3. Enabling DB2 JDBC tracing does not enable DB2 CLI tracing. The DB2 JDBC Type 2 Driver depends on the DB2 CLI driver to access the database. Consequently, Java developers might also want to enable DB2 CLI tracing for additional information on how their applications interact with the database through the various software layers. DB2 JDBC and DB2 CLI trace options are

independent of each other and can be specified together in any order under the [COMMON] section of the `db2cli.ini` file.

DB2 CLI Driver trace versus ODBC driver manager trace:

It is important to understand the difference between an ODBC driver manager trace and a DB2 CLI driver trace. An ODBC driver manager trace shows the ODBC function calls made by an ODBC application to the ODBC driver manager. In contrast, a DB2 CLI driver trace shows the function calls made by the ODBC driver manager to the DB2 CLI driver *on behalf of the application*.

An ODBC driver manager might forward some function calls directly from the application to the DB2 CLI driver. However, the ODBC driver manager might also delay or avoid forwarding some function calls to the driver. The ODBC driver manager might also modify application function arguments or map application functions to other functions before forwarding the call on to the DB2 CLI driver.

Reasons for application function call intervention by the ODBC driver manager include:

- Applications written using ODBC 2.0 functions that have been deprecated in ODBC 3.0 will have the old functions mapped to new functions.
- ODBC 2.0 function arguments deprecated in ODBC 3.0 will be mapped to equivalent ODBC 3.0 arguments.
- The Microsoft® cursor library will map calls such as `SQLExtendedFetch()` to multiple calls to `SQLFetch()` and other supporting functions to achieve the same end result.
- ODBC driver manager connection pooling will usually defer `SQLDisconnect()` requests (or avoid them altogether if the connection gets reused).

For these and other reasons, application developers might find an ODBC driver manager trace to be a useful complement to the DB2 CLI driver trace.

For more information on capturing and interpreting ODBC driver manager traces, refer to the ODBC driver manager documentation. On the Windows platforms, refer to the Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference, also available online at: http://www.msdn.microsoft.com/.

DB2 CLI Driver, DB2 JDBC Type 2 Driver, and DB2 traces:

Internally, the DB2 JDBC Type 2 Driver makes use of the DB2 CLI driver for database access. For example, the Java getConnection() method is internally mapped by the DB2 JDBC Type 2 Driver to the DB2 CLI `SQLConnect()` function. As a result, Java developers might find a DB2 CLI trace to be a useful complement to the DB2 JDBC trace.

The DB2 CLI driver makes use of many internal and DB2 specific functions to do its work. These internal and DB2 specific function calls are logged in the DB2 trace. Application developers will not find DB2 traces useful, as they are only meant to assist IBM Service in problem determination and resolution.

DB2 CLI and DB2 JDBC traces and CLI or Java stored procedures:

On all workstation platforms, the DB2 CLI and DB2 JDBC trace facilities can be used to trace DB2 CLI and DB2 JDBC stored procedures.

Most of the DB2 CLI and DB2 JDBC trace information and instructions given in earlier sections is generic and applies to both applications and stored procedures equally. However, unlike applications which are clients of a database server (and typically execute on a machine separate from the database server), stored procedures execute at the database server. Therefore, the following additional steps must be taken when tracing DB2 CLI or DB2 JDBC stored procedures:

- Ensure the trace keyword options are specified in the `db2cli.ini` file located at the DB2 server.
- If the TraceRefreshInterval keyword is not set to a positive, non-zero value, ensure all keywords are configured correctly prior to database startup time (that is, when the db2start command is issued). Changing trace settings while the database server is running can cause unpredictable results. For example, if the TracePathName is changed while the server is running, then the next time a stored procedure is executed, some trace files might be written to the new path, while others are written to the original path. To ensure consistency, restart the server any time a trace keyword other than Trace or TracePIDList is modified.

Related concepts:

- "CLI and JDBC trace files" on page 197
- "db2cli.ini initialization file" in *Call Level Interface Guide and Reference, Volume 1*

Related reference:

- "CLI/ODBC configuration keywords listing by category" in *Call Level Interface Guide and Reference, Volume 1*
- "db2trc - Trace command" in *Command Reference*
- "GET CLI CONFIGURATION command" in *Command Reference*
- "Miscellaneous variables" in *Performance Guide*
- "SQLSetEnvAttr function (CLI) - Set environment attribute" in *Call Level Interface Guide and Reference, Volume 2*
- "UPDATE CLI CONFIGURATION command" in *Command Reference*

CLI and JDBC trace files

Applications that access the DB2 CLI and DB2 JDBC drivers can make use of the DB2 CLI and DB2 JDBC trace facilities. These utilities record all function calls made by the DB2 CLI or DB2 JDBC drivers to a log file which is useful for problem determination. This topic discusses how to access and interpret these log files generated by the tracing facilities:

- "CLI and JDBC trace file location"
- "CLI trace file interpretation" on page 199
- "JDBC trace file interpretation" on page 203

CLI and JDBC trace file location:

If the TraceFileName keyword was used in the `db2cli.ini` file to specify a fully qualified file name, then the DB2 CLI trace log file will be in the location specified. If a relative file name was specified for the DB2 CLI trace log file name, the location of that file will depend on what the operating system considers to be the current path of the application.

Note: If the user executing the application does not have sufficient authority to write to the trace log file in the specified path, no file will be generated and no warning or error is given.

If either or both of the TracePathName and JDBCTracePathName keywords were used in the `db2cli.ini` file to specify fully qualified directories, then the DB2 CLI and DB2 JDBC trace log files will be in the location specified. If a relative directory name was specified for either or both trace directories, the operating system will determine its location based on what it considers to be the current path of the application.

Note: If the user executing the application does not have sufficient authority to write trace files in the specified path, no file will be generated and no warning or error is given. If the specified trace path does not exist, it will not be created.

The DB2 CLI and DB2 JDBC trace facilities automatically use the application's process ID and thread sequence number to name the trace log files when the TracePathName and JDBCTracePathName keywords have been set. For example, a DB2 CLI trace of an application with three threads might generate the following DB2 CLI trace log files: `100390.0`, `100390.1`, `100390.2`.

Similarly, a DB2 JDBC trace of a Java application with two threads might generate the following JDBC trace log files: `7960main.trc`, `7960Thread-1.trc`.

Note: If the trace directory contains both old and new trace log files, file date and time stamp information can be used to locate the most recent trace files.

If no DB2 CLI or DB2 JDBC trace output files appear to have been created:

- Verify that the trace configuration keywords are set correctly in the `db2cli.ini` file. Issuing the `db2 GET CLI CFG FOR SECTION COMMON` command from the command line processor is a quick way to do this.
- Ensure the application is restarted after updating the `db2cli.ini` file. Specifically, the DB2 CLI and DB2 JDBC trace facilities are initialized during application startup. Once initialized, the DB2 JDBC trace facility cannot be reconfigured. The DB2 CLI trace facility can be reconfigured at run time but only if the TraceRefreshInterval keyword was appropriately specified prior to application startup.

  Note: Only the Trace and TracePIDList DB2 CLI keywords can be reconfigured at run time. *Changes made to other DB2 CLI keywords, including TraceRefreshInterval, have no effect without an application restart.*
- If the TraceRefreshInterval keyword was specified prior to application startup, and if the Trace keyword was initially set to 0, ensure that enough time has elapsed for the DB2 CLI trace facility to reread the Trace keyword value.
- If either or both the TracePathName and JDBCTracePathName keywords are used to specify trace directories, ensure those directories exist prior to starting the application.
- Ensure the application has write access to the specified trace log file or trace directory.
- Check the DB2CLIINIPATH environment variable. If set, the DB2 CLI and DB2 JDBC trace facilities expect the `db2cli.ini` file to be at the location specified by this variable.

- If the application uses ODBC to interface with the DB2 CLI driver, verify that one of the `SQLConnect()`, `SQLDriverConnect()` or `SQLBrowseConnect()` functions have been successfully called. No entries will be written to the DB2 CLI trace log files until a database connection has successfully been made.

CLI trace file interpretation:

DB2 CLI traces always begin with a header that identifies the process ID and thread ID of the application that generated the trace, the time the trace began, and product specific information such as the local DB2 build level and DB2 CLI driver version. For example:

```
1   [ Process: 1227, Thread: 1024 ]
2   [ Date, Time:           01-27-2002 13:46:07.535211 ]
3   [ Product:              QDB2/LINUX 7.1.0 ]
4   [ Level Identifier:     02010105 ]
5   [ CLI Driver Version:   07.01.0000 ]
6   [ Informational Tokens: "DB2 v7.1.0","n000510","" ]
```

Note: Trace examples used in this section have line numbers added to the left hand side of the trace. These line numbers have been added to aid the discussion and will *not* appear in an actual DB2 CLI trace.

Immediately following the trace header, there are usually a number of trace entries related to environment and connection handle allocation and initialization. For example:

```
7   SQLAllocEnv( phEnv=&bffff684 )
8       –––> Time elapsed - +9.200000E-004 seconds

9   SQLAllocEnv( phEnv=0:1 )
10      <––– SQL_SUCCESS   Time elapsed - +7.500000E-004 seconds

11  SQLAllocConnect( hEnv=0:1, phDbc=&bffff680 )
12      –––> Time elapsed - +2.334000E-003 seconds

13  SQLAllocConnect( phDbc=0:1 )
14      <––– SQL_SUCCESS   Time elapsed - +5.280000E-004 seconds

15  SQLSetConnectOption( hDbc=0:1, fOption=SQL_ATTR_AUTOCOMMIT, vParam=0 )
16      –––> Time elapsed - +2.301000E-003 seconds

17  SQLSetConnectOption( )
18      <––– SQL_SUCCESS   Time elapsed - +3.150000E-004 seconds

19  SQLConnect( hDbc=0:1, szDSN="SAMPLE", cbDSN=-3, szUID="", cbUID=-3,
                              szAuthStr="", cbAuthStr=-3 )
20      –––> Time elapsed - +7.000000E-005 seconds
21  ( DBMS NAME="DB2/LINUX", Version="07.01.0000", Fixpack="0x22010105" )

22  SQLConnect( )
23      <––– SQL_SUCCESS   Time elapsed - +5.209880E-001 seconds
24  ( DSN=""SAMPLE"" )

25  ( UID=" " )

26  ( PWD="*" )
```

In the above trace example, notice that there are two entries for each DB2 CLI function call (for example, lines 19-21 and 22-26 for the `SQLConnect()` function call). This is always the case in DB2 CLI traces. The first entry shows the input parameter values passed to the function call while the second entry shows the function output parameter values and return code returned to the application.

The above trace example shows that the `SQLAllocEnv()` function successfully allocated an environment handle (phEnv=0:1) at line 9. That handle was then passed to the `SQLAllocConnect()` function which successfully allocated a database connection handle (phDbc=0:1) as of line 13. Next, the `SQLSetConnectOption()` function was used to set the phDbc=0:1 connection's SQL_ATTR_AUTOCOMMIT attribute to SQL_AUTOCOMMIT_OFF (vParam=0) at line 15. Finally, `SQLConnect()` was called to connect to the target database (SAMPLE) at line 19.

Included in the input trace entry of the `SQLConnect()` function on line 21 is the build and FixPak level of the target database server. Other information that might also appear in this trace entry includes input connection string keywords and the code pages of the client and server. For example, suppose the following information also appeared in the `SQLConnect()` trace entry:

```
( Application Codepage=819, Database  Codepage=819,
  Char Send/Recv Codepage=819, Graphic Send/Recv Codepage=819,
  Application Char Codepage=819, Application Graphic Codepage=819 )
```

This would mean the application and the database server were using the same code page (819).

The return trace entry of the `SQLConnect()` function also contains important connection information (lines 24-26 in the above example trace). Additional information that might be displayed in the return entry includes any PATCH1 or PATCH2 keyword values that apply to the connection. For example, if PATCH2=27,28 was specified in the db2cli.ini file under the COMMON section, the following line should also appear in the `SQLConnect()` return entry:

```
( PATCH2="27,28" )
```

Following the environment and connection related trace entries are the statement related trace entries. For example:

```
27  SQLAllocStmt( hDbc=0:1, phStmt=&bffff684 )
28      ——> Time elapsed - +1.868000E-003 seconds

29  SQLAllocStmt( phStmt=1:1 )
30      <—— SQL_SUCCESS   Time elapsed - +6.890000E-004 seconds

31  SQLExecDirect( hStmt=1:1, pszSqlStr="CREATE TABLE GREETING (MSG
                                 VARCHAR(10))", cbSqlStr=-3 )
32      ——> Time elapsed - +2.863000E-003 seconds
33  ( StmtOut="CREATE TABLE GREETING (MSG VARCHAR(10))" )

34  SQLExecDirect( )
35      <—— SQL_SUCCESS   Time elapsed - +2.387800E-002 seconds
```

In the above trace example, the database connection handle (phDbc=0:1) was used to allocate a statement handle (phStmt=1:1) at line 29. An unprepared SQL statement was then executed on that statement handle at line 31. If the TraceComm=1 keyword had been set in the `db2cli.ini` file, the `SQLExecDirect()` function call trace entries would have shown additional client-server communication information as follows:

```
SQLExecDirect( hStmt=1:1, pszSqlStr="CREATE TABLE GREETING (MSG
                             VARCHAR(10))", cbSqlStr=-3 )
    ——> Time elapsed - +2.876000E-003 seconds
( StmtOut="CREATE TABLE GREETING (MSG VARCHAR(10))" )

    sqlccsend( ulBytes - 232 )
    sqlccsend( Handle - 1084869448 )
    sqlccsend( ) - rc - 0, time elapsed - +1.150000E-004
    sqlccrecv( )
    sqlccrecv( ulBytes - 163 ) - rc - 0, time elapsed - +2.243800E-002

SQLExecDirect( )
    <--- SQL_SUCCESS   Time elapsed - +2.384900E-002 seconds
```

Notice the additional sqlccsend() and sqlccrecv() function call information in this trace entry. The sqlccsend() call information reveals how much data was sent from the client to the server, how long the transmission took, and the success of that transmission (0 = SQL_SUCCESS). The sqlccrecv() call information then reveals how long the client waited for a response from the server and the amount of data included in the response.

Often, multiple statement handles will appear in the DB2 CLI trace. By paying close attention to the statement handle identifier, one can easily follow the execution path of a statement handle independent of all other statement handles appearing in the trace.

Statement execution paths appearing in the DB2 CLI trace are usually more complicated than the example shown above. For example:

```
36  SQLAllocStmt( hDbc=0:1, phStmt=&bffff684 )
37      ---> Time elapsed - +1.532000E-003 seconds

38  SQLAllocStmt( phStmt=1:2 )
39      <--- SQL_SUCCESS   Time elapsed - +6.820000E-004 seconds

40  SQLPrepare( hStmt=1:2, pszSqlStr="INSERT INTO GREETING VALUES ( ? )",
                cbSqlStr=-3 )
41      ---> Time elapsed - +2.733000E-003 seconds
42  ( StmtOut="INSERT INTO GREETING VALUES ( ? )" )

43  SQLPrepare( )
44      <--- SQL_SUCCESS   Time elapsed - +9.150000E-004 seconds

45  SQLBindParameter( hStmt=1:2, iPar=1, fParamType=SQL_PARAM_INPUT,
                      fCType=SQL_C_CHAR, fSQLType=SQL_CHAR, cbColDef=14,
                      ibScale=0, rgbValue=&080eca70, cbValueMax=15,
                      pcbValue=&080eca4c )
46      ---> Time elapsed - +4.091000E-003 seconds

47  SQLBindParameter( )
48      <--- SQL_SUCCESS   Time elapsed - +6.780000E-004 seconds

49  SQLExecute( hStmt=1:2 )
50      ---> Time elapsed - +1.337000E-003 seconds
51  ( iPar=1, fCType=SQL_C_CHAR, rgbValue="Hello World!!!", pcbValue=14,
      piIndicatorPtr=14 )

52  SQLExecute( )
53      <--- SQL_ERROR   Time elapsed - +5.951000E-003 seconds
```

In the above trace example, the database connection handle (phDbc=0:1) was used to allocate a second statement handle (phStmt=1:2) at line 38. An SQL statement with one parameter marker was then prepared on that statement handle at line 40. Next, an input parameter (iPar=1) of the appropriate SQL type (SQL_CHAR) was bound to the parameter marker at line 45. Finally, the statement was executed at line 49. Notice that both the contents and length of the input parameter (rgbValue="Hello World!!!", pcbValue=14) are displayed in the trace on line 51.

The SQLExecute() function fails at line 52. If the application calls a diagnostic DB2 CLI function like SQLError() to diagnose the cause of the failure, then that cause will appear in the trace. For example:

```
54  SQLError( hEnv=0:1, hDbc=0:1, hStmt=1:2, pszSqlState=&bffff680,
              pfNativeError=&bfffee78, pszErrorMsg=&bffff280,
              cbErrorMsgMax=1024, pcbErrorMsg=&bfffee76 )
55      ———> Time elapsed - +1.512000E-003 seconds

56  SQLError( pszSqlState="22001", pfNativeError=-302, pszErrorMsg="[IBM][CLI
       Driver][DB2/LINUX] SQL0302N  The value of a host variable in the EXECUTE
       or OPEN statement is too large for its corresponding use.
       SQLSTATE=22001", pcbErrorMsg=157 )
57      <——— SQL_SUCCESS   Time elapsed - +8.060000E-004 seconds
```

The error message returned at line 56 contains the DB2 native error code that was generated (SQL0302N), the sqlstate that corresponds to that code (SQLSTATE=22001) and a brief description of the error. In this example, the source of the error is evident: on line 49, the application is trying to insert a string with 14 characters into a column defined as VARCHAR(10) on line 31.

If the application does not respond to a DB2 CLI function warning or error return code by calling a diagnostic function like SQLError(), the warning or error message should still be written to the DB2 CLI trace. However, the location of that message in the trace may not be close to where the error actually occurred. Furthermore, the trace will indicate that the error or warning message was not retrieved by the application. For example, if not retrieved, the error message in the above example might not appear until a later, seemingly unrelated DB2 CLI function call as follows:

```
SQLDisconnect( hDbc=0:1 )
    ———> Time elapsed - +1.501000E-003 seconds
    sqlccsend( ulBytes - 72 )
    sqlccsend( Handle - 1084869448 )
    sqlccsend( ) - rc - 0, time elapsed - +1.080000E-004
    sqlccrecv( )
    sqlccrecv( ulBytes - 27 ) - rc - 0, time elapsed - +1.717950E-001
( Unretrieved error message="SQL0302N  The value of a host variable in the
  EXECUTE or OPEN statement is too large for its corresponding use.
  SQLSTATE=22001" )

SQLDisconnect( )
    <——— SQL_SUCCESS   Time elapsed - +1.734130E-001 seconds
```

The final part of a DB2 CLI trace should show the application releasing the database connection and environment handles that it allocated earlier in the trace. For example:

```
58  SQLTransact( hEnv=0:1, hDbc=0:1, fType=SQL_ROLLBACK )
59      ———> Time elapsed - +6.085000E-003 seconds
60  ( ROLLBACK=0 )

61  SQLTransact( )
        <——— SQL_SUCCESS   Time elapsed - +2.220750E-001 seconds

62  SQLDisconnect( hDbc=0:1 )
63      ———> Time elapsed - +1.511000E-003 seconds

64  SQLDisconnect( )
65      <——— SQL_SUCCESS   Time elapsed - +1.531340E-001 seconds

66  SQLFreeConnect( hDbc=0:1 )
67      ———> Time elapsed - +2.389000E-003 seconds

68  SQLFreeConnect( )
```

```
69      <——— SQL_SUCCESS   Time elapsed - +3.140000E-004 seconds

70  SQLFreeEnv( hEnv=0:1 )
71      ———> Time elapsed - +1.129000E-003 seconds

72  SQLFreeEnv( )
73      <——— SQL_SUCCESS   Time elapsed - +2.870000E-004 seconds
```

JDBC trace file interpretation:

DB2 JDBC traces always begin with a header that lists important system information such as key environment variable settings, the SDK for Java or JRE level, the DB2 JDBC driver level, and the DB2 build level. For example:

```
1   ========================================================
2   |    Trace beginning on 2002-1-28 7:21:0.19
3   ========================================================

4   System Properties:
5   ------------------
6   user.language = en
7   java.home = c:\Program Files\SQLLIB\java\jdk\bin\..
8   java.vendor.url.bug =
9   awt.toolkit = sun.awt.windows.WToolkit
10  file.encoding.pkg = sun.io
11  java.version = 1.1.8
12  file.separator = \
13  line.separator =
14  user.region = US
15  file.encoding = Cp1252
16  java.compiler = ibmjitc
17  java.vendor = IBM Corporation
18  user.timezone = EST
19  user.name = db2user
20  os.arch = x86
21  java.fullversion = JDK 1.1.8 IBM build n118p-19991124 (JIT ibmjitc
                       V3.5-IBMJDK1.1-19991124)
22  os.name = Windows NT
23  java.vendor.url = http://www.ibm.com/
24  user.dir = c:\Program Files\SQLLIB\samples\java
25  java.class.path =
         .:C:\Program Files\SQLLIB\lib;C:\Program Files\SQLLIB\java;
         C:\Program Files\SQLLIB\java\jdk\bin\
26  java.class.version = 45.3
27  os.version = 5.0
28  path.separator = ;
29  user.home = C:\home\db2user
30  ----------------------------------------
```

Note: Trace examples used in this section have line numbers added to the left hand side of the trace. These line numbers have been added to aid the discussion and will *not* appear in an actual DB2 JDBC trace.

Immediately following the trace header, one usually finds a number of trace entries related to initialization of the JDBC environment and database connection establishment. For example:

```
31  jdbc.app.DB2Driver –> DB2Driver() (2002-1-28 7:21:0.29)
32  | Loaded db2jdbc from java.library.path
33  jdbc.app.DB2Driver <– DB2Driver() [Time Elapsed = 0.01]

34  DB2Driver - connect(jdbc:db2:sample)

35  jdbc.app.DB2ConnectionTrace –> connect( sample, info, db2driver, 0, false )
                   (2002-1-28 7:21:0.59)
36  | 10: connectionHandle = 1
```

```
37  jdbc.app.DB2ConnectionTrace <— connect() [Time Elapsed = 0.16]

38  jdbc.app.DB2ConnectionTrace —> DB2Connection (2002-1-28 7:21:0.219)
39  | source = sample
40  | Connection handle = 1
41  jdbc.app.DB2ConnectionTrace <— DB2Connection
```

In the above trace example, a request to load the DB2 JDBC driver was made on line 31. This request returned successfully as reported on line 33.

The DB2 JDBC trace facility uses specific Java classes to capture the trace information. In the above trace example, one of those trace classes, DB2ConnectionTrace, has generated two trace entries numbered 35-37 and 38-41.

Line 35 shows the connect() method being invoked and the input parameters to that method call. Line 37 shows that the connect() method call has returned successfully while line 36 shows the output parameter of that call (Connection handle = 1).

Following the connection related entries, one usually finds statement related entries in the JDBC trace. For example:

```
42  jdbc.app.DB2ConnectionTrace —> createStatement() (2002-1-28 7:21:0.219)
43  | Connection handle = 1
44  | jdbc.app.DB2StatementTrace —> DB2Statement( con, 1003, 1007 )
          (2002-1-28 7:21:0.229)
45  | jdbc.app.DB2StatementTrace <— DB2Statement() [Time Elapsed = 0.0]
46  | jdbc.app.DB2StatementTrace —> DB2Statement (2002-1-28 7:21:0.229)
47  | | Statement handle = 1:1
48  | jdbc.app.DB2StatementTrace <— DB2Statement
49  jdbc.app.DB2ConnectionTrace <— createStatement - Time Elapsed = 0.01

50  jdbc.app.DB2StatementTrace —> executeQuery(SELECT * FROM EMPLOYEE WHERE
                                 empno = 000010) (2002-1-28 7:21:0.269)
51  | Statement handle = 1:1
52  | jdbc.app.DB2StatementTrace —> execute2( SELECT * FROM EMPLOYEE WHERE
                                   empno = 000010 ) (2002-1-28 7:21:0.269)
52  | | jdbc.DB2Exception —> DB2Exception() (2002-1-28 7:21:0.729)
53  | | | 10: SQLError = [IBM][CLI Driver][DB2/NT] SQL0401N  The data types of
                the operands for the operation "=" are not compatible.
                SQLSTATE=42818
54  | | |     SQLState = 42818
55  | | |     SQLNativeCode = -401
56  | | |     LineNumber = 0
57  | | |     SQLerrmc = =
58  | | jdbc.DB2Exception <— DB2Exception() [Time Elapsed = 0.0]
59  | jdbc.app.DB2StatementTrace <— executeQuery - Time Elapsed = 0.0
```

On line 42 and 43, the DB2ConnectionTrace class reported that the JDBC createStatement() method had been called with connection handle 1. Within that method, the internal method DB2Statement() was called as reported by another DB2 JDBC trace facility class, DB2StatementTrace. Notice that this internal method call appears 'nested' in the trace entry. Lines 47-49 show that the methods returned successfully and that statement handle 1:1 was allocated.

On line 50, an SQL query method call is made on statement 1:1, but the call generates an exception at line 52. The error message is reported on line 53 and contains the DB2 native error code that was generated (SQL0401N), the sqlstate that corresponds to that code (SQLSTATE=42818) and a brief description of the error. In this example, the error results because the EMPLOYEE.EMPNO column is defined as CHAR(6) and not an integer value as assumed in the query.

Related concepts:

- "CLI/ODBC/JDBC trace facility" on page 192

Related reference:

- "Miscellaneous variables" in *Performance Guide*
- "Trace CLI/ODBC configuration keyword" in *Call Level Interface Guide and Reference, Volume 1*
- "TraceComm CLI/ODBC configuration keyword" in *Call Level Interface Guide and Reference, Volume 1*
- "TraceFileName CLI/ODBC configuration keyword" in *Call Level Interface Guide and Reference, Volume 1*
- "TracePathName CLI/ODBC configuration keyword" in *Call Level Interface Guide and Reference, Volume 1*
- "TracePIDList CLI/ODBC configuration keyword" in *Call Level Interface Guide and Reference, Volume 1*
- "TraceRefreshInterval CLI/ODBC configuration keyword" in *Call Level Interface Guide and Reference, Volume 1*

Chapter 8. Java 2 Platform, Enterprise Edition

The sections that follow describe the Java 2 Platform, Enterprise Edition (J2EE).

Java 2 Platform, Enterprise Edition Overview

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. The Java 2 Platform, Enterprise Edition (J2EE), reduces the cost and complexity of developing these multi-tier services, resulting in services that can be rapidly deployed and easily enhanced based on the requirements of the enterprise.

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

Related concepts:

- "Java 2 Platform, Enterprise Edition" on page 207

Java 2 Platform, Enterprise Edition

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*.

Related concepts:
- "Enterprise Java Beans" on page 215
- "Java 2 Platform, Enterprise Edition containers" on page 208

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

Related concepts:

- "Enterprise Java Beans" on page 215
- "Java Naming and Directory Interface (JNDI)" on page 209

Java 2 Platform, Enterprise Edition Server

Underlying a Java 2 Platform, Enterprise Edition (J2EE) container is the server of which the container is a part. 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 database, accessible through the JDBC API, for the storage of business data. The database is accessible from web components, enterprise beans, and application client components. The database need not be accessible from applets.

Related concepts:

- "Supported drivers for JDBC and SQLJ" on page 1

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), Novell Directory Services, and Network Information System (NIS); 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 DB2 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 DB2 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 UDB for iSeries 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 DB2 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 DB2 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:

`http://www.ibm.com/software/webservers/appserv/library.html`

Example of a distributed transaction that uses JTA methods

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 62 outlines code that executes local transactions.

```
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 62. 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. Distributed transactions typically involve multiple connections to the same data source or different data sources, which can include data sources from different manufacturers.

Figure 63 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.

```
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 63. Example of a distributed transaction under an application server

Figure 64 on page 212 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.

```
class XASample
{
  javax.sql.XADataSource xaDS1;
  javax.sql.XADataSource xaDS2;
  javax.sql.XAConnection xaconn1;
  javax.sql.XAConnection xaconn2;
  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;

    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.DB2XADataSource,
        * which can be used only if this is a DB2 connection.
        */
      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();
```

Figure 64. Example of a distributed transaction that uses the JTA (Part 1 of 4)

```
      // 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.
        rc1 = xares1.prepare(xid1);
        if(rc1 == javax.transaction.xa.XAResource.XA_OK)
        { // Prepare was successful. Prepare the second connection.
          rc2 = xares2.prepare(xid1);
          if(rc2 == javax.transaction.xa.XAResource.XA_OK)
          { // Both connections prepared successfully and neither was read-only.
            xares1.commit(xid1, false);
            xares2.commit(xid1, false);
          }
          else if(rc2 == javax.transaction.xa.XAException.XA_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.
          }
        }
      }
```

Figure 64. Example of a distributed transaction that uses the JTA (Part 2 of 4)

```
      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());
        xae.printStackTrace();
        try
        {
          xares1.rollback(xid1);
        }
        catch (javax.transaction.xa.XAException xae1)
        { // Report failure of rollback.
          System.out.println("distributed Transaction rollback xares1 failed");
          System.out.println("XAException error code = " + xae1.errorCode);
          System.out.println("XAException message = " + xae1.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();
      }
    }
```

Figure 64. Example of a distributed transaction that uses the JTA (Part 3 of 4)

```
    catch (java.sql.SQLException sqe)
    {
      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());
    }
  }
}
```

Figure 64. Example of a distributed transaction that uses the JTA (Part 4 of 4)

Recommendation: For better performance, complete a distributed transaction before you start another distributed or local transaction.

Related concepts:

- "Java transaction management" on page 209

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 DB2 database. 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 DB2 database. 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 DB2 database can be provided from the session bean or entity bean. The two services of the sample application demonstrate both approaches to accessing the DB2 database. In the first service, the entity bean is used:

```
//======================================================
// 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 DB2 sample database.

```
/=============================================
* 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;
```

Related reference:

- "Java WebSphere samples" on page 179

Chapter 9. JDBC and SQLJ connection pooling support

Connection pooling is part of JDBC `DataSource` support, and is supported by the IBM DB2 Driver for JDBC and SQLJ.

The IBM DB2 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 database server. 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 10. IBM DB2 Driver for JDBC and SQLJ support for connection concentrator and Sysplex workload balancing

The following topics contain information about IBM DB2 Driver for JDBC and SQLJ support for the connection concentrator and Sysplex workload balancing functions of DB2.

JDBC connection concentrator and Sysplex workload balancing

Java applications that use IBM DB2 Driver for JDBC and SQLJ type 4 connectivity to access DB2 for z/OS servers can take advantage of the connection concentrator and Sysplex workload balancing functions.

The IBM DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing functions are similar to the connection concentrator and Sysplex workload balancing functions of DB2 Connect.

The IBM DB2 Driver for JDBC and SQLJ connection concentrator can reduce the resources that DB2 for z/OS database servers require to support large numbers of client applications. The IBM DB2 Driver for JDBC and SQLJ connection concentrator function lets many connection objects use the same physical connection, which reduces the total number of physical connections to the database server.

IBM DB2 Driver for JDBC and SQLJ Sysplex workload balancing can improve availability of a data sharing group. When Sysplex workload balancing is enabled, the driver gets frequent status information about the members of a data sharing group. The driver uses this information to determine the data sharing member to which the next transaction should be routed. With Sysplex workload balancing, the DB2 for z/OS server and Workload Manager for z/OS (WLM) ensure that work is distributed efficiently among members of the data sharing group and that work is transferred to another member of a data sharing group if one member has a failure.

The IBM DB2 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 database server. When you enable the connection concentrator and Sysplex workload balancing, you set the maximum number of physical connections to the database server at any point in time by setting the maximum number of transport objects.

At the driver level, you set limits on the number of transport objects using IBM DB2 Driver for JDBC and SQLJ configuration properties.

At the connection level, you enable and disable the IBM DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing and set limits on the number of transport objects using DataSource properties.

You can monitor the global transport objects pool in either of the following ways:

- Using traces that you start using IBM DB2 Driver for JDBC and SQLJ configuration properties
- Using an application programming interface

Related concepts:

- "IBM DB2 Driver for JDBC and SQLJ configuration properties customization" on page 11

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

Example of enabling the IBM DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing

The following procedure is an example of enabling the IBM DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing functions with WebSphere Application Server. The values that are specified are not intended to be recommended values. You need to determine values based on factors like these:

- Availability of system resources
- The number of physical connections available
- The desired ratio of connection objects to transport objects

Prerequisites:

Server requirements:

- WLM for z/OS
- DB2 UDB for OS/390 and z/OS Version 7 or later, set up for data sharing

  The default values for special registers in all members of the data sharing group must be the same. The reason for this is that when the IBM DB2 Driver for JDBC and SQLJ balances the loads on each member of the data sharing group, it moves the user's connection from one member to another. If the user has set any special register values on the original data sharing member, the driver resets all special registers to their default values and then applies any special register changes to the new member. However, the IBM DB2 Driver for JDBC and SQLJ has no way to determine the default values for all members. If two members have different default values, the result of an SQL statement can differ, depending on which member the statement runs on.

Client requirements:

- IBM DB2 Driver for JDBC and SQLJ at the FixPak 10 level
- WebSphere Application Server, Version 5.1 or later

Procedure:

1. Verify that the IBM DB2 Driver for JDBC and SQLJ is at the correct level to support the connection concentrator and Sysplex workload balancing by issuing the following command in the command line processor:

   ```
   java com.ibm.db2.jcc.DB2Jcc -version
   ```

   Find a line in the output like this:

   ```
   [ibm][db2][jcc] Driver: IBM DB2 Driver for JDBC and SQLJ Architecture nnn xxx
   ```

   nnn should be 2.7 or later.

2. Set IBM DB2 Driver for JDBC and SQLJ configuration properties to enable the connection concentrator or Sysplex workload balancing for all `DataSource` instances that are created under the driver.

   Set the configuration properties in a DB2JccConfiguration.properties file.

   a. Create a DB2JccConfiguration.properties file or edit the existing DB2JccConfiguration.properties file.
   b. Set the following configuration properties:
      - db2.jcc.minTransportObjects
      - db2.jcc.maxTransportObjects
      - db2.jcc.maxTransportObjectWaitTime
      - db2.jcc.dumpPool
      - db2.jcc.dumpPoolStatisticsOnScheduleFile

      Start with settings similar to these:

      ```
      db2.jcc.minTransportObjects=0
      db2.jcc.maxTransportObjects=1500
      db2.jcc.maxTransportObjectWaitTime=-1
      db2.jcc.dumpPool=0
      db2.jcc.dumpPoolStatisticsOnScheduleFile=/home/WAS/logs/srv1/poolstats
      ```
   c. Add the directory path for DB2JccConfiguration.properties to the WebSphere Application Server IBM DB2 Driver for JDBC and SQLJ classpath.
3. Set IBM DB2 Driver for JDBC and SQLJ data source properties to enable the connection concentrator or Sysplex workload balancing.

   In the WebSphere Application Server administrative console, set the following properties for the data source that your application uses to connect to the database server:
   - enableSysplexWLB
   - enableConnectionConcentrator
   - maxTransportObjects

   Assume that you want the connection concentrator function as well the Sysplex workload balancing function. Start with settings similar to these:

Table 27. Example of data source property settings for IBM DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing

Property	**Setting**
enableSysplexWLB	true[1]
maxTransportObjects	100

Note:

1. enableConnectionConcentrator is set to true by default because enableSysplexWLB is set to true.

4. Restart WebSphere Application Server.

Related concepts:

- "JDBC connection concentrator and Sysplex workload balancing" on page 221

Techniques for monitoring IBM DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing

To monitor the IBM DB2 Driver for JDBC and SQLJ connection concentrator and Sysplex workload balancing, 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 DB2 Driver for JDBC and SQLJ configuration properties
- Using an application programming interface

Configuration properties for monitoring the global transport objects pool:

The db2.jcc.dumpPool, db2.jcc.dumpPoolStatisticsOnSchedule, and db2.jcc.dumpPoolStatisticsOnScheduleFile configuration properties control tracing of the global transport objects pool.

For example, the following set of configuration property settings cause Sysplex 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:

```
time Scheduled PoolStatistics npr:2575 nsr:2575 lwroc:439 hwroc:1764 coc:372
aooc:362 rmoc:362 nbr:2872 tbt:857520 tpo:10
```

The meanings of the fields are:

npr
The total number of requests that the IBM DB2 Driver for JDBC and SQLJ has made to the pool since the pool was created.

nsr
The number of successful requests that the IBM DB2 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 DB2 Driver for JDBC and SQLJ created since the pool was created.

aooc
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 DB2 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:

```
import com.ibm.db2.jcc.DB2PoolMonitor;
DB2PoolMonitor transportObjectPoolMonitor =
       DB2PoolMonitor.getPoolMonitor (DB2PoolMonitor.TRANSPORT_OBJECT);
```

After you create the DB2PoolMonitor object, you can use the following methods to monitor the pool.

getMonitorVersion

Format:

```
public int getMonitorVersion()
```

Retrieves the version of the DB2PoolMonitor class that is shipped with the IBM DB2 Driver for JDBC and SQLJ.

totalRequestsToPool

Format:

```
public int totalRequestsToPool()
```

Retrieves the total number of requests that the IBM DB2 Driver for JDBC and SQLJ has made to the pool since the pool was created.

successfullRequestsFromPool

Format:

```
public int successfullRequestsFromPool()
```

Retrieves the number of successful requests that the IBM DB2 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.

numberOfRequestsBlocked

Format:

```
public int numberOfRequestsBlocked()
```

Retrieves the number of requests that the IBM DB2 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.

totalTimeBlocked
Format:

```
public 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.

lightWeightReusedObjectCount
Format:

```
public 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.

heavyWeightReusedObjectCount
Format:

```
public int heavyWeightReusedObjectCount()
```

Retrieves the number of objects that were reused from the pool.

createdObjectCount
Format:

```
public int createdObjectCount()
```

Retrieves the number of objects that the IBM DB2 Driver for JDBC and SQLJ created since the pool was created.

agedOutObjectCount
Format:

```
public 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.

removedObjectCount
Format:

```
public int removedObjectCount()
```

Retrieves the number of objects that have been deleted from the pool since the pool was created.

totalPoolObjects
Format:

```
public int totalPoolObjects()
```

Retrieves the number of objects that are currently in the pool.

Related concepts:

- "JDBC connection concentrator and Sysplex workload balancing" on page 221

Chapter 11. JDBC and SQLJ reference

The topics that follow contain reference information about JDBC methods and SQLJ clauses.

- "Data types that map to SQL data types in JDBC applications"
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232
- "Driver support for JDBC APIs" on page 247
- "SQLJ statement reference" on page 265
- "IBM DB2 Driver for JDBC and SQLJ reference information" on page 300
- "Commands for SQLJ program preparation" on page 347

Data types that map to SQL data types in JDBC applications

The following tables summarize the mappings of Java data types to JDBC and SQL data types for a DB2 Database for Linux, UNIX, and Windows system.

Table 28 summarizes the mappings of Java data types to DB2 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 28. Mappings of Java data types to DB2 data types for updating DB2 tables

Java data type	SQL data type
short, boolean[1], byte[1]	SMALLINT
int, java.lang.Integer	INTEGER
long, java.lang.Long	BIGINT
float, java.lang.Float	REAL
double, java.lang.Double	DOUBLE
java.math.BigDecimal	DECIMAL(*p*,*s*)[2]
java.math.BigDecimal	DECFLOAT(*n*)[3,4]
java.lang.String	CHAR(*n*)[5]
java.lang.String	GRAPHIC(*m*)[6]
java.lang.String	VARCHAR(*n*)[7]
java.lang.String	VARGRAPHIC(*m*)[8]
java.lang.String	CLOB(*n*)[9]
java.lang.String	XML
byte[]	CHAR(*n*) FOR BIT DATA[5]
byte[]	VARCHAR(*n*) FOR BIT DATA[7]
byte[]	BINARY(*n*)[5]
byte[]	VARBINARY(*n*)[7]
byte[]	BLOB(*n*)[9,10]
byte[]	ROWID
byte[]	XML
java.sql.Blob	BLOB(*n*)[10]

Table 28. Mappings of Java data types to DB2 data types for updating DB2 tables (continued)

Java data type	SQL data type
java.sql.Blob	XML
java.sql.Clob	CLOB(*n*)[10]
java.sql.Clob	DBCLOB(*m*)[11]
java.sql.Clob	XML
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.io.ByteArrayInputStream	BLOB(*n*)[10]
java.io.StringReader	CLOB(*n*)[10]
java.io.ByteArrayInputStream	CLOB(*n*)[10]
java.io.InputStream	XML
com.ibm.db2.jcc.DB2RowID	ROWID
com.ibm.db2.DB2Xml	XML
java.net.URL	DATALINK[12]

Notes:

1. DB2 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 DB2 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 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 DB2 can determine the data type of the column.
10. *n*<=2147483647.
11. *m*<=1073741823.
12. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 29 summarizes the mappings of DB2 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 29. Mappings of DB2 data types to Java data types for retrieving data from DB2 tables

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
SMALLINT	short	byte, int, long, float, double, java.math.BigDecimal, boolean, java.lang.String

Table 29. Mappings of DB2 data types to Java data types for retrieving data from DB2 tables (continued)

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
INTEGER	int	short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String
BIGINT	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
DECFLOAT(*n*)[2,3]	java.math.BigDecimal	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
REAL	float	long, int, short, byte, double, java.math.BigDecimal, boolean, java.lang.String
DOUBLE	double	long, int, short, byte, float, java.math.BigDecimal, boolean, java.lang.String
CHAR(*n*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARCHAR(*n*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CHAR(*n*) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
VARCHAR(*n*) FOR BIT DATA	byte[]	java.lang.String, java.io.InputStream, java.io.Reader
BINARY(*n*)	byte[]	None
VARBINARY(*n*)	byte[]	None
GRAPHIC(*m*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
VARGRAPHIC(*m*)	java.lang.String	long, int, short, byte, float, double, java.math.BigDecimal, boolean, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.io.InputStream, java.io.Reader
CLOB(*n*)	java.sql.Clob	java.lang.String
BLOB(*n*)	java.sql.Blob	byte[][4]
DBCLOB(*m*)	No exact equivalent. Use java.sql.Clob.	
ROWID	com.ibm.db2.jcc.DB2RowID	byte[]
XML	com.ibm.db2.jcc.DB2Xml	byte[], java.lang.String, java.io.InputStream, java.io.Reader

Table 29. Mappings of DB2 data types to Java data types for retrieving data from DB2 tables (continued)

SQL data type	Recommended Java data type or Java object type	Other supported Java data types
DATE	java.sql.Date	java.sql.String, java.sql.Timestamp
TIME	java.sql.Time	java.sql.String, java.sql.Timestamp
TIMESTAMP	java.sql.Timestamp	java.sql.String, java.sql.Date, java.sql.Time, java.sql.Timestamp
DATALINK	java.net.URL[5]	

Notes:

1. 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.
2. *n*=16 or *n*=34.
3. DECFLOAT is valid for connections to DB2 Version 9.1 for z/OS or later database servers. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.
4. This mapping is valid only if DB2 can determine the data type of the column.
5. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 30 summarizes mappings of Java data types to JDBC data types and DB2 data types for 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 DB2 data types are for parameters in stored procedure or user-defined function invocations.

If more than one Java data type is listed in Table 30, the first data type is the **recommended** data type.

Table 30. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions

Java data type	JDBC data type	SQL data type
boolean[1]	BIT	SMALLINT
byte[1]	TINYINT	SMALLINT
short, java.lang.Integer	SMALLINT	SMALLINT
int, java.lang.Integer	INTEGER	INTEGER
long	BIGINT	BIGINT
float, java.lang.Float	REAL	REAL
float, java.lang.Float	FLOAT	REAL
double, java.lang.Double	DOUBLE	DOUBLE
java.math.BigDecimal	NUMERIC	DECIMAL
java.math.BigDecimal	DECIMAL	DECIMAL
java.lang.String	CHAR	CHAR
java.lang.String	CHAR	GRAPHIC
java.lang.String	VARCHAR	VARCHAR
java.lang.String	VARCHAR	VARGRAPHIC
java.lang.String	LONGVARCHAR	VARCHAR
java.lang.String	VARCHAR	CLOB(*n*)
java.lang.String	LONGVARCHAR	CLOB(*n*)

Table 30. Mappings of Java, JDBC, and SQL data types for calling stored procedures and user-defined functions (continued)

Java data type	JDBC data type	SQL data type
java.lang.String	CLOB	CLOB(*n*)
byte[]	BINARY	CHAR FOR BIT DATA
byte[]	VARBINARY	VARCHAR FOR BIT DATA
byte[]	BINARY	BINARY
byte[]	VARBINARY	VARBINARY
byte[]	LONGVARBINARY	VARCHAR FOR BIT DATA
byte[]	VARBINARY	BLOB(*n*)[2]
byte[]	LONGVARBINARY	BLOB(*n*)[2]
java.sql.Date	DATE	DATE
java.sql.Time	TIME	TIME
java.sql.Timestamp	TIMESTAMP	TIMESTAMP
java.sql.Blob	BLOB	BLOB
java.sql.Clob	CLOB	CLOB
java.sql.Clob	CLOB	DBCLOB
java.io.ByteArrayInputStream	None	BLOB(*n*)
java.io.StringReader	None	CLOB(*n*)
java.io.ByteArrayInputStream	None	CLOB(*n*)
com.ibm.db2.jcc.DB2RowID	com.ibm.db2.jcc.DB2Types.ROWID	ROWID
com.ibm.db2.jcc.DB2Xml	com.ibm.db2.jcc.DB2Types.XML	XML AS CLOB
java.net.URL	DATALINK	DATALINK[3]

Notes:

1. 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.
2. This mapping is valid only if DB2 can determine the data type of the column.
3. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 31 on page 232 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 31. 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

SQL data type in CREATE PROCEDURE or CREATE FUNCTION	Data type in Java stored procedure or user-defined function method[1]
SMALLINT	short, java.lang.Integer
INTEGER	int, java.lang.Integer
BIGINT	long, java.lang.Long
REAL	float, java.lang.Float
DOUBLE	double, java.lang.Double
DECIMAL	java.math.BigDecimal
DECFLOAT[2]	java.math.BigDecimal
CHAR	java.lang.String
GRAPHIC	java.lang.String
VARCHAR	java.lang.String
VARGRAPHIC	java.lang.String
CHAR FOR BIT DATA	byte[]
VARCHAR FOR BIT DATA	byte[]
BINARY	byte[]
VARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
DBCLOB	java.sql.Clob
ROWID	com.ibm.db2.jcc.DB2Types.ROWID
XML AS CLOB	com.ibm.db2.jcc.DB2Types.XML
DATALINK	java.net.URL[3]

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. DECFLOAT is valid for connections to DB2 Version 9.1 for z/OS or later database servers. Use of DECFLOAT requires the SDK for Java Version 5 (1.5) or later.
3. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Related reference:

- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

Properties for the IBM DB2 Driver for JDBC and SQLJ

Properties define how the connection to a particular data source should be made. Unless otherwise noted, properties can be set for a `DataSource` object or for a `Connection` object. Properties can be set in one of the following ways:

- Using `setXXX` methods

Properties are applicable to the following DB2-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`

See Summary of IBM DB2 Driver for JDBC and SQLJ extensions to JDBC for a summary of the property names and data types.

- In a `java.util.Properties` value in the *info* parameter of a `DriverManager.getConnection` call, as shown in Connect to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ.
- In a `java.lang.String` value in the *url* parameter of a DriverManager.getConnection call, as shown in Connect to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ.

The properties are:

blockingReadConnectionTimeout
The amount of time in seconds before a connection socket read times out. This property applies only to IBM DB2 Driver for JDBC and SQLJ type 4 connectivity, and affects all requests that are sent to the database server after a connection is successfully established. The default is 0. A value of 0 means that there is no timeout.

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. 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 value, but a Java `null` value is not valid.

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 DB2 Driver for JDBC and SQLJ supplies a value of db2jcc*thread-name*.

clientRerouteServerListJNDIName
Identifies a JNDI reference to a `DB2ClientRerouteServerList` instance in a JNDI repository of reroute server information. 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 database server fails

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 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.

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 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 value, but a Java `null` value is not valid.

connectionReuseProtocol
Specifies whether the connection state is reset when a connection is reused from a connection pool. Possible values are:

DIRTY_CONNECTION_REUSE
The connection state is not reset when a `Connection` is reused from a connection pool. Special register settings are not reset and temporary tables are not dropped. Specified property settings (derived from an application `DataSource` or WebSphere Application Server resource reference) might be passed by the pool module to the JDBC driver for reinitialization. Properties that are not passed by the pool module are not changed. All JDBC standard transient properties, such as isolation level, auto-commit mode, and read-only mode are reset to their JDBC defaults. These properties do not change:

- accountingInterval
- databaseName
- driverType
- pkList
- planName
- portNumber
- kerberosServerPrincipal
- password
- readOnly
- securityMechanism
- serverName
- user

RESET_CONNECTIONS_ON_REUSE
The connection state is reset when a `Connection` is reused from a connection pool. Special register settings are reset and temporary tables are dropped. Specified property settings (derived from an application `DataSource` or WebSphere Application Server resource reference) might be passed by the pool module to the JDBC driver for reinitialization. All JDBC standard transient properties, such as isolation level, auto-commit mode, and read-only mode are reset to their JDBC defaults. These properties do not change:

- accountingInterval
- databaseName
- driverType
- pkList
- planName
- portNumber
- kerberosServerPrincipal
- password
- readOnly
- securityMechanism
- serverName
- user

currentExplainMode
Specifies the value for the DB2 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 database servers that support the CURRENT EXPLAIN MODE special register, such as DB2 Database for Linux, UNIX, and Windows.

currentExplainSnapshot
Specifies the value for the DB2 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 database servers that support the CURRENT EXPLAIN SNAPSHOT special register, such as DB2 Database for Linux, UNIX, and Windows.

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.

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 database server.

currentMaintainedTableTypesForOptimization
Specifies a value that identifies the types of objects that can be considered when DB2 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 DB2 server searches these collections for JDBC and SQLJ packages.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the DB2 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 DB2 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 DB2 Driver for JDBC and SQLJ instance. To choose an instance of the IBM DB2 Driver for JDBC and SQLJ for a connection, you specify a currentPackageSet value that matches the collection ID for one of the IBM DB2 Driver for JDBC and SQLJ instances.

The precedence rules for the currentPackagePath and currentPackageSet properties follow the precedence rules for the DB2 CURRENT PACKAGESET and CURRENT PACKAGE PATH special registers.

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 Version 1 is performed to generate an access plan.

2 Specifies a level of optimization higher than that of DB2 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. Similar to 5 but without the heuristic rules.

9 Specifies a maximal 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.

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 a DB2 server.

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.

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 DB2 cursor. The data type of this property is int. Possible values are `TYPE_SCROLL_SENSITIVE_STATIC` (0), `TYPE_SCROLL_SENSITIVE_DYNAMIC` (1), or `TYPE_SCROLL_ASENSITIVE` (2). The default is `TYPE_SCROLL_SENSITIVE_STATIC`.

If the database server 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 UDB for iSeries database servers, which do not support sensitive static cursors, `java.sql.ResultSet.TYPE_SCROLL_SENSITIVE` always maps to SENSITIVE DYNAMIC.

databaseName
Specifies the name for the database server. This name is used as the *database* portion of the connection URL. The name depends on whether IBM DB2 Driver for JDBC and SQLJ type 4 connectivity or IBM DB2 Driver for JDBC and SQLJ type 2 connectivity is used.

For IBM DB2 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:
  ```
  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 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 DB2 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.

deferPrepares
Specifies whether to defer prepare operations until statement execution. The data type of this property is boolean. The default is `true` for IBM DB2 Driver for JDBC and SQLJ type 4 connectivity. The property is not applicable to IBM DB2 Driver for JDBC and SQLJ type 2 connectivity.

Deferring prepare operations can reduce network delays. However, if you defer prepare operations, you need to ensure that input data types match DB2 table column types.

description
A description of the data source. The data type of this property is String.

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.

enableConnectionConcentrator
Indicates whether the connection concentrator function of the IBM DB2 Driver for JDBC and SQLJ is enabled. The connection concentrator function is available only for connections to DB2 for z/OS servers.

The data type of enableConnectionConcentrator is boolean. The default is `false`. However, if enableSysplexWLB is set to `true`, the default is `true`.

enableSysplexWLB
Indicates whether the Sysplex workload balancing function of the IBM DB2 Driver for JDBC and SQLJ is enabled. The Sysplex workload balancing function is available only for connections to DB2 for z/OS servers.

The data type of enableSysplexWLB is boolean. The default is `false`. If enableSysplexWLB is set to `true`, enableConnectionConcentrator is set to true by default.

fullyMaterializeInputStreams
Indicates whether streams are fully materialized before they are sent from the client to a database server. 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.

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 database server supports progressive streaming:

- If the database server 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 database server supports progressive streaming:

  The JDBC driver ignores the value of fullyMaterializeLobData if the progresssiveLocators property is set to `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`.

This property has no effect on stored procedure parameters or LOBs that are fetched using scrollable cursors.

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 DB2. 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 DB2 Driver for JDBC and SQLJ type 4 connectivity.

For more information on using Kerberos security with the IBM DB2 Driver for JDBC and SQLJ, see Using Kerberos security under the IBM DB2 Driver for JDBC and SQLJ.

jdbcCollection
Specifies the collection ID for the packages that are used by an instance of the IBM DB2 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 DB2 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 DB2 Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

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 DB2 Driver for JDBC and SQLJ type 4 connectivity.

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 DB2 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.PrinterWriter. 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 database server fails.

  The data type of this property is int.The IBM DB2 Driver for JDBC and SQLJ uses the maxRetriesForClientReroute property only if the retryIntervalClientReroute property is also set.

maxTransportObjects
: Specifies the maximum number of transport objects that can be used for all connections with the associated DataSource object. The IBM DB2 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 database server.

  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.

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:

```
getConnection(user, password);
```

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.

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 database server. With progressive streaming, the database server 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 in int. Valid values are `DB2BaseDatSource.YES` (1) and `DB2BaseDatSource.NO` (2). If the progressiveStreaming property is not specified, the progressiveStreaming value is `DB2BaseDatSource.NOT_SET` (0).

If the connection is to a database server that supports progressive streaming, and the value of progressiveStreaming is `DB2BaseDatSource.YES` or `DB2BaseDatSource.NOT_SET`, the JDBC driver uses progressive streaming to return LOBs and XML data.

If the value of progressiveStreaming is `DB2BaseDataSource.NO`, or the database server 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.

DB2BaseDataSource.NOT_SET (0)

The effect of fullyMaterializeLobData depends on whether the database server supports progressive locators:

- If the database server 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 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 database server supports progressive locators:

  The JDBC driver ignores the value of fullyMaterializeLobData if the progresssiveLocators property is set to `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`.

This property has no effect on stored procedure parameters or LOBs that are fetched using scrollable cursors. LOB stored procedure parameters are always fully materialized. LOB locators are always used for data that is fetched using scrollable cursors.

queryCloseImplicit

Specifies whether cursors are closed immediately after all rows are fetched. queryCloseImplicit applies only to IBM DB2 Driver for JDBC and SQLJ type 4 connectivity to DB2 for z/OS database servers. Possible values are `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES` (1) and `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_NO` (2). The default is `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES`.

A value of `DB2BaseDataSource.QUERY_CLOSE_IMPLICIT_YES` can provide better performance because this setting results in less network traffic.

readOnly

Specifies whether the connection is read-only. The data type of this property is boolean. The default is `false`.

resultSetHoldability
Specifies whether cursors remain open after a commit operation. The data type of this property is int. Valid values are `HOLD_CURSORS_OVER_COMMIT` (1) or `CLOSE_CURSORS_AT_COMMIT` (2). These values are the same as the `ResultSet.HOLD_CURSORS_OVER_COMMIT` and `ResultSet.CLOSE_CURSORS_AT_COMMIT` constants that are defined in JDBC 3.0.

retrieveMessagesFromServerOnGetMessage
Specifies whether JDBC `SQLException.getMessage` calls cause the IBM DB2 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`, the following message 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.
```

If retrieveMessagesFromServerOnGetMessage is set to `false`, the following message is returned:

```
DB2 SQL error: SQLCODE: -204, SQLSTATE: 42704, SQLERRMC: ADMF001.NO_TABLE
```

An alternative to setting this property to `true` is to use the DB2-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 DB2 Driver for JDBC and SQLJ uses the retryIntervalClientReroute property only if the maxRetriesForClientRerouteOnly property is also set.

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.

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_SECURITY (11)
Kerberos

ENCRYPTED_USER_AND_DATA_SECURITY (12)
Encrypted user ID and encrypted security-sensitive data.

ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY (13)
Encrypted user ID and password, and encrypted security-sensitive data.

PLUGIN_SECURITY (15)
Plug-in security (DB2 Database for Linux, UNIX, and Windows only).

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 DB2 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 DB2 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 DB2 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 DB2 Driver for JDBC and SQLJ uses an SSL socket to connect to the DB2 server. If sslConnection is set to `true`, the connection uses an SSL socket. If sslConnection is set to `false`, the connection uses a plain socket.

statementReuseProtocol
Specifies how `Statement` objects are handled when a connection is returned to a connection pool. Possible values are:

NO_REUSE
`Statement` objects are closed when a connection is returned to the pool.

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.

DB2BaseDataSource.NOT_SET (0)

The effect of fullyMaterializeLobData depends on whether the database server supports progressive locators:

- If the database server 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 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 database server supports progressive locators:

  The JDBC driver ignores the value of fullyMaterializeLobData if the progresssiveLocators property is set to `DB2BaseDataSource.YES` or `DB2BaseDataSource.NOT_SET`.

This property has no effect on stored procedure parameters or LOBs that are fetched using scrollable cursors. LOB stored procedure parameters are always fully materialized. LOB locators are always used for data that is fetched using scrollable cursors.

supportsAsynchronousXARollback
Specifies whether the IBM DB2 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 DB2 shadow catalog tables or views that are searched when an application invokes a `DatabaseMetaData` method. The sysSchema property was formerly called cliSchema.

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*.

If traceFileName is not specified, *file-name* is traceFile. If traceFileName is also specified, *file-name* is the value traceFileName.

n is the *n*th 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.

traceFile

Specifies the name of a file into which the IBM DB2 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.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 DB2 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_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
  ```

usePool

Specifies whether the global transport object pool is used for the connection concentrator or Sysplex workload balancing functions of the IBM DB2 Driver for JDBC and SQLJ. The data type of usePool is boolean.

If usePool is set to `true`, the global transport object pool is checked for available transport objects before any new transport objects are created. If usePool is set to `false`, the global transport object pool is not checked before a new transport object is created. The transport object is created when an application requires it and deleted when the application no longer needs it.

The default for usePool is `true`. usePool is ignored if the connection concentrator or Sysplex workload balancing functions are disabled. Setting usePool to `false` can result in performance degradation and should be used only where an application must have a newly created transport object, or the application modifies the transport object so that it cannot be used by other applications.

useTargetColumnEncoding
Specifies whether to send single-byte character data for JDBC statement input parameters to the server in the encoding scheme of the target table column. The data type of this property is boolean. The default is `true`.

If useTargetColumnEncoding is false, or there is no encoding scheme information available for the target column, the data is sent to the database server in the UTF-8 or UCS-2 encoding scheme.

The value of useTargetColumnEncoding has no effect on mixed or double-byte character data. That data is sent to the server as Unicode.

The value of useTargetColumnEncoding has no effect on output data.

If useTargetColumnEncoding is true, and there is no Java runtime character-to-byte converter to convert the data to the CCSID of the DB2 table column, an exception is thrown.

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:

```
getConnection(user, password);
```

Related concepts:
- "JDBC connection concentrator and Sysplex workload balancing" on page 221
- "LOBs in JDBC applications with the IBM DB2 Driver for JDBC and SQLJ" on page 57
- "Security under the IBM DB2 Driver for JDBC and SQLJ" on page 142

Related tasks:
- "Connecting to a data source using the DataSource interface" on page 30
- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27

Related reference:
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335
- "Summary of IBM DB2 Driver for JDBC and SQLJ extensions to JDBC" on page 301

Driver support for JDBC APIs

The following tables list the JDBC interfaces and indicate which drivers supports them. The drivers and their supported platforms are:

Table 32. JDBC drivers for DB2 database servers

JDBC driver name	Associated DB2 database server
IBM DB2 Driver for JDBC and SQLJ	DB2 Database for Linux, UNIX, and Windows or DB2 for z/OS
DB2 JDBC Type 2 Driver for Linux, UNIX and Windows (deprecated)	DB2 Database for Linux, UNIX, and Windows

If a method has JDBC 2.0 and JDBC 3.0 forms, the IBM DB2 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 33. DB2 JDBC support for Array methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getArray	No	No
getBaseType	No	No
getBaseTypeName	No	No
getResultSet	No	No

Table 34. DB2 JDBC support for BatchUpdateException methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Exception	Yes	Yes
getUpdateCounts	Yes	Yes

Table 35. DB2 JDBC support for Blob methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getBinaryStream	Yes	Yes
getBytes	Yes	Yes
length	Yes	Yes
position	Yes	Yes
setBinaryStream[1]	Yes	No
setBytes[1]	Yes	No
truncate[1]	Yes	No

Notes:

1. This method can be used only if the `fullyMaterializeLobData` property is set to true.

Table 36. DB2 JDBC support for CallableStatement methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.sql.Statement	Yes	Yes
Methods inherited from java.sql.PreparedStatement	Yes[1]	Yes
getArray	No	No
getBigDecimal	Yes[2]	Yes
getBlob	Yes[2]	Yes
getBoolean	Yes[2]	Yes
getByte	Yes[2]	Yes
getBytes	Yes[2]	Yes
getClob	Yes[2]	Yes
getDate	Yes[2,3]	Yes[3]
getDouble	Yes[2]	Yes
getFloat	Yes[2]	Yes
getInt	Yes[2]	Yes
getLong	Yes[2]	Yes
getObject	Yes[2,4]	Yes[4]
getRef	No	No
getShort	Yes[2]	Yes
getString	Yes[2]	Yes
getTime	Yes[2,3]	Yes[3]
getTimestamp	Yes[2,3]	Yes[3]
getURL	Yes	No
registerOutParameter	Yes[5]	Yes[5]
setAsciiStream	Yes[6]	No
setBigDecimal	Yes[6]	No
setBinaryStream	Yes[6]	No
setBoolean	Yes[6]	No
setByte	Yes[6]	No
setBytes	Yes[6]	No
setCharacterStream	Yes[6]	No
setDate	Yes[6]	No
setDouble	Yes[6]	No
setFloat	Yes[6]	No
setInt	Yes[6]	No
setLong	Yes[6]	No
setNull	Yes[6,,7]	No
setObject	Yes[6,]	No
setShort	Yes[6]	No
setString	Yes[6]	No

Table 36. DB2 JDBC support for CallableStatement methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
setTime	Yes[6]	No
setTimestamp	Yes[6]	No
setURL	Yes	No
wasNull	Yes	Yes

Notes:

1. The inherited `getParameterMetaData` method is not supported if the database server is DB2 for z/OS.
2. The following forms of `CallableStatement.getXXX` methods are not supported if the database server is DB2 for z/OS:

   `get`*`XXX`*`(String `*`parameterName`*`)`
3. DB2 does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from DB2 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:

   `getObject(int `*`parameterIndex`*`, java.util.Map  `*`map`*`)`
5. The following form of the `registerOutParameter` method is not supported:

   `registerOutParameter(int `*`parameterIndex`*`, int `*`jdbcType`*`, String `*`typeName`*`)`
6. Not supported if the database server is DB2 for z/OS.
7. The following form of `setNull` is not supported:

   `setNull(int `*`parameterIndex`*`, int `*`jdbcType`*`, String `*`typeName`*`)`

Table 37. DB2 JDBC support for Clob methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getAsciiStream	Yes	Yes
getCharacterStream	Yes	Yes
getSubString	Yes	Yes
length	Yes	Yes
position	Yes	Yes
setAsciiStream[1]	Yes	No
setCharacterStream[1]	Yes	No
setString[1]	Yes	No
truncate[1]	Yes	No

Notes:

1. This method can be used only if the `fullyMaterializeLobData` property is set to true.

Table 38. DB2 JDBC support for Connection methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
clearWarnings	Yes	Yes

Table 38. DB2 JDBC support for Connection methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
close	Yes	Yes
commit	Yes	Yes
createStatement	Yes	Yes[1]
getAutoCommit	Yes	Yes
getCatalog	Yes	Yes
getHoldability	Yes	No
getMetaData	Yes	Yes
getTransactionIsolation	Yes	Yes
getTypeMap	No	No
getWarnings	Yes	Yes
isClosed	Yes	Yes
isReadOnly	Yes	Yes
nativeSQL	Yes	Yes
prepareCall	Yes	Yes
prepareStatement	Yes	Yes[1]
releaseSavepoint	Yes	No
rollback	Yes	Yes[1]
setAutoCommit	Yes	Yes
setCatalog	Yes	Yes
setReadOnly	Yes[2]	Yes
setSavepoint	Yes	No
setTransactionIsolation	Yes	Yes
setTypeMap	No	No

Notes:

1. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 forms of this method.
2. The driver does not use the setting. For the IBM DB2 Driver for JDBC and SQLJ, a connection can be set as read-only through the `readOnly` property for a `Connection` or `DataSource` object.

Table 39. DB2 JDBC support for ConnectionEvent methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.util.EventObject	Yes	Yes
getSQLException	Yes	Yes

Table 40. DB2 JDBC support for ConnectionEventListener methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
connectionClosed	Yes	Yes
connectionErrorOccurred	Yes	Yes

Table 41. DB2 JDBC support for ConnectionPoolDataSource methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getLoginTimeout	Yes	Yes
getLogWriter	Yes	Yes
getPooledConnection	Yes	Yes
setLoginTimeout	Yes[1]	Yes
setLogWriter	Yes	Yes

Note:

1. This method is not supported for IBM DB2 Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 42. DB2 JDBC support for DatabaseMetaData methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
allProceduresAreCallable	Yes	Yes
allTablesAreSelectable	Yes	Yes
dataDefinitionCausesTransactionCommit	Yes	Yes
dataDefinitionIgnoredInTransactions	Yes	Yes
deletesAreDetected	Yes	Yes
doesMaxRowSizeIncludeBlobs	Yes	Yes
getAttributes	Yes	No
getBestRowIdentifier	Yes	Yes
getCatalogs	Yes	Yes
getCatalogSeparator	Yes	Yes
getCatalogTerm	Yes	Yes
getColumnPrivileges	Yes	Yes
getColumns	Yes	Yes[1]
getConnection	Yes	Yes
getCrossReference	Yes	Yes
getDatabaseMajorVersion	Yes	No
getDatabaseMinorVersion	Yes	No
getDatabaseProductName	Yes	Yes
getDatabaseProductVersion	Yes	Yes
getDefaultTransactionIsolation	Yes	Yes
getDriverMajorVersion	Yes	Yes
getDriverMinorVersion	Yes	Yes
getDriverName	Yes	Yes
getDriverVersion	Yes	Yes
getExportedKeys	Yes	Yes
getExtraNameCharacters	Yes	Yes

Table 42. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getIdentifierQuoteString	Yes	Yes
getImportedKeys	Yes	Yes
getIndexInfo	Yes	Yes
getJDBCMajorVersion	Yes	No
getJDBCMinorVersion	Yes	No
getMaxBinaryLiteralLength	Yes	Yes
getMaxCatalogNameLength	Yes	Yes
getMaxCharLiteralLength	Yes	Yes
getMaxColumnNameLength	Yes	Yes
getMaxColumnsInGroupBy	Yes	Yes
getMaxColumnsInIndex	Yes	Yes
getMaxColumnsInOrderBy	Yes	Yes
getMaxColumnsInSelect	Yes	Yes
getMaxColumnsInTable	Yes	Yes
getMaxConnections	Yes	Yes
getMaxCursorNameLength	Yes	Yes
getMaxIndexLength	Yes	Yes
getMaxProcedureNameLength	Yes	Yes
getMaxRowSize	Yes	Yes
getMaxSchemaNameLength	Yes	Yes
getMaxStatementLength	Yes	Yes
getMaxStatements	Yes	Yes
getMaxTableNameLength	Yes	Yes
getMaxTablesInSelect	Yes	Yes
getMaxUserNameLength	Yes	Yes
getNumericFunctions	Yes	Yes
getPrimaryKeys	Yes	Yes
getProcedureColumns	Yes	Yes
getProcedures	Yes	Yes
getProcedureTerm	Yes	Yes
getResultSetHoldability	Yes	No
getSchemas	Yes	Yes[1]
getSchemaTerm	Yes	Yes
getSearchStringEscape	Yes	Yes
getSQLKeywords	Yes	Yes
getSQLStateType	Yes	No
getStringFunctions	Yes	Yes
getSuperTables	Yes[2]	No

Table 42. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getSuperTypes	Yes[2]	No
getSystemFunctions	Yes	Yes
getTablePrivileges	Yes	Yes
getTables	Yes	Yes[1]
getTableTypes	Yes	Yes
getTimeDateFunctions	Yes	Yes
getTypeInfo	Yes	Yes
getUDTs	No	Yes[2]
getURL	Yes	Yes
getUserName	Yes	Yes
getVersionColumns	Yes	Yes
insertsAreDetected	Yes	Yes
isCatalogAtStart	Yes	Yes
isReadOnly	Yes	Yes
nullPlusNonNullIsNull	Yes	Yes
nullsAreSortedAtEnd	Yes	Yes
nullsAreSortedAtStart	Yes	Yes
nullsAreSortedHigh	Yes	Yes
nullsAreSortedLow	Yes	Yes
othersDeletesAreVisible	Yes	Yes
othersInsertsAreVisible	Yes	Yes
othersUpdatesAreVisible	Yes	Yes
ownDeletesAreVisible	Yes	Yes
ownInsertsAreVisible	Yes	Yes
ownUpdatesAreVisible	Yes	Yes
storesLowerCaseIdentifiers	Yes	Yes
storesLowerCaseQuotedIdentifiers	Yes	Yes
storesMixedCaseIdentifiers	Yes	Yes
storesMixedCaseQuotedIdentifiers	Yes	Yes
storesUpperCaseIdentifiers	Yes	Yes
storesUpperCaseQuotedIdentifiers	Yes	Yes
supportsAlterTableWithAddColumn	Yes	Yes
supportsAlterTableWithDropColumn	Yes	Yes
supportsANSI92EntryLevelSQL	Yes	Yes
supportsANSI92FullSQL	Yes	Yes
supportsANSI92IntermediateSQL	Yes	Yes
supportsBatchUpdates	Yes	Yes
supportsCatalogsInDataManipulation	Yes	Yes

Table 42. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
supportsCatalogsInIndexDefinitions	Yes	Yes
supportsCatalogsInPrivilegeDefinitions	Yes	Yes
supportsCatalogsInProcedureCalls	Yes	Yes
supportsCatalogsInTableDefinitions	Yes	Yes
SupportsColumnAliasing	Yes	Yes
supportsConvert	Yes	Yes
supportsCoreSQLGrammar	Yes	Yes
supportsCorrelatedSubqueries	Yes	Yes
supportsDataDefinitionAndDataManipulationTransactions	Yes	Yes
supportsDataManipulationTransactionsOnly	Yes	Yes
supportsDifferentTableCorrelationNames	Yes	Yes
supportsExpressionsInOrderBy	Yes	Yes
supportsExtendedSQLGrammar	Yes	Yes
supportsFullOuterJoins	Yes	Yes
supportsGetGeneratedKeys	Yes	No
supportsGroupBy	Yes	Yes
supportsGroupByBeyondSelect	Yes	Yes
supportsGroupByUnrelated	Yes	Yes
supportsIntegrityEnhancementFacility	Yes	Yes
supportsLikeEscapeClause	Yes	Yes
supportsLimitedOuterJoins	Yes	Yes
supportsMinimumSQLGrammar	Yes	Yes
supportsMixedCaseIdentifiers	Yes	Yes
supportsMixedCaseQuotedIdentifiers	Yes	Yes
supportsMultipleOpenResults	Yes	No
supportsMultipleResultSets	Yes	Yes
supportsMultipleTransactions	Yes	Yes
supportsNamedParameters	Yes	No
supportsNonNullableColumns	Yes	Yes
supportsOpenCursorsAcross Commit	Yes	Yes
supportsOpenCursorsAcross Rollback	Yes	Yes
supportsOpenStatementsAcrossCommit	Yes	Yes
supportsOpenStatementsAcrossRollback	Yes	Yes
supportsOrderByUnrelated	Yes	Yes
supportsOuterJoins	Yes	Yes
supportsPositionedDelete	Yes	Yes
supportsPositionedUpdate	Yes	Yes
supportsResultSetConcurrency	Yes	Yes

Table 42. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
supportsResultSetHoldability	Yes	No
supportsResultSetType	Yes	Yes
supportsSavepoints	Yes	No
supportsSchemasInDataManipulation	Yes	Yes
supportsSchemasInIndexDefinitions	Yes	Yes
supportsSchemasInPrivilegeDefinitions	Yes	Yes
supportsSchemasInProcedureCalls	Yes	Yes
supportsSchemasInTableDefinitions	Yes	Yes
supportsSelectForUpdate	Yes	Yes
supportsStoredProcedures	Yes	Yes
supportsSubqueriesInComparisons	Yes	Yes
supportsSubqueriesInExists	Yes	Yes
supportsSubqueriesInIns	Yes	Yes
supportsSubqueriesInQuantifieds	Yes	Yes
supportsSuperTables	Yes	No
supportsSuperTypes	Yes	No
supportsTableCorrelationNames	Yes	Yes
supportsTransactionIsolationLevel	Yes	Yes
supportsTransactions	Yes	Yes
supportsUnion	Yes	Yes
supportsUnionAll	Yes	Yes
updatesAreDetected	Yes	Yes
usesLocalFilePerTable	Yes	Yes
usesLocalFiles	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver for Linux, UNIX and Windows does not support the JDBC 3.0 form of this method.
2. The method can be executed, but it returns an empty `ResultSet`.

Table 43. DB2 JDBC support for DataSource methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getConnection	Yes	Yes
getLoginTimeout	Yes	Yes[1]
getLogWriter	Yes	Yes
setLoginTimeout	Yes[2]	Yes[1]
setLogWriter	Yes	Yes

Table 43. DB2 JDBC support for DataSource methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM DB2 Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 44. DB2 JDBC support for DataTruncation methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Throwable	Yes	Yes
Methods inherited from java.sql.SQLException	Yes	Yes
Methods inherited from java.sql.SQLWarning	Yes	Yes
getDataSize	Yes	Yes
getIndex	Yes	Yes
getParameter	Yes	Yes
getRead	Yes	Yes
getTransferSize	Yes	Yes

Table 45. DB2 JDBC support for Driver methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
acceptsURL	Yes	Yes
connect	Yes	Yes
getMajorVersion	Yes	Yes
getMinorVersion	Yes	Yes
getPropertyInfo	Yes	Yes
jdbcCompliant	Yes	Yes

Table 46. DB2 JDBC support for DriverManager methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
deregisterDriver	Yes	Yes
getConnection	Yes	Yes
getDriver	Yes	Yes
getDrivers	Yes	Yes
getLoginTimeout	Yes	Yes[1]
getLogStream	Yes	Yes
getLogWriter	Yes	Yes
println	Yes	Yes
registerDriver	Yes	Yes

Table 46. DB2 JDBC support for DriverManager methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
setLoginTimeout	Yes[2]	Yes[1]
setLogStream	Yes	Yes
setLogWriter	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for IBM DB2 Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS.

Table 47. DB2 JDBC support for ParameterMetaData methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getParameterClassName	No	No
getParameterCount	Yes	No
getParameterMode	Yes	No
getParameterType	Yes	No
getParameterTypeName	Yes	No
getPrecision	Yes	No
getScale	Yes	No
isNullable	Yes	No
isSigned	Yes	No

Table 48. DB2 JDBC support for PooledConnection methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
addConnectionEventListener	Yes	Yes
close	Yes	Yes
getConnection	Yes	Yes
removeConnectionEventListener	Yes	Yes

Table 49. DB2 JDBC support for PreparedStatement methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.sql.Statement	Yes	Yes
addBatch	Yes	Yes
clearParameters	Yes	Yes
execute	Yes	Yes
executeQuery	Yes	Yes
executeUpdate	Yes	Yes
getMetaData	Yes	Yes

Table 49. DB2 JDBC support for PreparedStatement methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getParameterMetaData	Yes	Yes
setArray	No	No
setAsciiStream	Yes	Yes
setBigDecimal	Yes	Yes
setBinaryStream	Yes	Yes
setBlob	Yes	Yes
setBoolean	Yes	Yes
setByte	Yes	Yes
setBytes	Yes	Yes
setCharacterStream	Yes	Yes
setClob	Yes	Yes
setDate	Yes[1]	Yes[1]
setDouble	Yes	Yes
setFloat	Yes	Yes
setInt	Yes	Yes
setLong	Yes	Yes
setNull	Yes[2]	Yes[2]
setObject	Yes	Yes
setRef	No	No
setShort	Yes	Yes
setString	Yes[3]	Yes[3]
setTime	Yes[1]	Yes[1]
setTimestamp	Yes[1]	Yes[1]
setUnicodeStream	Yes	Yes
setURL	Yes	Yes

Notes:

1. DB2 does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone before sending the value to DB2 if you specify a form of the setDate, setTime, or setTimestamp method that includes a java.util.Calendar parameter.
2. The following form of `setNull` is not supported:

   `setNull(int *parameterIndex*, int *jdbcType*, String *typeName*)`
3. setString is not supported if the column has the FOR BIT DATA attribute or the data type is BLOB.

Table 50. DB2 JDBC support for Ref methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
get BaseTypeName	No	No

Table 51. DB2 JDBC support for ResultSet methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
absolute	Yes	Yes
afterLast	Yes	Yes
beforeFirst	Yes	Yes
cancelRowUpdates	Yes	No
clearWarnings	Yes	Yes
close	Yes	Yes
deleteRow	Yes	No
findColumn	Yes	Yes
first	Yes	Yes
getArray	No	No
getAsciiStream	Yes	Yes
getBigDecimal	Yes	Yes
getBinaryStream	Yes[1]	Yes
getBlob	Yes	Yes
getBoolean	Yes	Yes
getByte	Yes	Yes
getBytes	Yes	Yes
getCharacterStream	Yes	Yes
getClob	Yes	Yes
getConcurrency	Yes	Yes
getCursorName	Yes	Yes
getDate	Yes[2]	Yes[2]
getDouble	Yes	Yes
getFetchDirection	Yes	Yes
getFetchSize	Yes	Yes
getFloat	Yes	Yes
getInt	Yes	Yes
getLong	Yes	Yes
getMetaData	Yes	Yes
getObject	Yes[3]	Yes[3]
getRef	No	No
getRow	Yes	Yes
getShort	Yes	Yes
getStatement	Yes	Yes
getString	Yes	Yes
getTime	Yes[2]	Yes[2]
getTimestamp	Yes[2]	Yes[2]
getType	Yes	Yes

Table 51. DB2 JDBC support for ResultSet methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getUnicodeStream	Yes	Yes
getURL	Yes	Yes
getWarnings	Yes	Yes
insertRow	No	No
isAfterLast	Yes	Yes
isBeforeFirst	Yes	Yes
isFirst	Yes	Yes
isLast	Yes	Yes
last	Yes	Yes
moveToCurrentRow	Yes	No
moveToInsertRow	No	No
next	Yes	Yes
previous	Yes	Yes
refreshRow	Yes	No
relative	Yes	Yes
rowDeleted	Yes	No
rowInserted	No	No
rowUpdated	Yes	No
setFetchDirection	Yes	Yes
setFetchSize	Yes	Yes
updateArray	No	No
updateAsciiStream	Yes	No
updateBigDecimal	Yes	No
updateBinaryStream	Yes	No
updateBlob	Yes	No
updateBoolean	Yes	No
updateByte	Yes	No
updateBytes	Yes	No
updateCharacterStream	Yes	No
updateClob	Yes	No
updateDate	Yes	No
updateDouble	Yes	No
updateFloat	Yes	No
updateInt	Yes	No
updateLong	Yes	No
updateNull	Yes	No
updateObject	Yes	No
updateRef	No	No

Table 51. DB2 JDBC support for ResultSet methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
updateRow	Yes	No
updateShort	Yes	No
updateString	Yes	No
updateTime	Yes	No
updateTimestamp	Yes	No
wasNull	Yes	Yes

Notes:

1. `getBinaryStream` is not supported for CLOB columns.
2. DB2 does no timezone adjustment for datetime values. The JDBC driver adjusts a value for the local timezone after retrieving the value from DB2 if you specify a form of the getDate, getTime, or getTimestamp method that includes a java.util.Calendar parameter.
3. The following form of the `getObject` method is not supported:

   ```
   getObject(int parameterIndex, java.util.Map  map)
   ```

Table 52. DB2 JDBC support for ResultSetMetaData methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getCatalogName	Yes	Yes
getColumnClassName	No	Yes
getColumnCount	Yes	Yes
getColumnDisplaySize	Yes	Yes
getColumnLabel	Yes	Yes
getColumnName	Yes	Yes
getColumnType	Yes	Yes
getColumnTypeName	Yes	Yes
getPrecision	Yes	Yes
getScale	Yes	Yes
getSchemaName	Yes	Yes
getTableName	Yes	Yes
isAutoIncrement	Yes	Yes
isCaseSensitive	Yes	Yes
isCurrency	Yes	Yes
isDefinitelyWritable	Yes	Yes
isNullable	Yes	Yes
isReadOnly	Yes	Yes
isSearchable	Yes	Yes
isSigned	Yes	Yes
isWritable	Yes	Yes

Table 53. DB2 JDBC support for SQLData methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getSQLTypeName	No	No
readSQL	No	No
writeSQL	No	No

Table 54. DB2 JDBC support for SQLException methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Exception	Yes	Yes
getSQLState	Yes	Yes
getErrorCode	Yes	Yes
getNextException	Yes	Yes
setNextException	Yes	Yes

Table 55. DB2 JDBC support for SQLInput methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
readArray	No	No
readAsciiStream	No	No
readBigDecimal	No	No
readBinaryStream	No	No
readBlob	No	No
readBoolean	No	No
readByte	No	No
readBytes	No	No
readCharacterStream	No	No
readClob	No	No
readDate	No	No
readDouble	No	No
readFloat	No	No
readInt	No	No
readLong	No	No
readObject	No	No
readRef	No	No
readShort	No	No
readString	No	No
readTime	No	No
readTimestamp	No	No
wasNull	No	No

Table 56. DB2 JDBC support for SQLOutput methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
writeArray	No	No
writeAsciiStream	No	No
writeBigDecimal	No	No
writeBinaryStream	No	No
writeBlob	No	No
writeBoolean	No	No
writeByte	No	No
writeBytes	No	No
writeCharacterStream	No	No
writeClob	No	No
writeDate	No	No
writeDouble	No	No
writeFloat	No	No
writeInt	No	No
writeLong	No	No
writeObject	No	No
writeRef	No	No
writeShort	No	No
writeString	No	No
writeStruct	No	No
writeTime	No	No
writeTimestamp	No	No

Table 57. DB2 JDBC support for Statement methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
addBatch	Yes	Yes
cancel	Yes[1,2]	Yes
clearBatch	Yes	Yes
clearWarnings	Yes	Yes
close	Yes	Yes
execute	Yes	Yes[3]
executeBatch	Yes	Yes
executeQuery	Yes	Yes
executeUpdate	Yes	Yes[3]
getConnection	Yes	Yes
getFetchDirection	Yes	Yes
getFetchSize	Yes	Yes

Table 57. DB2 JDBC support for Statement methods (continued)

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getGeneratedKeys	Yes	No
getMaxFieldSize	Yes	Yes
getMaxRows	Yes	Yes
getMoreResults	Yes	Yes[3]
getQueryTimeout	Yes[2]	Yes
getResultSet	Yes	Yes
getResultSetConcurrency	Yes	Yes
getResultSetHoldability	Yes	No
getResultSetType	Yes	Yes
getUpdateCount[4]	Yes	Yes
getWarnings	Yes	Yes
setCursorName	Yes	Yes
setEscapeProcessing	Yes	Yes
setFetchDirection	Yes	Yes
setFetchSize	Yes	Yes
setMaxFieldSize	Yes	Yes
setMaxRows	Yes	Yes
setQueryTimeout	Yes[5]	Yes

Notes:

1. With IBM DB2 Driver for JDBC and SQLJ type 4 connectivity, you can execute Statement.cancel() only if the database server supports the DRDA INTRDBRQS (interrupt relational database request) command. Only DB2 for z/OS servers at the Version 9.1 or later level have this support. Therefore, with IBM DB2 Driver for JDBC and SQLJ type 4 connectivity, you can execute Statement.cancel() only for connections to DB2 for z/OS at Version 9 or later.
2. This method is supported only for:
   - IBM DB2 Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server at Version 9.1 or later
   - IBM DB2 Driver for JDBC and SQLJ type 4 connectivity to a DB2 for z/OS server at 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 IBM DB2 Driver for JDBC and SQLJ type 2 connectivity on DB2 for z/OS, this method is supported only for a *seconds* value of 0.

Table 58. DB2 JDBC support for Struct methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getSQLTypeName	No	No
getAttributes	No	No

Table 59. DB2 JDBC support for XAConnection methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from javax.sql.PooledConnection	Yes[1]	Yes
getXAResource	Yes[1]	Yes

Notes:

1. This method is supported for IBM DB2 Driver for JDBC and SQLJ type 2 connectivity to a DB2 Database for Linux, UNIX, and Windows server or IBM DB2 Driver for JDBC and SQLJ type 4 connectivity to a DB2 for z/OS server.

Table 60. DB2 JDBC support for XADataSource methods

JDBC method	IBM DB2 Driver for JDBC and SQLJ support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getLoginTimeout	Yes	Yes
getLogWriter	Yes	Yes
getXAConnection	Yes	Yes
setLoginTimeout	Yes	Yes
setLogWriter	Yes	Yes

Related reference:

- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

SQLJ statement reference

The topics that follow contain information about the syntax of SQLJ clauses.

- "SQLJ clause"
- "SQLJ host-expression" on page 266
- "SQLJ implements-clause" on page 266
- "SQLJ with-clause" on page 267
- "SQLJ connection-declaration-clause" on page 269
- "SQLJ iterator-declaration-clause" on page 269
- "SQLJ executable-clause" on page 271
- "SQLJ context-clause" on page 272
- "SQLJ statement-clause" on page 272
- "SQLJ SET-TRANSACTION-clause" on page 274
- "SQLJ assignment-clause" on page 275
- "SQLJ iterator-conversion-clause" on page 275

SQLJ clause

The SQL statements in an SQLJ program are in SQLJ clauses. The general syntax of an SQLJ clause is:

```
►►—#sql——┬—connection-declaration-clause—┬—;——————————————————————————————————►◄
         ├—iterator-declaration-clause———┤
         └—executable-clause—————————————┘
```

Keywords in an SQLJ clause are case sensitive, unless those keywords are part of an SQL statement in an executable clause.

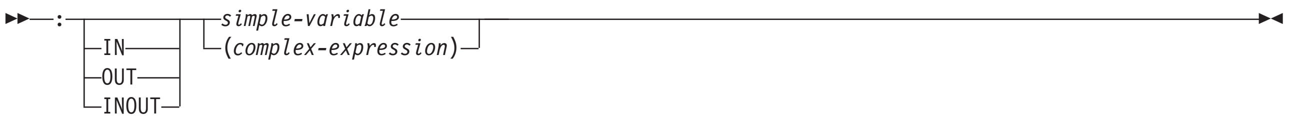
Related reference:

- "SQLJ connection-declaration-clause" on page 269
- "SQLJ executable-clause" on page 271
- "SQLJ iterator-declaration-clause" on page 269

SQLJ host-expression

A host expression is a Java variable or expression that is referenced by SQLJ clauses in an SQLJ application program.

Syntax:



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.

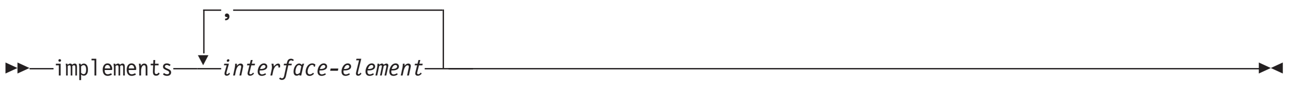
Related concepts:

- "Variables in SQLJ applications" on page 98

SQLJ implements-clause

The implements clause derives one or more classes from a Java interface.

Syntax:



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.

Related tasks:
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 101
- "Using scrollable iterators in an SQLJ application" on page 118

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:

```
►►──┬──┬─holdability=true───┬───────────────────────────────┬─────────────────────────►◄
    │  └─holdability=false──┘                               │
    │ ┌─sensitivity=ASENSITIVE─────────────────────────────┐│
    ├─┼─sensitivity=INSENSITIVE────────────────────────────┤┤
    │ └─sensitivity=SENSITIVE──┬─────────────────────────┬─┘│
    │                          │     ┌─dynamic=false─┐   │  │
    │                          └─,───┴─dynamic=true──┴───┘  │
    │                    ┌─,──────────────┐                 │
    │                    ▼                │                 │
    ├─updateColumns=─"────column-name─────┴──"──────────────┤
    ├─Java-ID=Java-constant-expression──────────────────────┤
    └─dataSource=─"──logical-datasource-name──"─────────────┘
```

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.

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.

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 DB2 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.

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 127

Related tasks:

- “Connecting to a data source using SQLJ” on page 92
- “Performing positioned UPDATE and DELETE operations in an SQLJ application” on page 101
- “Using scrollable iterators in an SQLJ application” on page 118

SQLJ connection-declaration-clause

The connection declaration clause declares a connection to a data source in an SQLJ application program.

Syntax:

```
►►─┬────────────────┬─context─Java-class-name─┬───────────────────┬─┬─────────────┬──────────►◄
   └─Java-modifiers─┘                         └─implements-clause─┘ └─with-clause─┘
```

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.

Related tasks:

- “Connecting to a data source using SQLJ” on page 92

Related reference:

- “SQLJ implements-clause” on page 266
- “SQLJ with-clause” on page 267

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:

```
>>─┬────────────────┬─iterator─Java-class-name─┬───────────────────┬─┬─────────────┬──────────────>
   └─Java-modifiers─┘                          └─implements-clause─┘ └─with-clause─┘

>─(─┬─positioned-iterator-column-declarations─┬─)──────────────────────────────────────────────><
    └─named-iterator-column-declarations──────┘
```

positioned-iterator-column declarations:

```
     ┌─,──────────────┐
     ▼                │
>>───Java-data-type───┴─────────────────────────────────────────────────────────────────────────><
```

named-iterator-column-declarations:

```
     ┌─,──────────────────────┐
     ▼                        │
>>───Java-data-type──Java-ID──┴─────────────────────────────────────────────────────────────────><
```

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.

Related concepts:

- "How an SQLJ application retrieves data from DB2 tables" on page 111

Related tasks:

- "Using a named iterator in an SQLJ application" on page 112
- "Using a positioned iterator in an SQLJ application" on page 114
- "Using scrollable iterators in an SQLJ application" on page 118

Related reference:

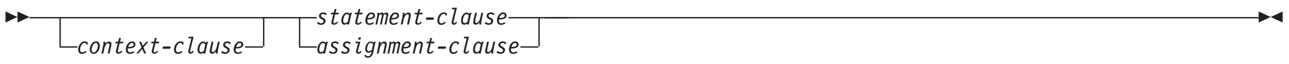
- "SQLJ implements-clause" on page 266
- "SQLJ with-clause" on page 267

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:



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.

Related reference:

- "SQLJ assignment-clause" on page 275
- "SQLJ context-clause" on page 272
- "SQLJ statement-clause" on page 272

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:

```
►►─[──┬─connection-context──────────────────────┬─]─────────────────────────────────────►◄
      ├─execution-context───────────────────────┤
      └─connection-context──,──execution 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.

Related tasks:

- "Connecting to a data source using SQLJ" on page 92
- "Controlling the execution of SQL statements in SQLJ" on page 130

SQLJ statement-clause

A statement clause contains an SQL statement or a SET TRANSACTION clause.

Syntax:

```
►►─{──┬─SQL-statement──────────┬─}──────────────────────────────────────────────────────►◄
      └─SET-TRANSACTION-clause─┘
```

Description:

SQL-statement
: You can include the DB2 Database for Linux, UNIX, and Windows SQL statements in Table 61 on page 273 in a statement clause.

SET-TRANSACTION-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. See SQLJ SET-TRANSACTION-clause for more information.

Table 61. Valid SQL statements in an SQLJ statement clause

ALTER DATABASE
ALTER FUNCTION
ALTER INDEX
ALTER PROCEDURE
ALTER STOGROUP
ALTER TABLE
ALTER TABLESPACE
CALL
COMMENT ON
COMMIT
CREATE ALIAS
CREATE DATABASE
CREATE DISTINCT TYPE
CREATE FUNCTION
CREATE GLOBAL TEMPORARY TABLE
CREATE INDEX
CREATE PROCEDURE
CREATE STOGROUP
CREATE SYNONYM
CREATE TABLE
CREATE TABLESPACE
CREATE TRIGGER
CREATE VIEW
DECLARE GLOBAL TEMPORARY TABLE
DELETE
DROP ALIAS
DROP DATABASE
DROP DISTINCT TYPE
DROP FUNCTION
DROP INDEX
DROP PACKAGE
DROP PROCEDURE
DROP STOGROUP
DROP SYNONYM
DROP TABLE
DROP TABLESPACE
DROP TRIGGER
DROP VIEW
FETCH
GRANT
INSERT
LOCK TABLE
MERGE
REVOKE
ROLLBACK
SAVEPOINT
SELECT INTO
SET CURRENT DEFAULT TRANSFORM GROUP
SET CURRENT DEGREE
SET CURRENT EXPLAIN MODE
SET CURRENT EXPLAIN SNAPSHOT
SET CURRENT ISOLATION
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
SET CURRENT OPTIMIZATION HINT
SET CURRENT PACKAGESET (USER is not supported)

Table 61. Valid SQL statements in an SQLJ statement clause (continued)

SET CURRENT PRECISION
SET CURRENT QUERY OPTIMIZATION
SET CURRENT REFRESH AGE
SET CURRENT SCHEMA
SET PATH
UPDATE

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.

Related tasks:

- "Setting the isolation level for an SQLJ transaction" on page 138

Related reference:

- "SQLJ SET-TRANSACTION-clause" on page 274

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.

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 127

Related reference:

- "SQLJ iterator-conversion-clause" on page 275
- "Fullselect" in *SQL Reference, Volume 1*
- "Select-statement" in *SQL Reference, Volume 1*

SQLJ iterator-conversion-clause

The iterator conversion clause converts a JDBC `ResultSet` to an iterator.

Syntax:

```
►►──CAST──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.

Related concepts:

- "Using SQLJ and JDBC in the same application" on page 127

sqlj.runtime reference

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.

Summary of interfaces and classes in the sqlj.runtime package

Table 62 summarizes the interfaces in sqlj.runtime.

Table 62. Summary of sqlj.runtime interfaces

Interface name	Purpose
ConnectionContext	Manages the SQL operations that are performed during a connection to a data source.
ForUpdate	Implemented by iterators that are used in a positioned UPDATE or DELETE statement.
NamedIterator	Implemented by iterators that are declared as named iterators.
PositionedIterator	Implemented by iterators that are declared as positioned iterators.
ResultSetIterator	Implemented by all iterators to allow query results to be processed using a JDBC `ResultSet`.
Scrollable	Provides a set of methods for manipulating scrollable iterators.

Table 63 summarizes the classes in sqlj.runtime.

Table 63. Summary of sqlj.runtime classes

Class name	Purpose
AsciiStream	A class for handling an input stream whose bytes should be interpreted as ASCII.

Table 63. Summary of sqlj.runtime classes (continued)

Class name	Purpose
BinaryStream	A class for handling an input stream whose bytes should be interpreted as binary.
CharacterStream	A class for handling an input stream whose bytes should be interpreted as Character.
DefaultRuntime	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.
ExecutionContext	Implemented when an SQLJ execution context is declared, to control the execution of SQL operations.
RuntimeContext	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.
SQLNullException	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.
StreamWrapper	Wraps a `java.io.InputStream` instance.
UnicodeStream	A class for handling an input stream whose bytes should be interpreted as Unicode.

Related reference:

- "sqlj.runtime.CharacterStream class" on page 290
- "sqlj.runtime.SQLNullException class" on page 298
- "sqlj.runtime.Scrollable interface" on page 286
- "sqlj.runtime.AsciiStream class" on page 288
- "sqlj.runtime.BinaryStream class" on page 289
- "sqlj.runtime.ConnectionContext interface" on page 277
- "sqlj.runtime.ExecutionContext class" on page 291
- "sqlj.runtime.ForUpdate interface" on page 282
- "sqlj.runtime.NamedIterator interface" on page 282
- "sqlj.runtime.ResultSetIterator interface" on page 283
- "sqlj.runtime.UnicodeStream class" on page 300
- "sqlj.runtime.PositionedIterator interface" on page 283

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 that are defined for the interface:

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.

getConnectedProfile

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.

getConnection

Format:

```
public abstract Connection getConnection()
```

Returns the underlying JDBC `Connection` object for the given connection context object.

getExecutionContext

Format:

```
public abstract ExecutionContext getExecutionContect()
```

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 in a concrete implementation of the ConnectionContext interface that results from translation of the statement #sql context Ctx;:

Ctx(String, boolean)
Format:

```
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:

```
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:

```
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 DB2 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.

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.

Constructors 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
```

Ctx(String, String)
Format:

```
public Ctx(String user, String password,
)
 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.

Additional methods that are generated in a concrete implementation of the `ConnectionContext` interface that results from translation of the statement `#sql context Ctx;`:

getDefaultContext
Format:

```
public static Ctx getDefaultContext()
```

Returns the default connection context object for the Ctx class.

getProfileKey
Format:

```
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:

```
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:

```
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:

```
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.

Related tasks:
- "Closing the connection to a data source in an SQLJ application" on page 140
- "Connecting to a data source using SQLJ" on page 92

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.

Related concepts:
- "Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application" on page 106

Related tasks:
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 101

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.

Related tasks:

- "Using a named iterator in an SQLJ application" on page 112

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.

Related tasks:

- "Using a positioned iterator in an SQLJ application" on page 114

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.

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.

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.

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.

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.

Related tasks:

- "Using a named iterator in an SQLJ application" on page 112
- "Using a positioned iterator in an SQLJ application" on page 114

Related reference:

- "SQLJ iterator-declaration-clause" on page 269

sqlj.runtime.Scrollable interface

`sqlj.runtime.Scrollable` is implemented when a scrollable iterator is declared. `sqlj.runtime.Scrollable` provides methods to move around in the result table and to check the position in the result table.

absolute(int)

Format:

```
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:

```
public abstract void afterLast() throws SQLException
```

Moves the iterator after the last row of the result table.

beforeFirst()

Format:

```
public abstract void beforeFirst() throws SQLException
```

Moves the iterator before the first row of the result table.

first()

Format:

```
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:

```
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:

```
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 throw, 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.

Related tasks:
- "Using scrollable iterators in an SQLJ application" on page 118

sqlj.runtime.AsciiStream class

The `sqlj.runtime.AsciiStream` class is for an input stream of ASCII data with a specified length. The `sqlj.runtime.AsciiStream` class is derived from the `java.io.InputStream` class, and extends the `sqlj.runtime.StreamWrapper` class. SQLJ interprets the bytes in an `sqlj.runtime.AsciiStream` object as ASCII characters. An InputStream object with ASCII characters needs to be passed as a `sqlj.runtime.AsciiStream` object.

Constructors:

AsciiStream(InputStream)

Format:

```
public AsciiStream(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 `AsciiStream` object.

AsciiStream(InputStream, int)

Format:

```
public AsciiStream(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 `AsciiStream` object.

length
: The length of the `InputStream` object that SQLJ interprets as an `AsciiStream` 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 an `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 an `BinaryStream` object.

length
The length of the `InputStream` object that SQLJ interprets as an `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 an `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 an `CharacterStream` object.

length
The length of the `Reader` object that SQLJ interprets as an `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. Use an execution context 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
```

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:

```
public synchronized 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:

```
public RTResultSet 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:

```
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:

```
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
The current batch limit.

getBatchUpdateCounts

Format:

```
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 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:

```
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:

```
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:

```
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:

```
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:

```
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:

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.

setMaxFieldSize

Format:

```
public void setMaxFieldSize(int max-bytes)
```

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:

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.

setMaxRows

Format:

```
public synchronized void setMaxRows(int max-rows)
```

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:

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.

setQueryTimeout

Format:

```
public synchronized void setQueryTimeout(int timeout-value)
```

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:

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.

Related tasks:

- "Controlling the execution of SQL statements in SQLJ" on page 130

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'.

Related reference:

- "Data types that map to SQL data types in JDBC applications" on page 227

sqlj.runtime.StreamWrapper class

The `sqlj.runtime.StreamWrapper` class wraps a `java.io.InputStream` instance and extends the `java.io.InputStream` class. The `sqlj.runtime.AsciiStream`, `sqlj.runtime.BinaryStream`, and `sqlj.runtime.UnicodeStream` classes extend `sqlj.runtime.StreamWrapper`. `sqlj.runtime.StreamWrapper` supports methods for specifying the length of `sqlj.runtime.AsciiStream`, `sqlj.runtime.BinaryStream`, and `sqlj.runtime.UnicodeStream` objects.

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.

Related reference:

- "sqlj.runtime.AsciiStream class" on page 288
- "sqlj.runtime.BinaryStream class" on page 289
- "sqlj.runtime.UnicodeStream class" on page 300

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 an `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 DB2 Driver for JDBC and SQLJ reference information

The topics that follow contain information that is specific to the IBM DB2 Driver for JDBC and SQLJ.

- "DB2-only classes and interfaces" on page 301
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

- "SQLJ differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 342
- "Error codes issued by the IBM DB2 Driver for JDBC and SQLJ" on page 344
- "SQLSTATEs issued by the IBM DB2 Driver for JDBC and SQLJ" on page 345
- "How to find IBM DB2 Driver for JDBC and SQLJ version and environment information" on page 346

DB2-only classes and interfaces

The following topics discuss classes and interfaces that are defined only by the IBM DB2 Driver for JDBC and SQLJ.

Summary of IBM DB2 Driver for JDBC and SQLJ extensions to JDBC

The IBM DB2 Driver for JDBC and SQLJ provides a set of extensions to the support that is provided by the JDBC specification.

To use DB2-only methods in classes that have corresponding, standard classes, cast an instance of the related, standard JDBC class to an instance of the DB2-only class. For example:

```
javax.sql.DataSource ds =
 new com.ibm.db2.jcc.DB2SimpleDataSource();
((com.ibm.db2.jcc.DB2BaseDataSource) ds).setServerName("sysmvs1.stl.ibm.com");
```

Table 64 summarizes the DB2-only interfaces.

Table 64. Summary of DB2-only interfaces provided by the IBM DB2 Driver for JDBC and SQLJ

Interface name	Purpose
DB2Connection	Extends the `java.sql.Connection` interface.
DB2DatabaseMetaData	Extends the `java.sql.DatabaseMetaData` interface.
DB2Diagnosable	Provides a mechanism for getting DB2 diagnostics from a DB2 `SQLException`.
DB2PreparedStatement	Extends the `com.ibm.db2.jcc.DB2Statement` and `java.sql.PreparedStatement` interfaces.
DB2RowID	Used for declaring Java objects for use with the DB2 ROWID data type.
DB2Statement	Extends the `java.sql.Statement` interface.
DB2SystemMonitor	Used for collecting system monitoring data for a connection.
DB2Xml	Used for updating data in XML columns and retrieving data from XML columns.

Table 65 summarizes the DB2-only classes.

Table 65. Summary of DB2-only classes provided by the IBM DB2 Driver for JDBC and SQLJ

Class name	Purpose
DB2BaseDataSource	The abstract data source parent class for all DB2-specific implementations of `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, and `javax.sql.XADataSource`.
DB2JCCPlugin	The abstract class for implementation of JDBC security plug-ins.
DB2ClientRerouteServerList	Implements the `java.io.Serializable` and `javax.naming.Referenceable` interfaces.
DB2ConnectionPoolDataSource	A factory for `PooledConnection` objects.
DB2ExceptionFormatter	Contains methods for printing diagnostic information to a stream.

Table 65. Summary of DB2-only classes provided by the IBM DB2 Driver for JDBC and SQLJ (continued)

Class name	Purpose
DB2PooledConnection	Provides methods that an application server can use to switch users on a preexisting trusted connection.
DB2SimpleDataSource	Extends the `DataBaseDataSource` class. Does not support connection pooling or distributed transactions.
DB2Sqlca	An encapsulation of the DB2 SQLCA.

Related reference:
- "SQLCA (SQL communications area)" in *SQL Reference, Volume 1*
- "DB2Sqlca class" on page 326
- "DB2BaseDataSource class" on page 302
- "DB2ClientRerouteServerList class" on page 305
- "DB2Connection interface" on page 306
- "DB2ConnectionPoolDataSource class" on page 318
- "DB2Diagnosable interface" on page 320
- "DB2ExceptionFormatter class" on page 320
- "DB2PooledConnection class" on page 322
- "DB2PreparedStatement interface" on page 325
- "DB2RowID interface" on page 325
- "DB2SimpleDataSource class" on page 325
- "DB2Statement interface" on page 327
- "DB2SystemMonitor interface" on page 328
- "DB2Xml interface" on page 333

DB2BaseDataSource class

The `com.ibm.db2.jcc.DB2BaseDataSource` class is the abstract data source parent class for all DB2-specific implementations of `javax.sql.DataSource`, `javax.sql.ConnectionPoolDataSource`, and `javax.sql.XADataSource`.

DB2BaseDataSource properties:

The following properties are defined only for the IBM DB2 Driver for JDBC and SQLJ. See Properties for the IBM DB2 Driver for JDBC and SQLJ for explanations of these properties.

Each of these properties has a `setXXX` method to set the value of the property and a `getXXX` method to retrieve the value. A `setXXX` method has this form:

```
void setProperty-name(data-type property-value)
```

A `getXXX` method has this form:

```
data-type getProperty-name()
```

Property-name is the unqualified property name, with the first character capitalized.

Table 66 on page 303 lists the IBM DB2 Driver for JDBC and SQLJ properties and their data types.

Table 66. DB2BaseDataSource properties and their data types

Property name	Data type
com.ibm.db2.jcc.DB2BaseDataSource.accountingInterval (DB2 for z/OS only)	String
com.ibm.db2.jcc.DB2BaseDataSource.blockingReadConnectionTimeout	int
com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation	String
com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation	String
com.ibm.db2.jcc.DB2BaseDataSource.clientDebugInfo	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramId	String
com.ibm.db2.jcc.DB2BaseDataSource.clientProgramName	String
com.ibm.db2.jcc.DB2BaseDataSource.clientRerouteServerListJNDIName	String
com.ibm.db2.jcc.DB2BaseDataSource.clientUser	String
com.ibm.db2.jcc.DB2BaseDataSource.clientWorkstation	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainMode	String
com.ibm.db2.jcc.DB2BaseDataSource.currentExplainSnapshot	String
com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath	String
com.ibm.db2.jcc.DB2BaseDataSource.currentLockTimeout (DB2 for z/OS only)	int
com.ibm.db2.jcc.DB2BaseDataSource.currentMaintainedTableTypesForOptimization	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath	String
com.ibm.db2.jcc.DB2BaseDataSource.currentPackageSet	String
com.ibm.db2.jcc.DB2BaseDataSource.currentQueryOptimization	int
com.ibm.db2.jcc.DB2BaseDataSource.currentRefreshAge	long
com.ibm.db2.jcc.DB2BaseDataSource.cursorSensitivity	int
com.ibm.db2.jcc.DB2BaseDataSource.currentSchema	String
com.ibm.db2.jcc.DB2BaseDataSource.currentSQLID	String
com.ibm.db2.jcc.DB2BaseDataSource.databaseName	String
com.ibm.db2.jcc.DB2BaseDataSource.deferPrepares	boolean
com.ibm.db2.jcc.DB2BaseDataSource.description	String
com.ibm.db2.jcc.DB2BaseDataSource.driverType	int
com.ibm.db2.jcc.DB2BaseDataSource.enableConnectionConcentrator	boolean
com.ibm.db2.jcc.DB2BaseDataSource.enableSysplexWLB	boolean
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeInputStreams	boolean
com.ibm.db2.jcc.DB2BaseDataSource.fullyMaterializeLobData	boolean
com.ibm.db2.jcc.DB2BaseDataSource.gssCredential	Object
com.ibm.db2.jcc.DB2BaseDataSource.jdbcCollection	String
com.ibm.db2.jcc.DB2BaseDataSource.keepDynamic	int
com.ibm.db2.jcc.DB2BaseDataSource.kerberosServerPrincipal	String
com.ibm.db2.jcc.DB2BaseDataSource.loginTimeout	int
com.ibm.db2.jcc.DB2BaseDataSource.logWriter	PrintWriter
com.ibm.db2.jcc.DB2BaseDataSource.maxRetriesForClientReroute	int
com.ibm.db2.jcc.DB2BaseDataSource.maxTransportObjects	int
com.ibm.db2.jcc.DB2BaseDataSource.pkList	String
com.ibm.db2.jcc.DB2BaseDataSource.planName	String

Table 66. DB2BaseDataSource properties and their data types (continued)

Property name	Data type
com.ibm.db2.jcc.DB2BaseDataSource.plugin (DB2 Database for Linux, UNIX, and Windows only)	Object
com.ibm.db2.jcc.DB2BaseDataSource.pluginName (DB2 Database for Linux, UNIX, and Windows only)	String
com.ibm.db2.jcc.DB2BaseDataSource.portNumber	int
com.ibm.db2.jcc.DB2BaseDataSource.progressiveStreaming	int
com.ibm.db2.jcc.DB2BaseDataSource.queryCloseImplicit	int
com.ibm.db2.jcc.DB2BaseDataSource.readOnly	boolean
com.ibm.db2.jcc.DB2BaseDataSource.resultSetHoldability	int
com.ibm.db2.jcc.DB2BaseDataSource.retrieveMessagesFromServerOnGetMessage	boolean
com.ibm.db2.jcc.DB2BaseDataSource.retryIntervalForClientReroute	int
com.ibm.db2.jcc.DB2BaseDataSource.returnAlias	short
com.ibm.db2.jcc.DB2BaseDataSource.securityMechanism	int
com.ibm.db2.jcc.DB2BaseDataSource.sendDataAsIs	boolean
com.ibm.db2.jcc.DB2BaseDataSource.serverName	String
com.ibm.db2.jcc.DB2BaseDataSource.sslConnection	boolean
com.ibm.db2.jcc.DB2BaseDataSource.streamBufferSize	int
com.ibm.db2.jcc.DB2BaseDataSource.supportsAsynchronousXARollback	int
com.ibm.db2.jcc.DB2BaseDataSource.sysSchema	String
com.ibm.db2.jcc.DB2BaseDataSource.traceDirectory	String
com.ibm.db2.jcc.DB2BaseDataSource.traceFile	String
com.ibm.db2.jcc.DB2BaseDataSource.traceLevel	int
com.ibm.db2.jcc.DB2BaseDataSource.useCachedCursor	boolean
com.ibm.db2.jcc.DB2BaseDataSource.usePool	boolean
com.ibm.db2.jcc.DB2BaseDataSource.user	String
com.ibm.db2.jcc.DB2BaseDataSource.useTargetColumnEncoding	boolean
com.ibm.db2.jcc.DB2BaseDataSource.useTransactionRedirect	boolean

DB2BaseDataSource methods:

In addition to the `getXXX` and `setXXX` methods for the DB2BaseDataSource properties, the following methods are defined only for the IBM DB2 Driver for JDBC and SQLJ.

getReference

Format:

```
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`

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

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 DB2 database servers.

getAlternateServerName

Format:

```
public String[] getAlternateServerName()
```

Retrieves an array that contains the names of the alternate DB2 database 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 DB2 database server.

getPrimaryServerName

Format:

```
public String[] getPrimaryServerName()
```

Retrieves the name of the primary DB2 database 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 DB2 database servers.

setAlternateServerName

Format:

```
public void setAlternateServerName(String[] alternateServer)
```

Sets the alternate server names for DB2 database 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 DB2 database server.

setPrimaryServerName

Format:

```
public void setPrimaryServerName(String primaryServer)
```

Sets the primary server name for a DB2 database server. This value is an IP address or a DNS server name.

Related concepts:

- "IBM DB2 Driver for JDBC and SQLJ client reroute support" on page 86

DB2Connection interface

The `com.ibm.db2.jcc.DB2Connection` interface extends the `java.sql.Connection` interface.

DB2Connection methods:

The following methods are defined only for the IBM DB2 Driver for JDBC and SQLJ.

changeDB2Password

Format:

```
public abstract void changeDB2Password(String oldPassword,
  String newPassword)
  throws java.sql.SQLException
```

Changes the password for accessing the DB2 database server, for the user of the `Connection` object.

Parameter descriptions:

oldPassword
The original password for the `Connection`.

newPassword
The new password for the `Connection`.

deregisterDB2XmlObject

Formats:

```
public void deregisterDB2XmlObject(String sqlIdSchema,
  String sqlIdName)
  throws SQLException
```

Removes a previously registered XML schema from DB2. `deregisterDB2XmlObject` calls the SYSPROC.XSR_REMOVE stored procedure to remove the XML schema.

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, DB2 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, DB2 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 database server.

getDB2ClientAccountingInformation

Format:

```
public String getDB2ClientAccountingInformation()
  throws SQLException
```

Returns accounting information for the current client.

getDB2ClientApplicationInformation

Format:

```
public String getDB2ClientApplicationInformation()
  throws java.sql.SQLException
```

Returns application information for the current client.

getDB2ClientUser

Format:

```
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.

getDB2ClientWorkstation

Format:

```
public String getDB2ClientWorkstation()
  throws java.sql.SQLException
```

Returns current client workstation name for the current client.

getDB2Correlator

Format:

```
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:

```
public String getDB2CurrentPackagePath()
  throws java.sql.SQLException
```

Returns the list of DB2 package collections that are searched for JDBC and SQLJ packages.

getDB2CurrentPackageSet

Format:

```
public String getDB2CurrentPackageSet()
  throws java.sql.SQLException
```

Returns the collection ID for the connection.

getDB2SystemMonitor

Format:

```
public abstract DB2SystemMonitor getDB2SystemMonitor()
  throws java.sql.SQLException
```

Returns the system monitor object for the connection. Each IBM DB2 Driver for JDBC and SQLJ connection can have a single system monitor.

getJccLogWriter

Format:

```
public PrintWriter getJccLogWriter()
  throws java.sql.SQLException
```

Returns the current trace destination for the IBM DB2 Driver for JDBC and SQLJ trace.

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 DB2 catalog for that server.

Parameter descriptions:

jarFile

The contents of the JAR file that is to be defined to the DB2 server.

jarFileLength

The length of the JAR file that is to be defined to the DB2 server.

jarId

The DB2 name of the JAR, 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*, DB2 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.

isDB2Alive

Format:

```
public boolean DB2Connection.isDB2Alive()
  throws java.sql.SQLException
```

Returns `true` if the socket for a connection to a DB2 server is still active.

isDB2GatewayConnection

Format:

```
public boolean DB2Connection.isDB2GatewayConnection()
  throws java.sql.SQLException
```

Returns `true` if the connection to the server goes through an intermediate DB2 Connect gateway. Returns `false` otherwise.

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.

Parameter descriptions:

properties
New properties for the connection. These properties override any properties that are already defined on the `DB2Connection` instance.

recycleDB2Connection

Format:

```
public void recycleDB2Connection()
  throws SQLException
```

Notifies the underlying physical connection of a recycle event. `Statement` objects on the `Connection` are closed or recycled for reuse, depending on pool configuration settings.

registerDB2XmlSchema

Formats:

```
public void registerDB2XmlSchema(String[] sqlIdSchema,
  String[] sqlIdName,
  String[] xmlSchemaLocations,
  InputStream[] xmlSchemaDocuments,
  int[] xmlSchemaDocumentsLengths,
  InputStream[] xmlSchemaDocumentsProperties,
  int[] xmlSchemaDocumentsPropertiesLengths,
  InputStream xmlSchemaProperties,
  int xmlSchemaPropertiesLength,
  boolean isUsedForShredding)
  throws SQLException
public void registerDB2XmlSchema(String[] sqlIdSchema,
  String[] sqlIdName,
  String[] xmlSchemaLocations,
  String[] xmlSchemaDocuments,
  String[] xmlSchemaDocumentsProperties,
  String xmlSchemaProperties,
  boolean isUsedForShredding)
  throws SQLException
```

Provides one or more XML schema documents for registering an XML schema in DB2. `registerDB2XmlSchema` calls the SYSPROC.XSR_REGISTER, SYSPROC.XSR_ADDSCHEMADOC, and SYSPROC.XSR_COMPLETE stored procedures to register 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, DB2 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, DB2 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. DB2 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 30MB. 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. DB2 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 5MB. 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. DB2 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 5MB. 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.

removeDB2JavaStoredProcedure
Format:

```
public void DB2Connection.replaceDB2JavaStoredProcedure(
  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 DB2 catalog for that server.

Parameter descriptions:

jarId
The DB2 name of the JAR, 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*, DB2 uses the SQL authorization ID that is in the CURRENT SCHEMA special register.

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 DB2 catalog for that server.

Parameter descriptions:

jarFile
The contents of the JAR file that is to be replaced on the DB2 server.

jarFileLength
The length of the JAR file that is to be replace on the DB2 server.

jarId
The DB2 name of the JAR, 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*, DB2 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.

reuseDB2Connection (trusted connection reuse)
Formats:

```
public Connection reuseDB2Connection(byte[] cookie,
  String user,
  String password,
  String usernameRegistry,
  byte[] userSecToken,
  String originalUser,
  java.util.Properties properties)
  throws java.sql.SQLException
```

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 DB2 database server 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 DB2 user.

password
The password for *user*.

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 DB2 server 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 DB2Connection reuseDB2Connection(String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
```

```
public DB2Connection reuseDB2Connection(
  org.ietf.jgss.GSSCredential gssCredential,
  java.util.Properties properties)
  throws java.sql.SQLException
```

In a heterogeneous pooling environment, reuses 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 reuse without reauthentication)
Formats:

```
public DB2Connection reuseDB2Connection(java.util.Properties properties)
  throws java.sql.SQLException

public DB2Connection reuseDB2Connection()
  throws java.sql.SQLException
public DB2Connection reuseDB2Connection(int connectionReuseProtocol,
  java.util.Properties properties)
  throws java.sql.SQLException
public DB2Connection reuseDB2Connection(int connectionReuseProtocol)
  throws java.sql.SQLException
```

Reuses an existing Connection instance without reauthentication. The second and fourth forms of the method is intended for reuse of a `Connection` instance when the properties do not change. The third and fourth forms of `getDB2Connection` let you specify whether to reset the connection properties when the connection is reused.

Parameter description:

properties
Properties for the reused connection. These properties override any properties that are already defined on the `DB2Connection` instance.

connectionReuseProtocol
Specifies whether the connection state is reset when a connection is reused from a connection pool. This value overrides the `connectionReuseProtocol` property value. Possible values are:

DIRTY_CONNECTION_REUSE
The connection state is not reset when a `Connection` is reused from a connection pool. Special register settings are not reset and temporary tables are not dropped. Specified property settings (derived from an application `DataSource` or WebSphere Application Server resource reference) might be passed by the pool module to the JDBC driver for reinitialization. Properties that are not passed

by the pool module are not changed. All JDBC standard transient properties, such as isolation level, auto-commit mode, and read-only mode are reset to their JDBC defaults. These properties do not change:
- accountingInterval
- databaseName
- driverType
- pkList
- planName
- portNumber
- kerberosServerPrincipal
- password
- readOnly
- securityMechanism
- serverName
- user

RESET_CONNECTIONS_ON_REUSE
The connection state is reset when a `Connection` is reused from a connection pool. Special register settings are reset and temporary tables are dropped. Specified property settings (derived from an application `DataSource` or WebSphere Application Server resource reference) might be passed by the pool module to the JDBC driver for reinitialization. All JDBC standard transient properties, such as isolation level, auto-commit mode, and read-only mode are reset to their JDBC defaults. These properties do not change:
- accountingInterval
- databaseName
- driverType
- pkList
- planName
- portNumber
- kerberosServerPrincipal
- password
- readOnly
- securityMechanism
- serverName
- user

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.

setDB2ClientDebugInfo
Formats:

```
public void setDB2ClientDebugInformation(String debugInfo)
  throws java.sql.SQLException
public void setDB2ClientDebugInformation(String mgrInfo,
  String traceInfo)
  throws java.sql.SQLException
```

Sets a value for the DB2 CLIENT DEBUGINFO connection attribute, to notify the DB2 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. 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.

The setDB2ClientDebugInfo method applies only to connections to DB2 for z/OS database servers.

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,Llvl
```

The parts of the string are:

M*ip:port*
Session manager IP address and port number

I*ip* Client IP address

P*pid* Client process ID

T*tid* Client thread ID (optional)

C*id* Data connection generated ID

L*lvl* 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.

```
Llvl
```

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. 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:

```
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.

setDB2ClientWorkstation
Format:

```
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.

setDB2CurrentPackagePath

Format:

```
public void setDB2CurrentPackagePath(String packagePath)
  throws java.sql.SQLException
```

Specifies a list of collection IDs that the DB2 server searches for JDBC and SQLJ packages.

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 DB2 Driver for JDBC and SQLJ instance that is used for the connection.

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.

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 DB2 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 DB2 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 DB2 Driver for JDBC and SQLJ for valid values.

Related concepts:

- "JDBC and SQLJ problem diagnosis with the IBM DB2 Driver for JDBC and SQLJ" on page 181

Related tasks:

- "Providing extended client information to the DB2 server with the IBM DB2 Driver for JDBC and SQLJ" on page 66

DB2ConnectionPoolDataSource class

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` 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).

DB2ConnectionPoolDataSource properties:

These properties are defined only for the IBM DB2 Driver for JDBC and SQLJ. See Properties for the IBM DB2 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:

```
void setProperty-name(data-type property-value)
```

A `getXXX` method has this form:

```
data-type getProperty-name()
```

Property-name is the unqualified property name, with the first character capitalized.

Table 67 lists the IBM DB2 Driver for JDBC and SQLJ properties and their data types.

Table 67. DB2ConnectionPoolDataSource properties and their data types

Property name	Data type
com.ibm.db2.jcc.DB2ConnectionPoolDataSource.connectionReuseProtocol	int
com.ibm.db2.jcc.DB2ConnectionPoolDataSource.statementReuseProtocol	int

DB2ConnectionPoolDataSource methods:

getDB2PooledConnection

Formats:

```
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:

```
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. 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.

Related concepts:

- Chapter 9, "JDBC and SQLJ connection pooling support," on page 219

Related reference:

- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232

DB2Diagnosable interface

The `com.ibm.db2.jcc.DB2Diagnosable` interface provides a mechanism for getting DB2 diagnostics from a DB2 `SQLException`.

DB2Diagnosable methods:

The following methods are defined only for the IBM DB2 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 DB2 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 DB2 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 DB2 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.

Related tasks:

- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77
- "Handling SQL warnings in an SQLJ application" on page 139

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 DB2 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.

Related concepts:

- "Example of a trace program under the IBM DB2 Driver for JDBC and SQLJ" on page 184

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 DB2 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

Related concepts:

- "IBM DB2 Driver for JDBC and SQLJ security plugin support" on page 151

DB2PooledConnection class

The `com.ibm.db2.jcc.DB2PooledConnection` class provides methods that an application server can use to switch users on a preexisting trusted connection.

DB2PooledConnection methods:

The following methods are defined only for the IBM DB2 Driver for JDBC and SQLJ.

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.

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 DB2 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 DB2 user.

password
The password for *user*.

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 DB2 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:

```
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 DB2 to establish the authorization ID for the database server. *user* and *password* can be specified only under the following circumstances:

- When the connection pool is configured with the WebSphere Application Server resource reference authentication property *res-auth* is set to 1 (true).
- When the DB2ConnectionPoolDataSource.connectionReuseProtocol property is *not* set to DIRTY_CONNECTION_REUSE.

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 reuse without reauthentication)
Formats:

```
public java.sql.Connection getDB2Connection(
  java.util.Properties properties)
  throws java.sql.SQLException
public DB2Connection getDB2Connection(int connectionReuseProtocol,
  java.util.Properties properties)
  throws java.sql.SQLException
```

Reuses a untrusted connection, without reauthentication.

The second form of `getDB2Connection` lets you specify whether to reset the connection properties when the connection is reused.

Parameter descriptions:

properties
Properties for the reused connection. These properties override any properties that are already defined on the `DB2PooledConnection` instance.

connectionReuseProtocol
Specifies whether the connection state is reset when a connection is reused from a connection pool. This value overrides the `connectionReuseProtocol` property value. Possible values are:

DIRTY_CONNECTION_REUSE
The connection state is not reset when a `Connection` is reused from a connection pool. Special register settings are not reset and temporary tables are not dropped. Specified property settings (derived from an application `DataSource` or WebSphere Application Server resource reference) might be passed by the pool module to the JDBC driver for reinitialization. Properties that are not passed by the pool module are not changed. All JDBC standard transient properties, such as isolation level, auto-commit mode, and read-only mode are reset to their JDBC defaults. These properties do not change:

- accountingInterval
- databaseName
- driverType
- pkList
- planName
- portNumber
- kerberosServerPrincipal
- password
- readOnly
- securityMechanism
- serverName
- user

RESET_CONNECTIONS_ON_REUSE
The connection state is reset when a `Connection` is reused from a connection pool. Special register settings are reset and temporary tables are dropped. Specified property settings (derived from an application `DataSource` or WebSphere Application Server resource reference) might be passed by the pool module to the JDBC driver for reinitialization. All JDBC standard transient properties, such as isolation level, auto-commit mode, and read-only mode are reset to their JDBC defaults. These properties do not change:

- accountingInterval
- databaseName
- driverType
- pkList
- planName
- portNumber
- kerberosServerPrincipal
- password
- readOnly
- securityMechanism
- serverName
- user

recycleDB2Connection
Format:

```
public void recycleDB2Connection()
  throws SQLException
```

Notifies the underlying physical connection of a recycle event. `Statement` objects on the `Connection` are closed or recycled for reuse, depending on pool configuration settings. This method is used under a pooling model in which the pool module provides the logical connection wrapper.

Related concepts:
- Chapter 9, "JDBC and SQLJ connection pooling support," on page 219

Related reference:
- "DB2ConnectionPoolDataSource class" on page 318

DB2PreparedStatement interface

The `com.ibm.db2.jcc.DB2PreparedStatement` interface extends the `com.ibm.db2.jcc.DB2Statement` and `java.sql.PreparedStatement` interfaces.

DB2PreparedStatement methods:

The following methods are defined only for the IBM DB2 Driver for JDBC and SQLJ.

executeDB2QueryBatch
Format:

```
public void executeDB2QueryBatch()
  throws java.sql.SQLException
```

Executes a statement batch that contains queries with parameters.

Related tasks:
- "Making batch queries in JDBC applications" on page 48

DB2RowID interface

The `com.ibm.db2.jcc.DB2RowID` interface is used for declaring Java objects for use with the DB2 ROWID data type.

DB2RowID methods:

The following method is defined only for the IBM DB2 Driver for JDBC and SQLJ.

getBytes
Format:

```
public byte[] getBytes()
```

Converts a com.ibm.jcc.DB2RowID object to bytes.

Related concepts:
- "ROWIDs in JDBC with the IBM DB2 Driver for JDBC and SQLJ" on page 61
- "ROWIDs in SQLJ with the IBM DB2 Driver for JDBC and SQLJ" on page 130

DB2SimpleDataSource class

The `com.ibm.db2.jcc.DB2SimpleDataSource` class extends the `DataBaseDataSource` class. A `DataBaseDataSource` 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 DB2 Driver for JDBC and SQLJ-only properties.

DB2SimpleDataSource properties:

The following property is defined only for the IBM DB2 Driver for JDBC and SQLJ. See Properties for the IBM DB2 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 DB2 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.

Related tasks:

- "Connecting to a data source using the DataSource interface" on page 30
- "Creating and deploying DataSource objects" on page 33

DB2Sqlca class

The `com.ibm.db2.jcc.DB2Sqlca` class is an encapsulation of the DB2 SQLCA. .

DB2Sqlca methods:

The following methods are defined only for the IBM DB2 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()
```

Returns an array, each element of which contains an SQLCA SQLERRMC token.

getSqlErrd
Format:

```
public abstract int[] getSqlErrd()
```

Returns an array, each element of which contains an SQLCA SQLERRP value.

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.

Related tasks:
- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77
- "Handling SQL warnings in an SQLJ application" on page 139

DB2Statement interface

The `com.ibm.db2.jcc.DB2Statement` interface extends the `java.sql.Statement` interface.

DB2Statement methods:

The following methods are defined only for the IBM DB2 Driver for JDBC and SQLJ.

getDB2ClientProgramId
Format:

```
public String getDB2ClientProgramId()
  throws java.sql.SQLException
```

Returns the user-defined client program identifier for the connection, which is stored on the database server.

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. 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.

Related reference:

- "DB2PreparedStatement interface" on page 325

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 DB2 Driver for JDBC and SQLJ.

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 DB2 Driver for JDBC and SQLJ.

enable

Format:

```
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:

```
public long getApplicationTimeMillis()
  throws java.sql.SQLException
```

Returns the sum of the application, JDBC driver, network I/O, and DB2 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:

```
public long getCoreDriverTimeMicros()
  throws java.sql.SQLException
```

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 DB2 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 DB2 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:

```
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 DB2 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 DB2 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 DB2 server for low-priority SQL requests. Network I/O time intervals include DB2 server elapsed time.

`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 DB2 server elapsed times that were collected while system monitoring was enabled. The time is in microseconds.

The DB2 server reports elapsed times under these conditions:

- The server supports returning elapsed time data to the client.

  Currently, DB2 Database for Linux, UNIX, and Windows servers do not support this function.
- The server performs operations that can be monitored. For example, DB2 server elapsed time is not returned for commits or rollbacks.

DB2 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 server. Network time to receive or send the data stream is not included.

a DB2 server elapsed time interval is the difference, in microseconds, between these points in the server processing:

Interval beginning
When the operating system dispatches DB2 to process a TCP/IP message that is received from the JDBC driver.

Interval end
When DB2 is ready to issue the TCP/IP command to return the reply message to the client.

`getServerTimeMicros` returns 0 if system monitoring is disabled. Calling this method without first calling the `stop` method results in an `SQLException`.

start

Format:

```
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`.

Related tasks:

- "System monitoring for the IBM DB2 Driver for JDBC and SQLJ" on page 189

DB2XADataSource class

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` 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).

DB2XADataSource methods:

getDB2TrustedXAConnection

Formats:

```
public Object[] getDB2TrustedXAConnection(String user,
  String password,
  java.util.Properties properties)
  throws java.sql.SQLException
public Object[] getDB2TrustedXAConnection(
  java.util.Properties properties)
  throws java.sql.SQLException
public Object[] getDB2TrustedXAConnection(
  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. 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:

```
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. 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:

```
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.

Related concepts:
- "Example of a distributed transaction that uses JTA methods" on page 210
- "Java transaction management" on page 209

Related tasks:
- "Creating and deploying DataSource objects" on page 33

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 DB2 Driver for JDBC and SQLJ.

close

Format:

```
public void close()
  throws SQLException
```

Releases the resources that are associated with a com.ibm.jcc.DB2Xml object.

getDB2AsciiStream

Format:

```
public java.io.InputStream getDB2AsciiStream()
  throws SQLExceptionn
```

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 UTF-8-encoded binary stream.

getDB2Bytes

Format:

```
public byte[] getDB2Bytes()
  throws SQLExceptionn
```

Retrieves data from a DB2Xml object as a UTF-8-encoded byte array.

getDB2CharacterStream

Format:

```
public java.io.Reader getDB2CharacterStream()
  throws SQLExceptionn
```

Retrieves data from a DB2Xml object as a `java.io.Reader` object.

getDB2String

Format:

```
public String getDB2String()
  throws SQLExceptionn
```

Retrieves data from a DB2Xml object as a `String` value.

getDB2XmlAsciiStream

Format:

```
public InputStream getDB2XmlAsciiStream()
  throws SQLExceptionn
```

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:

```
public java.io.InputStream getDB2XmlBinaryStream(String targetEncoding)
  throws SQLExceptionn
```

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:

```
public byte[] getDB2XmlBytes(String targetEncoding)
  throws SQLExceptionn
```

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:

```
public java.io.Reader getDB2XmlCharacterStream()
  throws SQLExceptionn
```

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:

```
public String getDB2XmlString()
  throws SQLExceptionn
```

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:

```
public boolean isDB2XmlClosed()
  throws SQLException
```

Indicates whether a com.ibm.jcc.DB2Xml object has been closed.

Related concepts:
- "XML column updates in JDBC applications" on page 68
- "XML column updates in SQLJ applications" on page 134
- "XML data retrieval in JDBC applications" on page 70
- "XML data retrieval in SQLJ applications" on page 136

Related reference:
- "Mappings of CCSIDs to encoding names for serialized XML output data" in *XML Guide*
- "Data types that map to SQL data types in JDBC applications" on page 227

JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers

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 DB2 Driver for JDBC and SQLJ.

Supported methods:

For a comparison of method support by the JDBC drivers, see Driver support for JDBC APIs.

Support for scrollable and updatable ResultSets:

The IBM DB2 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 DB2 Driver for JDBC and SQLJ
-

Difference in error codes and SQLSTATEs returned for driver errors:

The IBM DB2 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 DB2 Driver for JDBC and SQLJ and SQLSTATEs issued by the IBM DB2 Driver for JDBC and SQLJ.

The JDBC/SQLJ driver for z/OS return ODBC SQLSTATEs when internal errors occur.

How much error message text is returned:

With the IBM DB2 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 DB2 Driver for JDBC and SQLJ security mechanisms, seeSecurity under the IBM DB2 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 DB2 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 DB2 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.

Result of an executeUpdate call that affects no rows:

The IBM DB2 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 DB2 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.

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 DB2 Driver for JDBC and SQLJ does not throw an exception until the subsequent `PreparedStatement.executeUpdate` method executes. Therefore, for the IBM DB2 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 DB2 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.

How character conversion is done:

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 DB2 Driver for JDBC and SQLJ, character data that is sent from the database server to the client is converted using Java's built-in character converters. The conversions that the IBM DB2 Driver for JDBC and SQLJ supports are limited to those that are supported by the underlying JRE implementation.

A IBM DB2 Driver for JDBC and SQLJ client using type 4 connectivity sends data to the database server 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.

Those drivers use CCSID information from the database server if it is available. The drivers convert input parameter data to the CCSID of the database server 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 DB2 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 DB2 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 DB2 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 DB2 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 DB2 Driver for JDBC and SQLJ supports the standard data type mappings of Java objects to JDBC data types when it converts *x* to a JDBC data type:

```
setObject(parameterIndex, x)
```

The DB2 JDBC Type 2 Driver supports the non-standard mapping of *x* from `java.lang.Character` to CHAR.

Support for ResultSet.getBinaryStream against a character column:

The IBM DB2 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.getBinaryStream against a binary column:

With the IBM DB2 Driver for JDBC and SQLJ, when you execute `ResultSet.getBinaryStream` 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.getBinaryStream` 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 DB2 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 DB2 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 DB2 Driver for JDBC and SQLJ, when you execute `ResultSet.next()` on a closed cursor, an `SQLException` is thrown. This conforms with the JDBC standard.

With the DB2 JDBC Type 2 Driver, when you execute `ResultSet.next()` on a closed cursor, a value of `false` is returned, and now exception is thrown.

Result of specifying null arguments in DatabaseMetaData calls:

With the IBM DB2 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.

Support for DATALINKs:

The IBM DB2 Driver for JDBC and SQLJ does not support the DATALINK SQL type.

The DB2 JDBC Type 2 Driver supports the DATALINK type in method calls of these forms:

- `PreparedStatement.setObject(`*`parameterIndex`*`, `*`x`*`, DB2Constants.DATALINK)`
- `PreparedStatement.setObject(`*`parameterIndex`*`, `*`x`*`, java.sql.Types.DATALINK)` (Java 1.4 or later)
- `PreparedStatement.setURL( `*`parameterIndex`*`, java.net.URL)`
- `PreparedStatement.setObject( `*`parameterIndex`*`, java.net.URL)`
- `PreparedStatement.setObject(`*`parameterIndex`*`, java.net.URL, java.sql.Types.DATALINK)` (Java 1.4 or later)
- `ResultSet.getString` for a DATALINK column
- `ResultSet.getURL` for a DATALINK column

Folding of method arguments to uppercase:

The IBM DB2 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 DB2 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 DB2 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 DB2 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 database server.

The DB2 JDBC Type 2 Driver removes white-space characters from SQLstatement text before it passes that text to the database server.

Result of executing PreparedStatement.executeBatch:

When a PreparedStatement.executeBatch statement is executed under the IBM DB2 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.executeBatch 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 DB2 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:

```
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 DB2 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.executeBatch` call if a parameter is not set for any of the statements in the batch.

Ability to call uncatalogued stored procedures:

The IBM DB2 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 DB2 Driver for JDBC and SQLJ driver, if the database server 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 database servers 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.

Connection.Commit() and Connection.Rollback() when autocommit mode is enabled:

In the current release of DB2, the IBM DB2 Driver for JDBC and SQLJ does not let you execute `Connection.Commit()` and `Connection.Rollback()` when autocommit mode is enabled. This behavior is compliant with the JDBC specification. Previous releases of the IBM DB2 Driver for JDBC and SQLJ, and earlier JDBC drivers allowed `Connection.Commit()` and `Connection.Rollback()` when autocommit mode was enabled.

Related concepts:
- "Security under the DB2 JDBC Type 2 Driver" on page 141
- "Security under the IBM DB2 Driver for JDBC and SQLJ" on page 142
- "LOBs in JDBC applications with the IBM DB2 Driver for JDBC and SQLJ" on page 57

Related tasks:
- "Connecting to a data source using the DataSource interface" on page 30
- "Connecting to a data source using the DriverManager interface with the IBM DB2 Driver for JDBC and SQLJ" on page 27
- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77
- "Making batch updates in JDBC applications" on page 42
- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 49
- "Calling stored procedures using CallableStatement methods" on page 53
- "Updating data in DB2 tables using the PreparedStatement.executeUpdate method" on page 40
- "Creating and modifying DB2 objects using the Statement.executeUpdate method" on page 39

Related reference:
- "Driver support for JDBC APIs" on page 247
- "Error codes issued by the IBM DB2 Driver for JDBC and SQLJ" on page 344
- "Data types that map to SQL data types in JDBC applications" on page 227
- "Properties for the IBM DB2 Driver for JDBC and SQLJ" on page 232
- "SQLSTATEs issued by the IBM DB2 Driver for JDBC and SQLJ" on page 345

SQLJ differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers

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 DB2 Driver for JDBC and SQLJ.

SQLJ support in the IBM DB2 Driver for JDBC and SQLJ differs from SQLJ support in the other DB2 JDBC drivers in the following areas:

db2sqljcustomize errors and the -collection parameter:

The db2sqljcustomize utility that is part of the IBM DB2 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 target of a bind operation with the db2sqljcustomize utility is a DB2 for z/OS server, and the -collection parameter contains any lowercase characters. db2sqljcustomize 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 DB2 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 DB2 Driver for JDBC and SQLJ. ***Before you can run those SQLJ applications under the IBM DB2 Driver for JDBC and SQLJ, you must retranslate and recustomize the applications using the IBM DB2 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 DB2 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 [ctxt] hv = {VALUES (MY_ROUTINE(1))};
```

For the IBM DB2 Driver for JDBC and SQLJ, you need to change that statement to something like this:

```
#sql [ctxt] {SELECT MY_ROUTINE(1) INTO :hv FROM SYSIBM.SYSDUMMY1};
```

Compound SQL statement support:

The DB2 JDBC Type 2 Driver supports compound SQL statements in an SQLJ statement clause, but the IBM DB2 Driver for JDBC and SQLJ does not. Therefore, you need to modify your SQLJ applications that include SQLJ statements with BEGIN COMPOUND and END COMPOUND. If you use compound statements to do batch updates, you can use the SQLJ batch update programming interfaces instead.

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 DB2 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 DB2 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.

Related tasks:
- "Connecting to a data source using SQLJ" on page 92

Error codes issued by the IBM DB2 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 DB2 Driver for JDBC and SQLJ. Currently, the IBM DB2 Driver for JDBC and SQLJ issues the following error codes:

Table 68. Error codes issued by the IBM DB2 Driver for JDBC and SQLJ

Error Code	Message text and explanation
+4204	Errors were encountered and tolerated as specified by the RETURN DATA UNTIL clause. **Explanation:** 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`.
-4200	Invalid operation: An invalid COMMIT or ROLLBACK has been called in an XA environment during a Global Transaction. **Explanation:** 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.
-4201	Invalid operation: setAutoCommit(true) is not allowed during Global Transaction. **Explanation:** 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.

Table 68. Error codes issued by the IBM DB2 Driver for JDBC and SQLJ (continued)

Error Code	Message text and explanation
-4203	Error executing *function*. Server returned *rc*. **Explanation:** An error occurred on an XA connection during execution of an SQL statement. For network optimization, the IBM DB2 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.
-4450	Feature not supported: *feature-name* is not supported.
-4496	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 DB2 server for an XA connection.
-4497	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.
-4498	A connection failed but has been re-established. The host name or IP address is \host-name\\ and the service name or port number is *port*. Special registers may or may not be re-attempted (Reason code = *rc*). **Explanation:** *host-name* and *port* indicate the database server at which the connection is reestablished. *rc* can have the following values: **1** The database server attempted to reset special registers to their original values. **2** The database server did not attempt to reset special registers to their original values. The application is rolled back to the previous commit point.
-4499	A fatal error occurred that resulted in a disconnect. **Explanation:** One possible cause is that a network error caused a socket to disconnect.
-99999	The IBM DB2 Driver for JDBC and SQLJ issued an error that does not yet have an error code.

Related tasks:
- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77
- "Handling SQL errors in an SQLJ application" on page 138

Related reference:
- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

SQLSTATEs issued by the IBM DB2 Driver for JDBC and SQLJ

SQLSTATEs in the range 46600 to 466ZZ are reserved for the IBM DB2 Driver for JDBC and SQLJ. Currently, the IBM DB2 Driver for JDBC and SQLJ returns a null SQLSTATE value for an internal error, unless the error is a DRDA error. The following SQLSTATEs are issued for DRDA errors:

02506 Tolerable error. This SQLSTATE is issued for SQLCODE +4204.

08003	A connection does not exist.
08004	The application server rejected establishment of the connection.
08506	Client reroute exception. This SQLSTATE is issued for SQLCODE -4498.
22021	A character is not in the coded character set.
24501	The identified cursor is not open.
2D521	SQL COMMIT or ROLLBACK are invalid in the current operating environment.
58008	Execution failed due to a distribution protocol error that will not affect the successful execution of subsequent DDM commands or SQL statements.
58009	Execution failed due to a distribution protocol error that caused deallocation of the conversation.
58010	Execution failed due to a distribution protocol error that will affect the successful execution of subsequent DDM commands or SQL statements.
58014	The DDM command is not supported.
58015	The DDM object is not supported.
58016	The DDM parameter is not supported.
58017	The DDM parameter value is not supported.

Related tasks:

- "Handling an SQLException under the IBM DB2 Driver for JDBC and SQLJ" on page 77
- "Handling SQL errors in an SQLJ application" on page 138

Related reference:

- "JDBC differences between the IBM DB2 Driver for JDBC and SQLJ and other DB2 JDBC drivers" on page 335

How to find IBM DB2 Driver for JDBC and SQLJ version and environment information

To determine the version of the IBM DB2 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:

```
►►──java──com.ibm.db2.jcc.DB2Jcc──┬──────────┬──┬────────────────┬──┬───────┬──────────────────────►◄
                                  └─-version─┘  └─-configuration─┘  └─-help─┘
```

DB2Jcc option descriptions:

-version
Specifies that the IBM DB2 Driver for JDBC and SQLJ displays its name and version.

-configuration
Specifies that the IBM DB2 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.

```
(myid@mymachine) /home/myid $ java com.ibm.db2.jcc.DB2Jcc -version
IBM DB2 Driver for JDBC and SQLJ Architecture 2.1.29 Test Build

(myid@mymachine) /home/myid $ java com.ibm.db2.jcc.DB2Jcc -configuration
[ibm][db2][jcc] BEGIN TRACE_DRIVER_CONFIGURATION
[ibm][db2][jcc] Driver: IBM DB2 Driver for JDBC and SQLJ Architecture 2.1.29 Test Build
[ibm][db2][jcc] Compatible JRE versions: { 1.4 }
[ibm][db2][jcc][ibm][db2][jcc] Target server licensing restrictions: { z/OS: disabled; SQLDS: disabl
ed; iSeries: disabled; DB2 for Unix/Windows: disabled; Cloudscape:disabled }
[ibm][db2][jcc] Range checking enabled: true
[ibm][db2][jcc] Bug check level: 0xff
[ibm][db2][jcc] Default fetch size: 64
[ibm][db2][jcc] Default isolation: 2
[ibm][db2][jcc] Collect performance statistics: false
[ibm][db2][jcc] No security manager detected.
[ibm][db2][jcc] Detected local client host: mymachine/9.99.99.999
[ibm][db2][jcc] Access to package sun.io is permitted by security manager.
[ibm][db2][jcc] JDBC 1 system property jdbc.drivers = null
[ibm][db2][jcc] Java Runtime Environment version 1.4.2
[ibm][db2][jcc] Java Runtime Environment vendor = IBM Corporation
[ibm][db2][jcc] Java vendor URL = http://www.ibm.com/
[ibm][db2][jcc] Java installation directory = /wsdb/v91/bldsupp/AIX/jdk1.4.2/jre
[ibm][db2][jcc] Java Virtual Machine specification version = 1.0
[ibm][db2][jcc] Java Virtual Machine specification vendor = Sun Microsystems Inc.
[ibm][db2][jcc] Java Virtual Machine specification name = Java Virtual Machine Specification
[ibm][db2][jcc] Java Virtual Machine implementation version = 1.4.2
[ibm][db2][jcc] Java Virtual Machine implementation vendor = IBM Corporation
[ibm][db2][jcc] Java Virtual Machine implementation name = Classic VM
[ibm][db2][jcc] Java Runtime Environment specification version = 1.4.2
[ibm][db2][jcc] Java Runtime Environment specification vendor = Sun Microsystems Inc.
[ibm][db2][jcc] Java Runtime Environment specification name = Java Platform API Specification
[ibm][db2][jcc] Java class format version number = 46.0
[ibm][db2][jcc] Java class path = .:/home/myid/sqllib/java/db2jcc.jar:/home/myid/sqllib/java/db2
java.zip:/home/myid/sqllib/java/sqlj.zip:/home/myid/sqllib/java/runtime.zip:/wsdb/v91/bldsupp/AI
X/jdk1.4.2/jdbc2.0_stdext/jdbc2_0-stdext.jar:/wsdb/v91/bldsupp/AIX/jdk1.4.2/jta1.0.1/jta-spec1_0_1.j
ar:/wsdb/v91/bldsupp/AIX/jdk1.4.2/jndi1.2/lib/jndi.jar:/home/myid/util:./test:/home/myid/build/c
ur/engn/lib/db2jcc_license_cisuz.jar:/home/myid/build/cur/engn/lib/db2jcc_license_cu.jar
[ibm][db2][jcc] Java native library path = /wsdb/v91/bldsupp/AIX/jdk1.4.2/jre/bin:/wsdb/v91/bldsupp/
AIX/jdk1.4.2/jre/bin/classic:/home/myid/sqllib/lib:/local/cobol:/usr/lib
[ibm][db2][jcc] Path of extension directory or directories = /wsdb/v91/bldsupp/AIX/jdk1.4.2/jre/lib/
ext
[ibm][db2][jcc] Operating system name = AIX
[ibm][db2][jcc] Operating system architecture = ppc
[ibm][db2][jcc] Operating system version = 4.3
[ibm][db2][jcc] File separator ("/" on UNIX) = /
[ibm][db2][jcc] Path separator (":" on UNIX) = :
[ibm][db2][jcc] User's account name = myid
[ibm][db2][jcc] User's home directory = /home/myid
[ibm][db2][jcc] User's current working directory = /home/myid
[ibm][db2][jcc] END TRACE_DRIVER_CONFIGURATION
(myid@mymachine) /home/myid $
```

Figure 65. Sample DB2Jcc output

Commands for SQLJ program preparation

The topics that follow contain commands for SQLJ program preparation.
- "sqlj - SQLJ translator" on page 348
- "db2sqljcustomize - SQLJ profile customizer" on page 351
- "db2sqljbind - SQLJ profile binder" on page 361
- "db2sqljprint - SQLJ profile printer" on page 367

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──┐  ┌─-linemap=NO──┐  ┌─-smap=NO──┐
►──┼────────────────┼──┼──────────────┼──┼───────────┼──┬─────────────────────┬──┬──────────────┬──►
   └─-compile=false─┘  └─-linemap=YES─┘  └─-smap=YES─┘  └─-encoding=encoding──┘  └─-db2optimize─┘

►──┬────────────┬──┬─────────┬──┬──────────┬──┬─────────┬──┬──────────────────────────┬──────────►
   └─-ser2class─┘  └─-status─┘  └─-version─┘  └─-C-help─┘  │ ┌──────────────────────┐ │
                                                           │ ▼                 (1)  │ │
                                                           └───-Ccompiler-option────┴─┘

►──┬──────────────────┬──┬──────────────────────────┬─────────────────────────────────────────►◄
   │ ┌──────────────┐ │  │ ┌──────────────────────┐ │
   │ ▼              │ │  │ ▼                      │ │
   └───-JJVM-option─┴─┘  └───SQLJ-source-file-name─┴─┘
```

Notes:

1 The -C-classpath and -C-sourcepath options are used by the SQLJ translator as well as by the Java compiler.

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. The default directory 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. These files include:
- The serialized profile files (.ser files)
- If the sqlj command invokes the Java compiler, the class files that are generated by the compiler (.class files)

The default directory 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.

-props=*properties-file*
Specifies the name of a file from which the SQLJ translator is to obtain a list of options.

-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, the IBM DB2 Driver for JDBC and SQLJ file db2jcc.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 DB2 Driver for JDBC and SQLJ. The information is in this form:

```
IBM SQLJ xxxx.xxxx.xx
```

-C-help
Specifies that the SQLJ translator displays help information for the Java compiler.

-C*compiler-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.

-sourcepath
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.

-J*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-Xmine2M
```

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*_SJProfile*IDNumber*.ser`
- If the SQLJ translator invokes the Java compiler, the class files that the compiler generates.

Examples:

```
sqlj -encoding=UTF8 -C-O MyApp.sqlj
```

Related reference:
- "db2sqljbind - SQLJ profile binder" on page 361
- "db2sqljcustomize - SQLJ profile customizer" on page 351
- "db2sqljprint - SQLJ profile printer" on page 367

db2sqljcustomize - SQLJ profile customizer

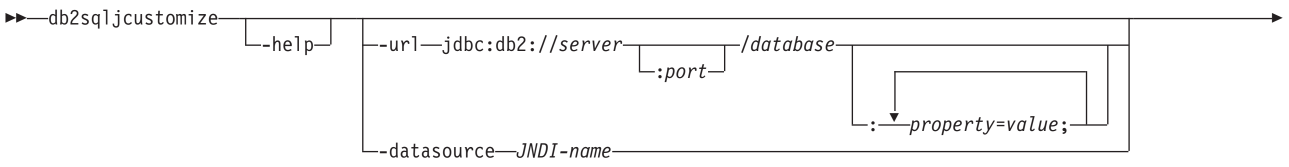
db2sqljcustomize processes an SQLJ profile, which contains embedded SQL statements. By default, db2sqljcustomize produces four DB2 packages: one for each isolation level. db2sqljcustomize augments the profile with DB2-specific information for use at run time.

Authorization:

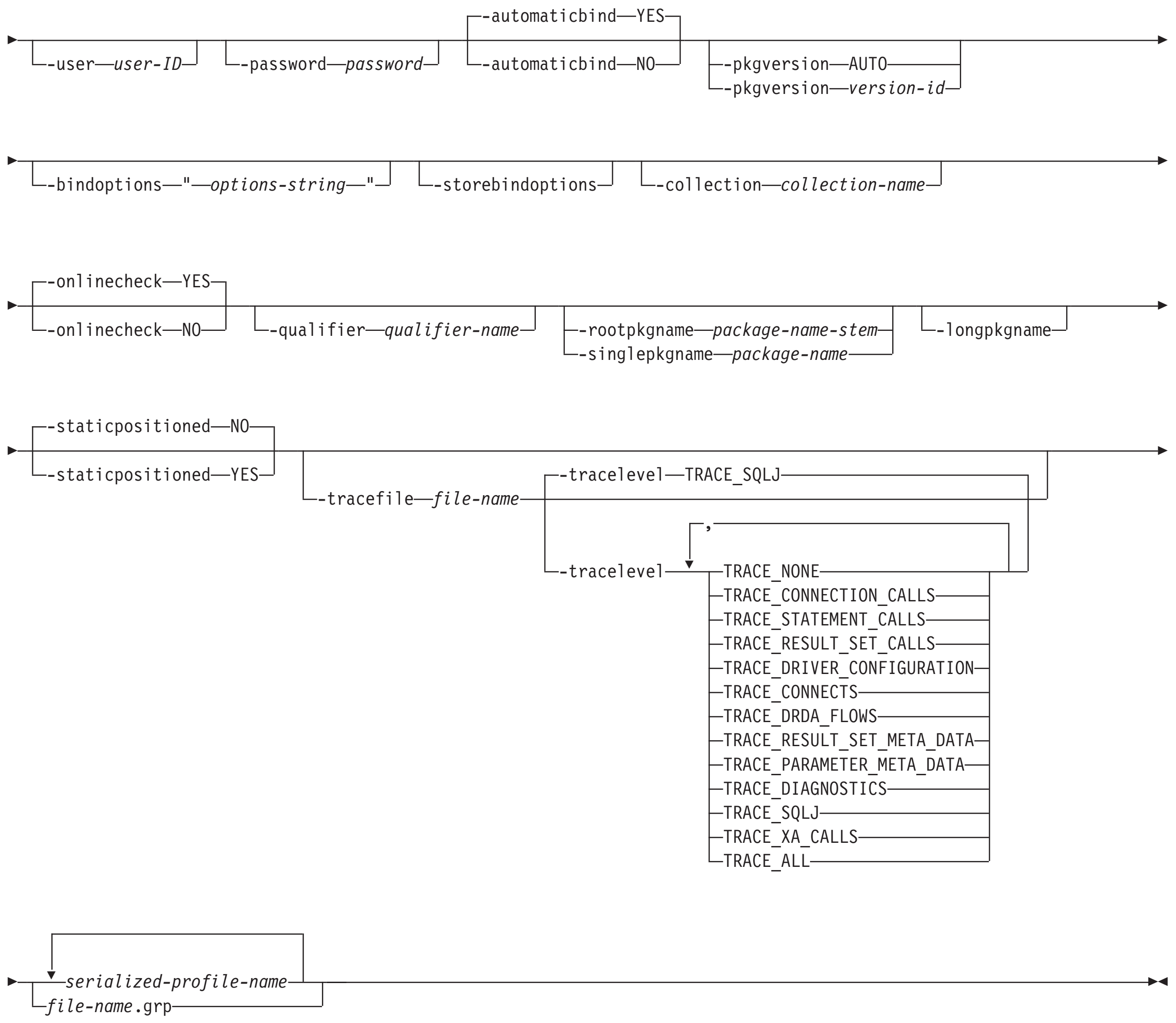
The privilege set of the process must include one of the following authorities:
- SYSADM authority
- DBADM authority
- If the package does not exist, the BINDADD privilege, and one of the following privileges:
  - CREATEIN privilege
  - IMPLICIT_SCHEMA authority on the database if the schema name of 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. If the user has SYSADM authority, but no explicit privileges to complete the bind, the DB2 database manager grants explicit DBADM authority automatically.

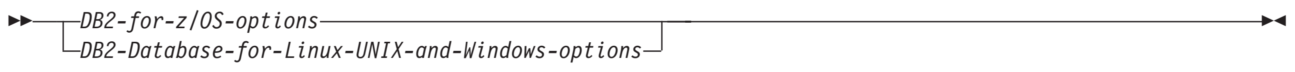
Command syntax:



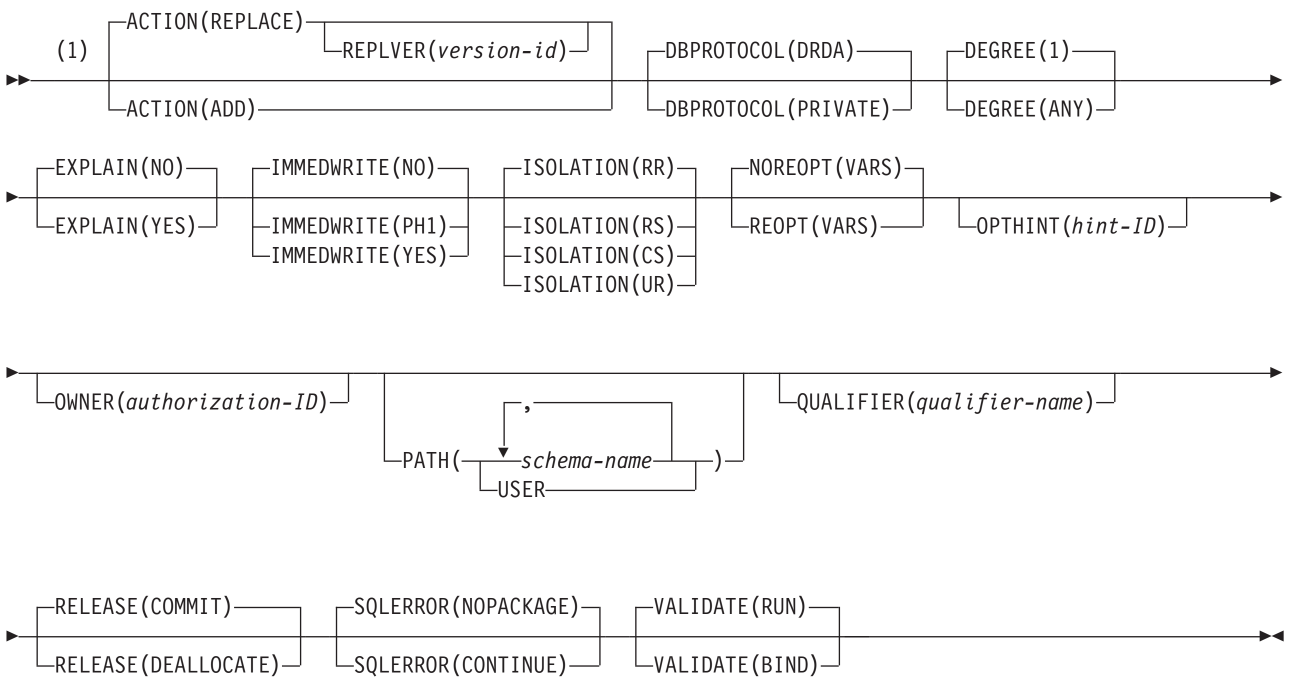
db2sqljcustomize - SQLJ profile customizer



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

```
     (1) ┌BLOCKING UNAMBIG─┐             ┌DEGREE 1──┐ ┌EXPLAIN NO──┐ ┌EXPLSNAP NO──┐
►►───────┼─────────────────┼──┬────────┬─┼──────────┼─┼────────────┼─┼─────────────┼──────────►
         ├BLOCKING ALL─────┤  ├DEC 15──┤ └DEGREE ANY┘ └EXPLAIN YES─┘ ├EXPLSNAP ALL─┤
         └BLOCKING NO──────┘  └DEC 31──┘                             └EXPLSNAP YES─┘

   ┌FEDERATED NO──┐                          ┌INSERT DEF─┐ ┌ISOLATION CS─┐
►──┼──────────────┼─┬──────────────────────┬─┼───────────┼─┼─────────────┼────────────────────►
   └FEDERATED YES─┘ └FUNCPATH schema-name──┘ └INSERT BUF─┘ ├ISOLATION RR─┤
                                                           ├ISOLATION RS─┤
                                                           └ISOLATION UR─┘

►──┬──────────────────────────┬─┬─────────────────────────────┬─┬──────────────────────────────┬──►
   └OWNER authorization-ID────┘ └QUALIFIER qualifier-name─────┘ └QUERYOPT optimization-level───┘

   ┌SQLERROR NOPACKAGE─┐ ┌SQLWARN YES─┐ ┌STATICREADONLY NO──┐ ┌VALIDATE RUN──┐
►──┼───────────────────┼─┼────────────┼─┼───────────────────┼─┼──────────────┼──────────────────►◄
   └SQLERROR CONTINUE──┘ └SQLWARN NO──┘ └STATICREADONLY YES─┘ └VALIDATE BIND─┘
```

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 MVS 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.

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:

```
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. For the definitions of these properties, see Properties for the IBM DB2 Driver for JDBC and SQLJ.

-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 DB2 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 DB2 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 db2sqljbind 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 serialied 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 `db2sqljbind` 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-stem***1**	For isolation level UR
*package-name-stem***2**	For isolation level CS
*package-name-stem***3**	For isolation level RS
*package-name-stem***4**	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.

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 69 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 69. Parts of a package name that is generated by db2sqljcustomize

Package name part	Number of bytes	Value
Bytes-from-program-name	*m*=min(Length(*program-name*), *maxlen*–1–Length(*IDNumber*))	First *m* bytes of *program-name*, in uppercase
IDNumber	Length(*IDNumber*)	*IDNumber*
PkgIsolation	1	1, 2, 3, or 4. This value represents the transaction isolation level for the package. See Table 70.

Table 70 shows the values of the *PkgIsolation* portion of a package name that is generated by db2sqljcustomize.

Table 70. PkgIsolation values and associated isolation levels

PkgNumber value	Isolation level for package
1	Uncommitted read (UR)
2	Cursor stability (CS)
3	Read stability (RS)
4	Repeatable read (RR)

Example: Suppose that a profile name is ThisIsMyProg_SJProfile111.ser. The db2sqljcustomize option -longpkgname is not specified. Therefore, *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, *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. *Bytes-from-program-name* is A. *IDNumber* is 0. Therefore, the four package names are:

```
A01
A02
A03
A04
```

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.

-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. 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.

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 -staticpostioned 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 perfom 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.

Related reference:

- "BIND command" in *Command Reference*
- "db2sqljprint - SQLJ profile printer" on page 367
- "db2sqljbind - SQLJ profile binder" on page 361

db2sqljbind - SQLJ profile binder

db2sqljbind binds DB2 packages for a serialized profile that was previously customized with the db2sqljcustomize command.

Authorization:

The privilege set of the process must include one of the following authorities:
- SYSADM authority
- 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. If the user has SYSADM authority, but no explicit privileges to complete the bind, the DB2 database manager grants explicit DBADM authority automatically.

Command syntax:

```
►►──db2sqljbind──┬───────┬──-url──jdbc:db2://server──┬───────┬──/database──┬──────────────────────────┬──────────►
                 └─-help─┘                            └─:port─┘             │   ┌──────────────────┐   │
                                                                            │   ▼                  │   │
                                                                            └─:───property=value;──┴───┘

►──-user──user-ID──-password──password──┬──────────────────────────────────────┬───────────────────────────────►
                                         └─-bindoptions──"──options-string──"─┘

   ┌─-staticpositioned──NO──┐
►──┴─-staticpositioned──YES─┴────────────────────────────────────────────────────────────────────────────────────►
```

```
►─┬──────────────────────────────────────────────────────────────────────────┬──────────►
  └─-tracefile──file-name─┬─-tracelevel──TRACE_SQLJ──────────────────────────┬┘
                          │               ┌─,──────────────────────────┐     │
                          │               ▼                            │     │
                          └─-tracelevel──────┬─TRACE_NONE──────────────┬─┴─────┘
                                             ├─TRACE_CONNECTION_CALLS──┤
                                             ├─TRACE_STATEMENT_CALLS───┤
                                             ├─TRACE_RESULT_SET_CALLS──┤
                                             ├─TRACE_DRIVER_CONFIGURATION─┤
                                             ├─TRACE_CONNECTS──────────┤
                                             ├─TRACE_DRDA_FLOWS────────┤
                                             ├─TRACE_RESULT_SET_META_DATA─┤
                                             ├─TRACE_PARAMETER_META_DATA──┤
                                             ├─TRACE_DIAGNOSTICS───────┤
                                             ├─TRACE_SQLJ──────────────┤
                                             ├─TRACE_XA_CALLS──────────┤
                                             └─TRACE_ALL───────────────┘


  ┌──────────────────────────┐
  ▼                          │
►───serialized-profile-name──┴──────────────────────────────────────────────────────►◄
```

options-string:

```
►►──┬─DB2-for-z/OS-options─────────────────────────────┬─────────────────────────►◄
    └─DB2-Database-for-Linux-UNIX-and-Windows-options──┘
```

DB2 for z/OS options:

```
                .-ACTION(REPLACE)----------------------.   .-DBPROTOCOL(DRDA)----.   .-DEGREE(1)---.
       (1)      |                 '-REPLVER(version-id)-'  |                     |   |             |
>>-------------+--------------------------------------+---+---------------------+---+-------------+------->
               '-ACTION(ADD)--------------------------'   '-DBPROTOCOL(PRIVATE)-'   '-DEGREE(ANY)-'

   .-EXPLAIN(NO)--.   .-IMMEDWRITE(NO)--.   .-ISOLATION(RR)-.   .-NOREOPT(VARS)-.
>--+--------------+---+-----------------+---+---------------+---+---------------+--+--------------------+-->
   '-EXPLAIN(YES)-'   +-IMMEDWRITE(PH1)-+   +-ISOLATION(RS)-+   '-REOPT(VARS)---'  '-OPTHINT(hint-ID)---'
                      '-IMMEDWRITE(YES)-'   +-ISOLATION(CS)-+
                                            '-ISOLATION(UR)-'

>--+---------------------------+--+---------------------------------+--+---------------------------------+-->
   '-OWNER(authorization-ID)---'  |           .-,-----------.       |  '-QUALIFIER(qualifier-name)-------'
                                  |           V             |       |
                                  '-PATH(--+----schema-name-+---+-)-'
                                           '-USER---------------'

   .-RELEASE(COMMIT)-----.   .-SQLERROR(NOPACKAGE)-.   .-VALIDATE(RUN)--.
>--+---------------------+---+---------------------+---+----------------+-------------------------------><
   '-RELEASE(DEALLOCATE)-'   '-SQLERROR(CONTINUE)--'   '-VALIDATE(BIND)-'
```

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─┐
►►──────┼────────────────┼───┬──────┬──┼──────────┼──┼───────────┼──┼────────────┼──────────►
        ├BLOCKING ALL────┤   ├DEC 15┤  └DEGREE ANY┘  └EXPLAIN YES┘  ├EXPLSNAP ALL┤
        └BLOCKING NO─────┘   └DEC 31┘                               └EXPLSNAP YES┘

  ┌FEDERATED NO─┐                          ┌INSERT DEF┐  ┌ISOLATION CS┐
►─┼─────────────┼──┬────────────────────┬──┼──────────┼──┼────────────┼──────────────────────►
  └FEDERATED YES┘  └FUNCPATH schema-name┘  └INSERT BUF┘  ├ISOLATION RR┤
                                                         ├ISOLATION RS┤
                                                         └ISOLATION UR┘

►─┬───────────────────────┬──┬─────────────────────────┬──┬──────────────────────────────┬──►
  └OWNER authorization-ID┘  └QUALIFIER qualifier-name┘  └QUERYOPT optimization-level┘

  ┌SQLERROR NOPACKAGE┐  ┌SQLWARN YES┐  ┌STATICREADONLY NO─┐  ┌VALIDATE RUN─┐
►─┼──────────────────┼──┼───────────┼──┼──────────────────┼──┼─────────────┼────────────────►◄
  └SQLERROR CONTINUE─┘  └SQLWARN NO─┘  └STATICREADONLY YES┘  └VALIDATE BIND┘
```

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. This URL is used if the -automaticbind or -onlinecheck option is YES. The variable parts of the `-url` value are:

server
The domain name or IP address of the MVS 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.

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:

```
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. For the definitions of these properties, see Properties for the IBM DB2 Driver for JDBC and SQLJ.

-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 DB2 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.

-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
: Specifies the name 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.

  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.

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 a 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 UDPATE 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.

Related reference:

- "BIND command" in *Command Reference*
- "db2sqljcustomize - SQLJ profile customizer" on page 351
- "db2sqljprint - SQLJ profile printer" on page 367

- "sqlj - SQLJ translator" on page 348

db2sqljprint - SQLJ profile printer

db2sqljprint prints the contents of a DB2 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
```

Related reference:

- "db2sqljcustomize - SQLJ profile customizer" on page 351
- "db2sqljbind - SQLJ profile binder" on page 361

Appendix A. DB2 Database technical information

Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF CD)
  - printed books
- Command line help
  - Command help
  - Message help
- Sample programs

IBM periodically makes documentation updates available. If you access the online version on the DB2 Information Center at ibm.com®, you do not need to install documentation updates because this version is kept up-to-date by IBM. If you have installed the DB2 Information Center, it is recommended that you install the documentation updates. Documentation updates allow you to update the information that you installed from the *DB2 Information Center CD* or downloaded from Passport Advantage as new information becomes available.

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 Redbooks™ online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how we can improve the DB2 documentation, send an e-mail 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 e-mail 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.

Related concepts:

- "Features of the DB2 Information Center" in *Online DB2 Information Center*
- "Sample files" in *Samples Topics*

Related tasks:

- "Invoking command help from the command line processor" in *Command Reference*
- "Invoking message help from the command line processor" in *Command Reference*
- "Updating the DB2 Information Center installed on your computer or intranet server" on page 375

Related reference:

- "DB2 technical library in PDF format" on page 370

DB2 technical library in PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/shop/publications/order.

Although the tables identify books available in print, the books might not be available in your country or region.

The information in these books is fundamental to all DB2 users; you will find this information useful whether you are a programmer, a database administrator, or someone who works with DB2 Connect or other DB2 products.

Table 71. DB2 technical information

Name	Form Number	Available in print
Administration Guide: Implementation	SC10-4221	Yes
Administration Guide: Planning	SC10-4223	Yes
Administrative API Reference	SC10-4231	Yes
Administrative SQL Routines and Views	SC10-4293	No
Call Level Interface Guide and Reference, Volume 1	SC10-4224	Yes
Call Level Interface Guide and Reference, Volume 2	SC10-4225	Yes
Command Reference	SC10-4226	No
Data Movement Utilities Guide and Reference	SC10-4227	Yes
Data Recovery and High Availability Guide and Reference	SC10-4228	Yes
Developing ADO.NET and OLE DB Applications	SC10-4230	Yes
Developing Embedded SQL Applications	SC10-4232	Yes
Developing SQL and External Routines	SC10-4373	No

Table 71. DB2 technical information (continued)

Name	Form Number	Available in print
Developing Java Applications	SC10-4233	Yes
Developing Perl and PHP Applications	SC10-4234	No
Getting Started with Database Application Development	SC10-4252	Yes
Getting started with DB2 installation and administration on Linux and Windows	GC10-4247	Yes
Message Reference Volume 1	SC10-4238	No
Message Reference Volume 2	SC10-4239	No
Migration Guide	GC10-4237	Yes
Net Search Extender Administration and User's Guide **Note:** HTML for this document is not installed from the HTML documentation CD.	SH12-6842	Yes
Performance Guide	SC10-4222	Yes
Query Patroller Administration and User's Guide	GC10-4241	Yes
Quick Beginnings for DB2 Clients	GC10-4242	No
Quick Beginnings for DB2 Servers	GC10-4246	Yes
Spatial Extender and Geodetic Data Management Feature User's Guide and Reference	SC18-9749	Yes
SQL Guide	SC10-4248	Yes
SQL Reference, Volume 1	SC10-4249	Yes
SQL Reference, Volume 2	SC10-4250	Yes
System Monitor Guide and Reference	SC10-4251	Yes
Troubleshooting Guide	GC10-4240	No
Visual Explain Tutorial	SC10-4319	No
What's New	SC10-4253	Yes
XML Extender Administration and Programming	SC18-9750	Yes
XML Guide	SC10-4254	Yes
XQuery Reference	SC18-9796	Yes

Table 72. DB2 Connect-specific technical information

Name	Form Number	Available in print
DB2 Connect User's Guide	SC10-4229	Yes
Quick Beginnings for DB2 Connect Personal Edition	GC10-4244	Yes

Table 72. DB2 Connect-specific technical information (continued)

Name	Form Number	Available in print
Quick Beginnings for DB2 Connect Servers	GC10-4243	Yes

Table 73. WebSphere Information Integration technical information

Name	Form Number	Available in print
WebSphere Information Integration: Administration Guide for Federated Systems	SC19-1001	Yes
WebSphere Information Integration: ASNCLP Program Reference for Replication and Event Publishing	SC19-1000	Yes
WebSphere Information Integration: Configuration Guide for Federated Data Sources	SC19-1034	No
WebSphere Information Integration: SQL Replication Guide and Reference	SC19-1002	Yes

Note: The DB2 Release Notes provide additional information specific to your product's release and fix pack level. For more information, see the related links.

Related concepts:

- "Overview of the DB2 technical information" on page 369
- "About the Release Notes" in *Release notes*

Related tasks:

- "Ordering printed DB2 books" on page 372

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* CD 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 CD 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 CD 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/db2help/.

Procedure:

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/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:
  - 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
    - The IBM Publications Web site at http://www.ibm.com/shop/publications/order. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the ″About this site″ link.
  - When you call, specify that you want to order a DB2 publication.
  - Provide your representative with the titles and form numbers of the books that you want to order.

Related concepts:

- "Overview of the DB2 technical information" on page 369

Related reference:

- "DB2 technical library in PDF format" on page 370

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.

Procedure:

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.

Related tasks:

- "Invoking command help from the command line processor" in *Command Reference*
- "Invoking message help from the command line processor" in *Command Reference*

Accessing different versions of the DB2 Information Center

For DB2 Version 9 topics, the DB2 Information Center URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/.

For DB2 Version 8 topics, go to the Version 8 Information Center URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/.

Related tasks:

- "Updating the DB2 Information Center installed on your computer or intranet server" on page 375

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.

Procedure:

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 the Firefox browser:

1. In Firefox, select the **Tools** —> **Options** —> **Languages** button. 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.

Related concepts:

- "Overview of the DB2 technical information" on page 369

Updating the DB2 Information Center installed on your computer or intranet server

If you have a locally-installed DB2 Information Center, updated topics can be available for download. The 'Last updated' value found at the bottom of most topics indicates the current level for that topic.

To determine if there is an update available for the entire DB2 Information Center, look for the 'Last updated' value on the Information Center home page. Compare the value in your locally installed home page to the latest value which is available on the IBM hosted Information Center home page. If they are the same, you have the latest documentation level and no update is required. If the are not the same, you should update your locally-installed Information Center.

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 download and apply updates.
2. Use the Update feature to determine if update packages are available from IBM. If update packages are available, use the Update feature to download the packages. (The Update feature is only available in stand-alone mode.)
3. Stop the stand-alone Information Center, and restart the DB2 Information Center service on your computer.

Procedure:

To update the DB2 Information Center installed on your computer or intranet server:

1. Stop the DB2 Information Center service.
   - 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:
     ```
     /etc/init.d/db2icdv9 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 `C:\Program Files\IBM\DB2 Information Center\Version 9` directory.
     c. Run the `help_start.bat` file using the fully qualified path for the DB2 Information Center:
        ```
        <DB2 Information Center dir>\doc\bin\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` directory.
     b. Run the `help_start.sh` file using the fully qualified path for the DB2 Information Center:
        ```
        <DB2 Information Center dir>/doc/bin/help_start
        ```

The systems default Web browser launches to display the stand-alone Information Center.

3. Click the Update button ( ). On the right hand panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the download process, check the selections you want to download, then click **Install Updates**.
5. After the download and installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center.
   - On Windows, run the `help_end.bat` file using the fully qualified path for the DB2 Information Center:
     ```
     <DB2 Information Center dir>\doc\bin\help_end.bat
     ```
   - On Linux, run the `help_end.sh` file using the fully qualified path for the DB2 Information Center:
     ```
     <DB2 Information Center dir>/doc/bin/help_end
     ```
7. Restart the DB2 Information Center service.
   - 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/db2icdv9 start
     ```

The updated DB2 Information Center displays the new and updated topics.

Related concepts:

- "DB2 Information Center installation options" in *Quick Beginnings for DB2 Servers*

Related tasks:

- "Installing the DB2 Information Center using the DB2 Setup wizard (Linux)" in *Quick Beginnings for DB2 Servers*
- "Installing the DB2 Information Center using the DB2 Setup wizard (Windows)" in *Quick Beginnings for DB2 Servers*

DB2 Visual Explain tutorial

The DB2 Visual Explain tutorial helps you learn about analyzing, optimizing, and tuning SQL statements for better performance. 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/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

DB2 Visual Explain tutorial:

To view the tutorial, click on the title.

Visual Explain Tutorial
: Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

Related concepts:

- "Visual Explain overview" in *Administration Guide: Implementation*

DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 products.

DB2 documentation
Troubleshooting information can be found in the DB2 Troubleshooting Guide or the Support and Troubleshooting section of the DB2 Information Center. There you will find information on how to isolate and identify 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 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.

Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/udb/support.html

Related concepts:

- "Introduction to problem determination" in *Troubleshooting Guide*
- "Overview of the DB2 technical information" on page 369

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.

Appendix B. Notices

IBM may not offer the products, services, or features discussed in this document in all 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
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, 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.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© (*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

Company, product, or service names identified in the documents of the DB2 Version 9 documentation library may be trademarks or service marks of International Business Machines Corporation or other companies. Information on the trademarks of IBM Corporation in the United States, other countries, or both is located at http://www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 documentation library:

Microsoft, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Itanium®, Pentium®, and Xeon® are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks 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.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Contacting IBM

To contact IBM in your country or region, check the IBM Directory of Worldwide Contacts at http://www.ibm.com/planetwide

To learn more about DB2 products, go to http://www.ibm.com/software/data/db2/.

Printed in USA

SC10-4233-00

DB2 9 BETA

Spine information:

IBM DB2

DB2 9 BETA

DB2 Version 9

Developing Java Applications