DB2 Universal Database for z/OS

Version 8

Application Programming Guide and Reference FOR JAVA™

SC18-7414-03

DB2 Universal Database for z/OS

Version 8

Application Programming Guide and Reference FOR JAVA™

SC18-7414-03

Note
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 333.

Fourth Edition, Softcopy Only (February 2006)

This edition applies to Version 8 of IBM DB2 Universal Database for z/OS (DB2 UDB for z/OS), product number 5625-DB2, and to any subsequent releases until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

This softcopy version is based on the printed edition of the book and includes the changes indicated in the printed version by vertical bars. Additional changes made to this softcopy version of the book since the hardcopy book was published are indicated by the hash (#) symbol in the left-hand margin. Editorial changes that have no technical significance are not noted.

This and other books in the DB2 for z/OS library are periodically updated with technical changes. These updates are made available to licensees of the product on CD-ROM and on the Web (currently at www.ibm.com/software/data/db2/zos/library.html). Check these resources to ensure that you are using the most current information.

Contents

About this book

This book describes DB2® UDB for z/OS® support for Java™. This support lets you access relational databases from Java application programs.

Who should read this book

This book is for the following users:

- DB2 UDB for z/OS application developers who are familiar with Structured Query Language (SQL) and who know the Java programming language.
- DB2 UDB for z/OS system programmers who are installing JDBC and SQLJ support.

Terminology and citations

In this information, DB2 Universal Database™ for z/OS is referred to as "DB2 UDB for z/OS." In cases where the context makes the meaning clear, DB2 UDB for z/OS is referred to as "DB2." When this information refers to titles of books in this library, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to *IBM® DB2 Universal Database for z/OS SQL Reference*.)

When referring to a DB2 product other than DB2 UDB for z/OS, this information uses the product's full name to avoid ambiguity.

The following terms are used as indicated:

DB2 Represents either the DB2 licensed program or a particular DB2 subsystem.

OMEGAMON
Refers to any of the following products:
- IBM Tivoli OMEGAMON XE for DB2 Performance Expert on z/OS
- IBM Tivoli OMEGAMON XE for DB2 Performance Monitor on z/OS
- IBM DB2 Performance Expert for Multiplatforms and Workgroups
- IBM DB2 Buffer Pool Analyzer for z/OS

C, C++, and C language
Represent the C or C++ programming language.

CICS® Represents CICS Transaction Server for z/OS or CICS Transaction Server for OS/390®.

IMS™ Represents the IMS Database Manager or IMS Transaction Manager.

MVS™
Represents the MVS element of the z/OS operating system, which is equivalent to the Base Control Program (BCP) component of the z/OS operating system.

RACF®
Represents the functions that are provided by the RACF component of the z/OS Security Server.

How to read the syntax diagrams

The following rules apply to the syntax diagrams that are used in this book:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►— symbol indicates the beginning of a statement.

  The —► symbol indicates that the statement syntax is continued on the next line.

  The ►— symbol indicates that a statement is continued from the previous line.

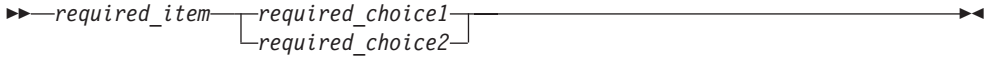
  The —►◄ symbol indicates the end of a statement.
- Required items appear on the horizontal line (the main path).

- Optional items appear below the main path.

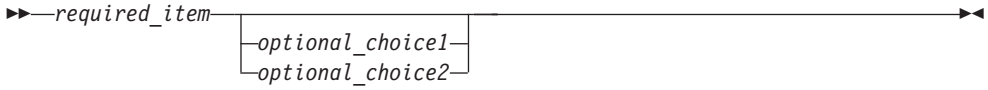
  If an optional item appears above the main path, that item has no effect on the execution of the statement and is used only for readability.

- If you can choose from two or more items, they appear vertically, in a stack.

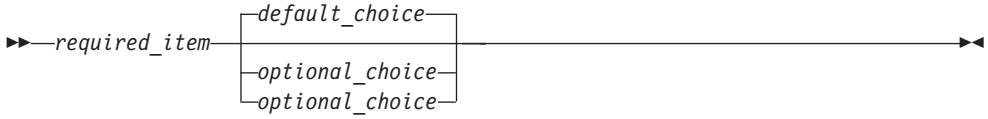
  If you *must* choose one of the items, one item of the stack appears on the main path.



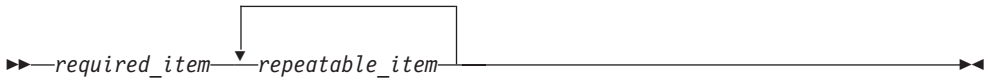
  If choosing one of the items is optional, the entire stack appears below the main path.



  If one of the items is the default, it appears above the main path and the remaining choices are shown below.



- An arrow returning to the left, above the main line, indicates an item that can be repeated.



  If the repeat arrow contains a comma, you must separate repeated items with a comma.

A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, FROM). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *column-name*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products. The major accessibility features in z/OS products, including DB2 UDB for z/OS, enable users to:

- Use assistive technologies such as screen reader and screen magnifier software
- Operate specific or equivalent features by using only a keyboard
- Customize display attributes such as color, contrast, and font size

Assistive technology products, such as screen readers, function with the DB2 UDB for z/OS user interfaces. Consult the documentation for the assistive technology products for specific information when you use assistive technology to access these interfaces.

Online documentation for Version 8 of DB2 UDB for z/OS is available in the Information management software for z/OS solutions information center, which is an accessible format when used with assistive technologies such as screen reader or screen magnifier software. The Information management software for z/OS solutions information center is available at the following Web site: http://publib.boulder.ibm.com/infocenter/dzichelp

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 UDB for z/OS documentation. You can use the following methods to provide comments:

- Send your comments by e-mail to db2pubs@vnet.ibm.com and include the name of the product, the version number of the product, and the number of the book. If you are commenting on specific text, please list the location of the text (for example, a chapter and section title, page number, or a help topic title).
- You can also send comments from the Web. Visit the library Web site at:

  www.ibm.com/software/db2zos/library.html

  This Web site has a feedback page that you can use to send comments.
- Print and fill out the reader comment form located at the back of this book. You can give the completed form to your local IBM branch office or IBM representative, or you can send it to the address printed on the reader comment form.

Summary of changes to this book

The principle changes to this book are:

- Chapter 2, "JDBC application programming," on page 5 contains new explanations and examples of JDBC methods. It also contains descriptions of JDBC 2.0 and selected JDBC 3.0 functions.
- Chapter 3, "SQLJ application programming," on page 61 contains explanations of new SQLJ capabilities that are associated with JDBC 2.0 and JDBC 3.0 functions.
- Chapter 5, "Creating Java stored procedures and user-defined functions" contains information on writing and running Java routines.
- "Special considerations for CICS applications" contains information on running JDBC™ and SQLJ programs in the CICS environment.
- Information on the DB2 Universal JDBC Driver has been added.
- Information on compiled Java stored procedures has been removed.
- VisualAge for Java information has been deleted.

Chapter 1. Introduction to Java application support

DB2® Universal Database 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 Universal Database™ 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.

This topic discusses the Java application development environment provided by DB2 Universal Database.

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 JDBC-ODBC Bridge driver is an example of 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.

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 UDB for OS/390® or z/OS® supports a type 2 driver and a driver that combines type 2 and type 4 JDBC implementations. The drivers that are supported in DB2 UDB for OS/390 or z/OS are:

DB2 Universal JDBC driver (type 2 and type 4):

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

The DB2 Universal JDBC Driver is an entirely new driver, rather than a follow-on to any other DB2 JDBC drivers. Therefore, you can expect some differences in behavior between this driver and other drivers.

The DB2 Universal JDBC Driver supports these JDBC and SQLJ functions:
- Most of the methods that are described in the JDBC 1.2 and JDBC 2.0 specifications, and some of the methods that are described in the JDBC 3.0 specifications. See "Comparison of driver support for JDBC APIs" on page 107.
- SQLJ statements that perform equivalent functions to all JDBC methods.
- Connections that are enabled for connection pooling. WebSphere Application Server or another application server does the connection pooling.
- Implementation of Java user-defined functions and stored procedures (Universal Driver 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 global transactions (*Distributed Transaction Processing: The XA Specification*, available from http://www.opengroup.org(Universal Driver type 4 connectivity to DB2 UDB for OS/390 Version 7, DB2 UDB for z/OS Version 8, or DB2 UDB for Linux, UNIX and Windows).

In general, you should use Universal Driver type 2 connectivity for Java programs that run on the same z/OS system or zSeries® logical partition (LPAR) as the target DB2 subsystem. Use Universal Driver type 4 connectivity for Java programs that run on a different z/OS system or LPAR from the target DB2 subsystem.

For z/OS systems or LPARs that do not have DB2 UDB for z/OS, the z/OS Application Connectivity to DB2 for z/OS optional feature can be installed to provide Universal Driver type 4 connectivity to a DB2 UDB for z/OS or DB2 UDB for Linux, UNIX, and Windows server.

To use the DB2 Universal JDBC Driver, you need Java 2 Technology Edition, SDK 1.3.1 or higher. To implement Java stored procedures or user-defined functions, you need Java 2 Technology Edition, SDK 1.3.1, SDK 1.4.1, or higher.

JDBC/SQLJ Driver for OS/390 and z/OS with JDBC 2.0 support (JDBC/SQLJ 2.0 Driver for OS/390 and z/OS):

The JDBC/SQLJ 2.0 Driver for OS/390 and z/OS is a type 2 driver that contains most of the functions that are described in the JDBC 1.2 specification. This driver also includes some of the functions that are described in the JDBC 2.0 specification. See "Comparison of driver support for JDBC APIs" on page 107 for a list of the JDBC methods that the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS supports.

The JDBC/SQLJ 2.0 Driver for OS/390 and z/OS supports these functions:
- Global transactions that run under WebSphere® Application Server Version 4.0 and above
- Implementation of Java user-defined functions and stored procedures
- SQLJ statements that perform equivalent functions to all JDBC methods

- Connection pooling

To use this driver, you need Java 2 Technology Edition, SDK 1.3 or higher. To implement Java stored procedures or user-defined functions, you need Java 2 Technology Edition, SDK 1.3.1, SDK 1.4.1 or higher.

The JDBC/SQLJ Driver for OS/390 and z/OS will not be supported in future releases of DB2. You should therefore consider moving to the DB2 Universal JDBC Driver.

Chapter 2. JDBC application programming

The following topics explain DB2 UDB for z/OS JDBC application support:
- "Basic JDBC application programming concepts (for all DB2 UDB for z/OS JDBC drivers)"
- "Advanced JDBC application programming concepts" on page 27
- "JDBC application programming concepts for the JDBC/SQLJ Driver for OS/390 and z/OS" on page 53

Basic JDBC application programming concepts (for all DB2 UDB for z/OS JDBC drivers)

The following topics contain basic information about writing JDBC applications that applies to all DB2 UDB for z/OS drivers:
- "Basic steps in writing a JDBC application"
- "Java packages for JDBC support" on page 8
- "Variables in JDBC applications" on page 8
- "How JDBC applications connect to a data source" on page 8
- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 10
- "Connecting to a data source using the DataSource interface" on page 12
- "How to determine which type of DB2 Universal JDBC Driver connectivity to use" on page 14
- "Setting the isolation level for a JDBC transaction" on page 15
- "JDBC connection objects" on page 16
- "Committing or rolling back JDBC transactions" on page 16
- "Closing a connection to a JDBC data source" on page 16
- "JDBC interfaces for executing SQL" on page 17
- "Creating and modifying DB2 objects using the Statement.executeUpdate method" on page 17
- "Retrieving data from DB2 tables using the Statement.executeQuery method" on page 18
- "Updating data in DB2 tables using the PreparedStatement.executeUpdate method" on page 19
- "Retrieving data from DB2 using the PreparedStatement.executeQuery method" on page 20
- "Calling stored procedures using CallableStatement methods" on page 21
- "Handling an SQLException under the DB2 Universal JDBC Driver" on page 22
- "Handling an SQLWarning under the DB2 Universal JDBC Driver" on page 26

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 is a simple program that demonstrates each task. This program runs on the DB2 Universal JDBC Driver.

```
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 DB2 Universal JDBC Driver topic.
    // For example, for Universal Driver type 2 connectivity,
    // args[0] might be MVS1DB2M. For Universal
    // Driver 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 DB2 Universal JDBC Driver
      Class.forName("com.ibm.db2.jcc.DB2Driver");                           3a
      System.out.println("**** Loaded the JDBC driver");

      // Create the connection using the DB2 Universal JDBC Driver
      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("**** Creaed 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 6:

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 "Java packages for JDBC support" on page 8.

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 "Variables in JDBC applications" on page 8 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 "How JDBC applications connect to a data source" on page 8 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 "JDBC interfaces for executing SQL" on page 17.

5 This `try/catch` block demonstrates the use of the `SQLException` class for SQL error handling. For more information on handling SQL errors, see "Handling an SQLException under the DB2 Universal JDBC Driver" on page 22 and "Handling an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 55. For information on handling SQL warnings, see "Handling an SQLWarning under the DB2 Universal JDBC Driver" on page 26.

6 This statement disconnects the application from the data source. See "Closing a connection to a JDBC data source" on page 16.

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 JDBC 2.0 standard extensions.

com.ibm.db2.jcc
Contains the DB2-specific implementation of JDBC for the DB2 Universal JDBC driver and some functions of the JDBC/SQLJ Driver for OS/390®.

COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver
Contains some functions of the DB2-specific implementation of JDBC/SQLJ Driver for OS/390.

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" on page 5 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" on page 127 shows the recommended mappings of Java data types and JDBC data types to SQL data types.

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 on page 9 shows how a Java™ application connects to a data source for a type 2 driver or DB2 Universal JDBC Driver type 2 connectivity.

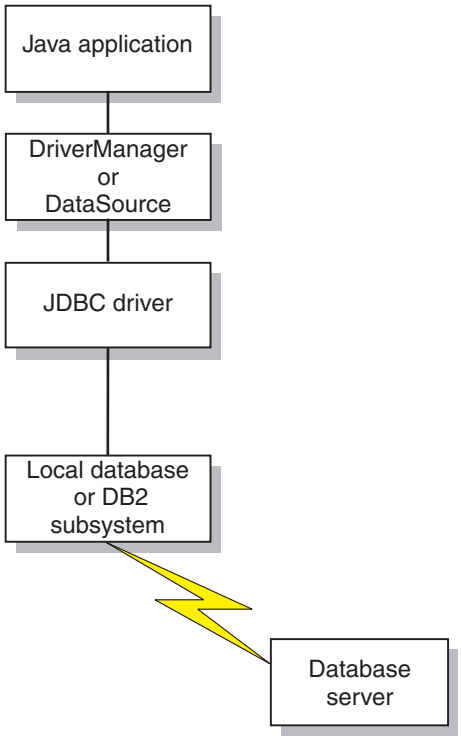
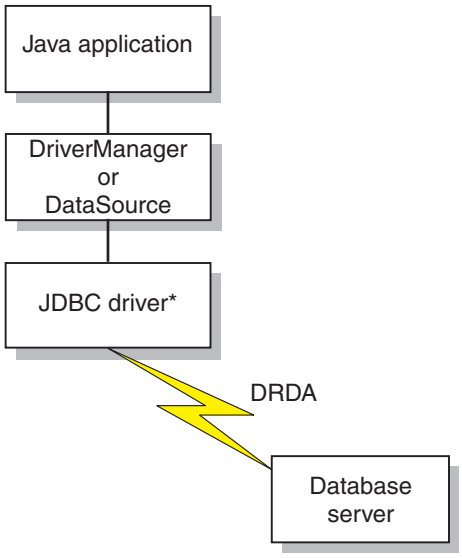


Figure 2. Java application flow for a type 2 driver or DB2 Universal JDBC Driver type 2 connectivity

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



*Java byte code executed under JVM

Figure 3. Java application flow for DB2 Universal JDBC Driver type 4 connectivity

The way that you connect to a data source depends on the version of JDBC that you use. Connecting using the `DriverManager` interface is available for all levels of JDBC. Connecting using the `DataSource` interface is available with JDBC 2.0 and above.

Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC 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 Universal JDBC Driver, 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 DB2 Universal JDBC Driver:

```
try {
  // Load the DB2® Universal JDBC Driver 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.

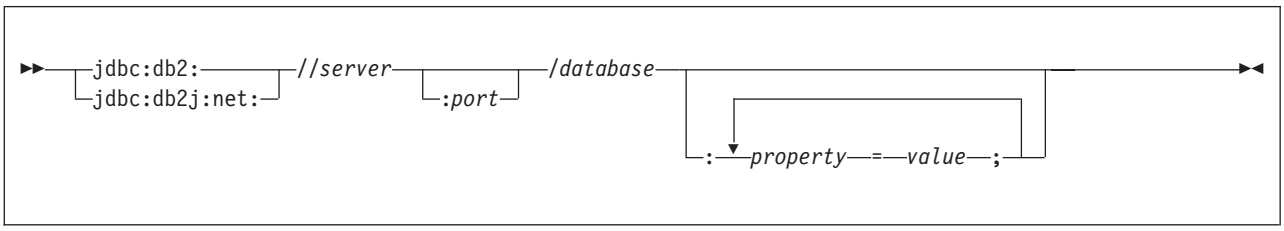
After you load the driver, you connect to the data source by invoking the `DriverManager.getConnection` method. You can use one of the following forms of `getConnection`:

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

The *url* argument represents a data source, and indicates what type of JDBC connectivity you are using.

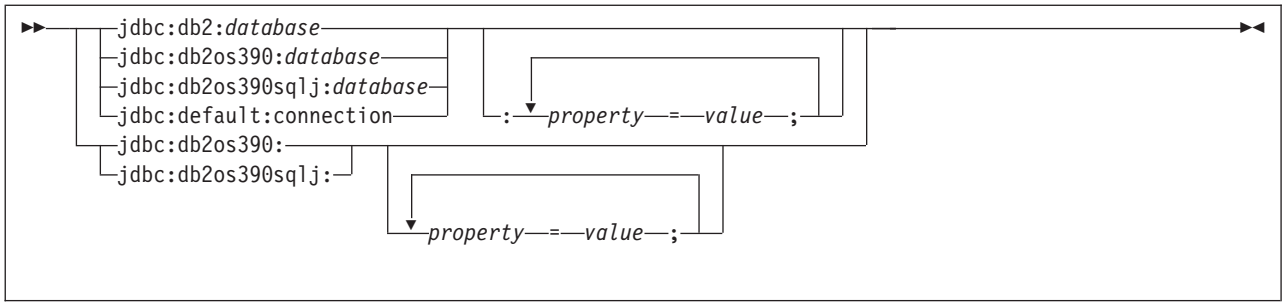
For DB2 Universal JDBC Driver type 4 connectivity, specify a URL of the following form:

Syntax for a URL for Universal Driver type 4 connectivity:



For DB2 Universal JDBC Driver type 2 connectivity, specify a URL of one of the following forms:

Syntax for a URL for Universal Driver type 2 connectivity:



The parts of the URL have the following meanings:

jdbc:db2: or jdbc:db2j:net: or jdbc:db2os390: or jdbc:db2os390sqlj: or jdbc:default:connection
The meanings of the initial portion of the URL are:

jdbc:db2: or jdbc:db2os390: or jdbc:db2os390sqlj:
Indicates that the connection is to a DB2 UDB for z/OS or DB2 UDB for Linux, UNIX, and Windows server. jdbc:db2os390: and jdbc:db2os390sqlj: are for compatibility of programs that were written for the JDBC/SQLJ Driver for OS/390.

db2:default:connection
Indicates that the URL is intended for environments that support an already-existing connection, such as CICS, IMS™, and stored procedures.

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 Universal Driver type 4 connectivity or Universal Driver type 2 connectivity is used.

For Universal Driver type 4 connectivity:

- If the connection is to a DB2 UDB 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 DB2 Universal JDBC Driver does not convert lowercase characters in the database value to uppercase for Universal Driver 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 UDB for z/OS server, all characters in *database* must be uppercase characters.
- If the connection is to a DB2 UDB 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 Universal Driver type 2 connectivity:

- *database* is a location name that is defined in the SYSIBM.LOCATIONS catalog table.

  All characters in the DB2 location name must be uppercase characters. However, when the connection is to a DB2 UDB for z/OS server, the DB2 Universal JDBC Driver converts lowercase characters in the database value to uppercase for Universal Driver type 2 connectivity.
- If the connection is to 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 DB2 Universal JDBC Driver" on page 185.

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 DB2 Universal JDBC Driver" on page 185 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
```

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. It is possible to create and use the DataSource object in the same application, as you do with the DriverManager interface. Figure 4 shows an example for the DB2 Universal JDBC Driver:

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

```
import java.sql.*;        // JDBC base
import javax.sql.*;       // JDBC 2.0 standard extension APIs
import com.ibm.db2.jcc.*; // DB2® Universal JDBC Driver   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
```

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 "Creating and deploying DataSource objects" on page 49 for information on DB2's DataSource implementations.

3 The setDatabaseName, setDescription, setUser, and setPassword methods assign attributes to the DB2SimpleDataSource object. See "DataSource properties for the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS" on page 196 for information about the attributes that you can set for a DB2SimpleDataSource object under the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS. See "Properties for the DB2 Universal JDBC Driver" on page 185 for information about the attributes that you can set for a DB2SimpleDataSource object under the DB2 Universal JDBC Driver.

4 Establishes a connection to the data source that DB2SimpleDataSource object db2ds represents.

However, a more flexible 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 "Creating and deploying DataSource objects" on page 49.

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
```

How to determine which type of DB2 Universal JDBC Driver connectivity to use

The DB2 Universal JDBC Driver 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 1. Comparison of Universal Driver type 2 connectivity and Universal Driver type 4 connectivity

Function	Universal Driver type 2 connectivity support	Universal Driver type 4 connectivity support
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

Table 1. Comparison of Universal Driver type 2 connectivity and Universal Driver type 4 connectivity (continued)

Function	Universal Driver type 2 connectivity support	Universal Driver type 4 connectivity support
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
CICS environment	Supported	Not supported
IMS environment	Supported	Not supported

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

Use Universal Driver 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.
- Your application runs in the CICS environment or IMS environment.

Use Universal Driver 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 do not have DB2 installed locally.

  Universal Driver type 2 connectivity relies on code that is part of DB2, but Universal Driver type 4 connectivity does not. Therefore, for Universal Driver type 4 connectivity, you do not need to have DB2 installed where the driver runs.
- You are using DB2 Universal JDBC Driver connection concentrator and Sysplex workload balancing support.

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 2 shows the values of *level* that you can specify in the `Connection.setTransactionIsolation` method and their DB2® equivalents.

Table 2. 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

With the JDBC/SQLJ Driver for OS/390 and z/OS, you can change the isolation level only at the beginning of a transaction.

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 "JDBC interfaces for executing SQL" on page 17.
- Gather information about the data source to which you are connected. This process is discussed in "Learning about a data source using DatabaseMetaData methods" on page 39.
- Commit or roll back transactions. You can commit transactions manually or automatically. These operations are discussed in "Committing or rolling back JDBC transactions."
- Close the connection to the data source. This operation is discussed in "Closing a connection to a JDBC data source."

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, DB2® performs a commit operation after every SQL statement completes. To determine whether autocommit mode is on, invoke the `Connection.getAutoCommit` method. To set autocommit mode on, invoke the `Connection.setAutoCommit(true)` method. To set autocommit mode off, invoke the `Connection.setAutoCommit(false)` method.

Connections that participate in global transactions cannot invoke the `setAutoCommit(true)` method. See Chapter 12, "JDBC and SQLJ global transaction support," on page 307 for information on global transactions.

Closing a connection to a JDBC data source

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.

JDBC interfaces for executing SQL

You execute SQL statements in a traditional SQL program to insert, update, and delete 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 DB2 Universal JDBC Driver, 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 DB2 Universal JDBC Driver, you can also use the `Statement` interface to call stored procedures, but those stored procedures must have no parameters. For the JDBC/SQLJ Driver for OS/390, you must use the `CallableStatement` interface, even if the stored procedure has 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" on page 107.

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 and DELETE statements that do not contain parameter markers
- With the DB2 Universal JDBC Driver, 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 6. Using Statement.executeUpdate

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 DB2 Universal JDBC Driver, 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 "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 36 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 DB2 Universal JDBC Driver also supports updatable and scrollable `ResultSets`. These are discussed in "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 46.

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" on page 107 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 7. Using Statement.executeQuery

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 DB2 Universal JDBC Driver, 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 value to first parameter    2
pstmt.setString(2,"000010");      // Assign value to second parameter
numUpd = pstmt.executeUpdate();   // Perform the update                 3
pstmt.close();                    // Close the PreparedStatement object 4
```

Figure 8. 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 9. Using PreparedStatement.executeUpdate for an SQL statement without parameter markers

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 DB2 Universal JDBC Driver, 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 "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 36 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 10. 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`.

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 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.
3. Invoke the `CallableStatement.registerOutParameter` method to indicate which parameters are output-only (OUT) parameters, or input and output (INOUT) parameters.
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.

5. If the stored procedure returns result sets, retrieve the result sets. See "Retrieving multiple result sets from a stored procedure in a JDBC application" on page 36.
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 11. Using CallableStatement methods for a stored procedure call with parameter markers

Handling an SQLException under the DB2 Universal JDBC 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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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 for the existence of a DB2Diagnosable object. If the object exists:
      1) Optional: Invoke the DB2Diagnosable.printTrace method to write all SQLException information to a java.io.PrintWriter object.
      2) Invoke the DB2Diagnosable.getThrowable method to determine whether an underlying java.lang.Throwable caused the SQLException.
      3) Invoke the DB2Diagnosable.getSqlca method to retrieve the DB2Sqlca object.
      4) Invoke the DB2Sqlca.getSqlCode method to retrieve an SQL error code value.
      5) 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.
      6) Invoke the DB2Sqlca.getSqlErrp method to retrieve the SQLERRP value.
      7) Invoke the DB2Sqlca.getSqlErrd method to retrieve the SQLERRD values in an array.
      8) Invoke the DB2Sqlca.getSqlWarn method to retrieve the SQLWARN values in an array.
      9) Invoke the DB2Sqlca.getSqlState method to retrieve the SQLSTATE value.

10) 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 DB2 Universal JDBC Driver. 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;
          diagnosable.printTrace (printWriter, "");                     3b1
          java.lang.Throwable throwable =
            diagnosable.getThrowable();                                 3b2
          if (throwable != null) {
            // Extract java.lang.Throwable information
            // such as message or stack trace.
            ...
          }
          DB2Sqlca sqlca = diagnosable.getSqlca();                      3b3
                                        // Get DB2Sqlca object
          if (sqlca != null) {          // Check that DB2Sqlca is not null
            int sqlCode = sqlca.getSqlCode(); // Get the SQL error code  3b4
            String sqlErrmc = sqlca.getSqlErrmc();                      3b5
                                        // Get the entire SQLERRMC
            String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
                                        // You can also retrieve the
                                        // individual SQLERRMC tokens
            String sqlErrp = sqlca.getSqlErrp();                        3b6
                                        // Get the SQLERRP
            int[] sqlErrd = sqlca.getSqlErrd();                         3b7
                                        // Get SQLERRD fields
            char[] sqlWarn = sqlca.getSqlWarn();                        3b8
                                        // Get SQLWARN fields
            String sqlState = sqlca.getSqlState();                      3b9
                                        // Get SQLSTATE
            String errMessage = sqlca.getMessage();                     3b10
                                        // 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 12. Processing an SQLException under the DB2 Universal JDBC Driver (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 12. Processing an SQLException under the DB2 Universal JDBC Driver (Part 2 of 2)

Handling an SQLWarning under the DB2 Universal JDBC 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.

Important:

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

When a call to `ResultSet.next` returns no rows, the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver, 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 13. Processing an SQLWarning

For an example of obtaining DB2-specific error information, see "Handling an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 55.

Advanced JDBC application programming concepts

The following topics contain more advanced information about writing JDBC applications that applies to all DB2 UDB for z/OS drivers:

- "Learning about parameters in a PreparedStatement using ParameterMetaData methods" on page 40
- "Making batch updates in JDBC applications" on page 41
- "Making batch queries in JDBC applications" on page 43
- "Retrieving information from a BatchUpdateException" on page 44
- "Characteristics of a JDBC ResultSet under the DB2 Universal JDBC Driver" on page 45
- "Specifying updatability, scrollability, and holdability for ResultSets in JDBC applications" on page 46
- "Creating and deploying DataSource objects" on page 49
- "Providing extended client information to the DB2 server with the DB2 Universal JDBC Driver" on page 50
- "System monitoring for the DB2 Universal JDBC Driver" on page 51

LOBs in JDBC applications with the DB2 Universal JDBC Driver

The DB2 Universal JDBC Driver includes all of the LOB support in the JDBC 2.0 specification, and some of the LOB support in the JDBC 3.0 specification. This driver also includes support for LOBs in additional methods and for additional data types.

CLOB data is always sent to the database server as a Unicode stream. The database server converts the data to the target code page.

LOB locator support: The DB2 Universal JDBC Driver can use LOB locators to retrieve data in LOB columns. To cause JDBC to use LOB locators to retrieve data from LOB columns, you need to set the `fullyMaterializeLobData` property to `false`. Properties are discussed in "Properties for the DB2 Universal JDBC Driver" on page 185.

`fullyMaterializeLobData` has no effect on stored procedure parameters or LOBs that are fetched using scrollable cursors. When you fetch data from a DB2 UDB server in the OS/390® or z/OS® environment using scrollable cursors, JDBC 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 DB2 subsystem. You cannot use a single LOB locator to move data between two different DB2 subsystems. To move LOB data between two DB2 subsystems, you need to materialize the LOB data when you retrieve it from a table in the first DB2 subsystem and then insert that data into the table in the second DB2 subsystem.

JDBC 3.0 methods supported by the DB2 Universal JDBC Driver: In addition to the methods in the JDBC 2.0 specification, the DB2 Universal JDBC Driver includes LOB support in the following JDBC 3.0 methods:

- You can update a BLOB column with the following `Blob` methods. The BLOB value must be fully materialized, which means that the `fullyMaterializeLobData` property value must be true.
  - `setBinaryStream`
  - `setBytes`
  - `truncate`
- You can update a CLOB column with the following `Clob` methods. The CLOB value must be fully materialized, which means that the `fullyMaterializeLobData` property value must be true.
  - `setAsciiStream`

– setCharacterStream
– setString
– truncate

Additional methods supported by the DB2 Universal JDBC Driver: In addition to the methods in the JDBC specification, the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver: If you are using Universal Driver type 2 connectivity, you cannot call a stored procedure that has DBCLOB OUT or INOUT parameters.

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

For Universal Driver type 2 connectivity to DB2® UDB for z/OS®, when the JDBC driver processes a `CallableStatement.setXXX` call for a stored procedure input parameter, or a `CallableStatement.registerOutParameter` call for a stored procedure output parameter, the driver cannot determine the parameter data types.

When the deferPrepares property is set to true, and the DB2 Universal JDBC Driver 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.

ROWIDs in JDBC with the DB2 Universal JDBC Driver

DB2® UDB 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 DB2 Universal JDBC Driver 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)
```

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 14. Creating and using a distinct type

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 DB2 Universal JDBC Driver 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.

Although the JDBC/SQLJ Driver for OS/390 and z/OS does not support these methods, you can still set savepoints, release savepoints, and roll back to savepoints by executing the SAVEPOINT, RELEASE SAVEPOINT, and ROLLBACK TO SAVEPOINT SQL statements using the `executeUpdate` or `execute` methods.

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
stmt.executeUpdate(
  "CREATE DISTINCT TYPE SHOESIZE AS INTEGER");
                                            // Create distinct type
con.commit();                               // Commit the create
stmt.executeUpdate(
  "CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)");
                                            // Create table with distinct type
con.commit();                               // Commit the create
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 15. Setting, rolling back to, and releasing a savepoint in a JDBC application

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, DECIMAL with a scale of zero, or a distinct type based on one of these types).

If you are using the DB2 Universal JDBC Driver, 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.

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

     Use the following form only for a table on any database server that supports identity columns and INSERT within SELECT.

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

     Use the following form for a table on any database server that supports identity columns.

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

     Use the following form only for a table on any database server that supports identity columns and INSERT within SELECT.

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

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

     Use the following form only for a table on any database server that supports identity columns and INSERT within SELECT.

     ```
     Statement.execute(sql-statement, String [] columnNames);
     ```
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 " +                          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 16. Retrieving automatically generated keys

With any JDBC driver, you can retrieve the most recently assigned value of an identity column using the DB2 UDB IDENTITY_VAL_LOCAL() built-in function. Execute code similar to this:

```
String idntVal;
Connection con;
Statement stmt;
ResultSet rs;
...
stmt = con.createStatement();     // Create a Statement object
rs = stmt.executeQuery("SELECT IDENTITY_VAL_LOCAL() FROM SYSIBM.SYSDUMMY1");
                                  // Get the result table from the query.
                                  // This is a single row with the most
                                  // recent identity column value.
while (rs.next()) {               // Position the cursor
 idntVal = rs.getString(1);       // Retrieve column value
 System.out.println("Identity column value = " + idntVal);
                                  // Print the column value
}
rs.close();                       // Close the ResultSet
stmt.close();                     // Close the Statement
```

Figure 17. Using IDENTITY_VAL_LOCAL() to return the most recent value of an identity column

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 or the `PreparedStatement.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 18. 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 "Learning about a ResultSet using ResultSetMetaData methods" on page 38 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 "Learning about a ResultSet using ResultSetMetaData methods" on page 38.
   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 19. Retrieving unknown result sets from a stored procedure

Keeping result sets open:

In Figure 19, invocation of getMoreResults() closes the ResultSet object that is returned by the previous invocation of getResultSet. However, if you are using the DB2 Universal JDBC Driver, 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. This form of getMoreResults requires JDK 1.4 or later.

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 that were previously kept open.

For example, the code in Figure 20 keeps all ResultSets open until the final ResultSet has been retrieved, and then closes all ResultSets.

```
CallableStatement cstmt;
ResultSet rs;
...
boolean resultsAvailable = cstmt.execute(); // Call the stored procedure
while (resultsAvailable) {                  // Test for result sets
 ResultSet rs = 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
}
resultsAvailable = cstmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);
                                            // Close the result sets
```

Figure 20. Keeping retrieved stored procedure result sets open

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 21. Using ResultSetMetaData methods to get information about a ResultSet

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 2.0 level before you invoke JDBC 2.0 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 2.0 functions, such as batch updates or scrollable ResultSets
- A list of scalar functions that the driver supports

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

1. Create a DatabaseMetaData object by invoking the getMetaData method on the connection.
2. Invoke DatabaseMetaData methods to get information about the data source.
3. If the method returns a ResultSet:
   a. In a loop, position the cursor using the next method, and retrieve data from each column of the current row of the ResultSet object using getXXX methods.
   b. Invoke the close method to close the ResultSet object.

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 22. Using DatabaseMetaData methods to get information about a data source

Learning about parameters in a PreparedStatement using ParameterMetaData methods

The DB2 Universal JDBC Driver 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;
int 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.getParameterType(i);                                 4
                                  // Get SQL type for each parameter
 System.out.println("SQL type of parameter " + i " is " + sqlType);
}
...
pstmt.close();                       // Close the PreparedStatement
```

Figure 23. Using ParameterMetaData methods to get information about a PreparedStatement

Making batch updates in JDBC applications

The JDBC drivers that support JDBC 2.0 and above support batch updates. 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. Disable `AutoCommit` for the `Connection` object.
2. Invoke the `createStatement` method to create a `Statement` object.
3. For each SQL statement that you want to execute in the batch, invoke the `addBatch` method.
4. Invoke the `executeBatch` method to execute the batch of statements.
5. 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. Invoke the `commit` method to commit the changes.

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

1. Disable `AutoCommit` for the `Connection` object.
2. Invoke the `prepareStatement` method to create a `PreparedStatement` object for the SQL statement with input parameters.
3. For each set of input parameter values:
   a. Execute `setXXX` methods to assign values to the input parameters.
   b. Invoke the `addBatch` method to add the set of input parameters to the batch.
4. Invoke the `executeBatch` method to execute the statements with all sets of parameters.
5. 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. Invoke the `commit` method to commit the changes.

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 {
...
  connection con.setAutoCommit(false);                    1
  PreparedStatement prepStmt = con.prepareStatement(
    "UPDATE DEPT SET MGRNO=? WHERE DEPTNO=?");            2
  prepStmt.setString(1,mgrnum1);                          3a
  prepStmt.setString(2,deptnum1);
  prepStmt.addBatch();                                    3b

  prepStmt.setString(1,mgrnum2);
  prepStmt.setString(2,deptnum2);
  prepStmt.addBatch();
  int [] numUpdates=prepStmt.executeBatch();              4
  for (int i=0; i < numUpdates.length; i++) {             5a
    if (numUpdates[i] == -2)
      System.out.println("Execution " + i +
        ": unknown number of rows updated");
    else
      System.out.println("Execution " + i +
        "successful: " numUpdates[i] + " rows updated");
  }
  con.commit();                                           5b
} catch(BatchUpdateException b) {
  // process BatchUpdateException
}
```

Figure 24. Performing a batch update

Making batch queries in JDBC applications

The DB2 Universal JDBC Driver 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 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(
    "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 25. Performing a batch query

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 updated. `getUpdateCounts` returns -2 if the number of updated rows cannot be determined, or -3 if an error occurred during an update.
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 26. 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.

Characteristics of a JDBC ResultSet under the DB2 Universal JDBC Driver

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 DB2 Universal JDBC Driver provides the capability to do these things.

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* or *sensitive*. Insensitive means that changes to the underlying table after the cursor is opened are not visible to the 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 DB2 Universal JDBC Driver" on page 185 for more information about the cursorSensitivity property.

Important: Like static scrollable cursors in any other language, JDBC static scrollable `ResultSets` use declared temporary tables for their internal processing. This means that before you can execute any applications that contain JDBC static scrollable `ResultSets`, your database administrator needs to create a temporary database and temporary table spaces for those declared temporary tables. See Part 2 of *DB2 Installation Guide* for detailed information on creating the temporary database and temporary table spaces.

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 you cannot insert into a result table, and 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" on page 107 for a complete list of the methods that are supported for `ResultSets`.

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 3 for a list of valid values for *resultSetType* and *resultSetConcurrency*.

Table 3. 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.

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 4.

Table 4. 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.

Table 4. ResultSet methods for positioning a scrollable cursor (continued)

Method	Positions the cursor
`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 46, 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
  updateString("PHONENO","4657");      // Update their phone number
  updateRow();                         // Update the row
 }
}
rs.close();                            // Close the ResultSet          5
stmt.close();                          // Close the Statement          6
```

Figure 27. Using a scrollable cursor

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 DB2 Universal JDBC Driver provides the following `DataSource` implementations:

- `com.ibm.db2.jcc.DB2SimpleDataSource`, which does not support connection pooling. You can use this implementation with Universal Driver type 2 connectivity or Universal Driver 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 Universal Driver type 4 connectivity.

The JDBC/SQLJ Driver for OS/390® provides the following `DataSource` implementations:

- `com.ibm.db2.jcc.DB2SimpleDataSource`, which does not contain built-in connection pooling.

Because CICS® does contain built-in connection pooling, you need to use this class for CICS applications.

- `com.ibm.db2.jcc.DB2DataSource`, which contains built-in connection pooling.

  See Chapter 10, "JDBC and SQLJ connection pooling support," on page 299 for a discussion of connection pooling.

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 28 shows how to perform these tasks.

```
import java.sql.*;        // JDBC base
import javax.naming.*;    // JNDI Naming Services
import javax.sql.*;       // JDBC 2.0 standard extension APIs
import com.ibm.db2.jcc.*; // DB2 implementation of JDBC 2.0
                          // 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 28. Example of creating and deploying a DataSource object

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.

Providing extended client information to the DB2 server with the DB2 Universal JDBC Driver

The DB2 Universal JDBC Driver 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 or workload management. 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 5.

Table 5. Methods that provide client information to the DB2 server

Method	Information provided
setDB2ClientUser	User name for a connection
setDB2ClientWorkstation	Client workstation name for a connection

Table 5. Methods that provide client information to the DB2 server (continued)

Method	Information provided
setDB2ClientApplicationInformation	Name of the application that is working with a connection
setDB2ClientAccountingInformation	Accounting information

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 5 on page 50.
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 29. Example of passing extended client information to a DB2 server

System monitoring for the DB2 Universal JDBC Driver

To assist you in monitoring the performance of your applications with the DB2 Universal JDBC Driver, the DB2SystemMonitor interface is provided. This interface contains methods that collect the following data 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 UDB 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.

To collect system monitoring data, 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 DB2 Universal JDBC Driver
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      System.out.println("**** Loaded the JDBC driver");

      // Create the connection using the DB2 Universal JDBC Driver
      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 DB2 Universal JDBC Driver, " + e);
    }
    catch(SQLException e)
    {
      System.out.println("SQLException: " + e);
      e.printStackTrace();
    }
  }
}
```

Figure 30. Example of using DB2SystemMonitor methods to collect system monitoring data

JDBC application programming concepts for the JDBC/SQLJ Driver for OS/390 and z/OS

The following topics contain information that applies only to the JDBC/SQLJ Driver for OS/390 and z/OS:

- "Connecting to a data source using the DriverManager interface with a JDBC/SQLJ Driver for OS/390 and z/OS" on page 54

Connecting to a data source using the DriverManager interface with a JDBC/SQLJ Driver for OS/390 and z/OS

A JDBC application establishes 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 JDBC/SQLJ Driver for OS/390 and z/OS, you load the driver by invoking the `Class.forName` method with one of the following arguments:

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

  This is the preferred name for the JDBC/SQLJ Driver for OS/390 and z/OS.
- `ibm.sql.DB2Driver`

  This name is available only to maintain compatibility with older DB2 UDB for z/OS JDBC applications. The `ibm.sql.DB2Driver` class automatically forwards all driver API calls to the `COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`.

The following code demonstrates loading a JDBC/SQLJ Driver for OS/390 and z/OS:

```
try {
  // Load the DB2 for z/OS driver
  Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
} 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, String user, String password);
getConnection(String url, java.util.Properties info);
```

The *url* argument of the `getConnection` method represents the data source. Specify one of the following *url* values for a DB2 UDB for z/OS data source:

```
jdbc:db2os390:location-name
jdbc:db2os390sqlj:location-name
```

Each format results in the same behavior. Both forms are provided for compatibility with existing DB2 UDB for z/OS JDBC applications.

If *location-name* is not the name of the local DB2 subsystem, *location-name* must be defined in the SYSIBM.LOCATIONS catalog table. If *location-name* is the local site,

location-name must have been specified in field DB2 LOCATION NAME of the DISTRIBUTED DATA FACILITY panel during DB2 installation.

In addition to the URL values shown above for a DB2 UDB for z/OS data source, the following URL has a special meaning for type 2 drivers.

```
jdbc:default:connection
```

This URL is intended for environments that support an already-existing connection, such as CICS, IMS, and stored procedures.

For some connections, you need to specify a user ID and password. To do that, use the form of the `getConnection` method that specifies *user* and *password*, or the form that specifies *info*.

The *info* argument is an object of type `java.util.Properties` that contains a set of driver properties for the connection. For the JDBC/SQLJ Driver for OS/390 and z/OS, you should specify only the `user` and `password` properties.

The following example demonstrates how to specify the user ID and password as properties when you create a connection to a data source:

```
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:db2os390:san_jose";
                                          // Set URL for data source
Connection con = DriverManager.getConnection(url, properties);
                                          // Create connection
```

Do not specify a user ID or password for a CICS or IMS connection.

Handling an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS

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 JDBC/SQLJ Driver for OS/390 and z/OS 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 method, 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

The basic steps for handling an SQLException in a JDBC program that runs under the JDBC/SQLJ Driver for OS/390 and z/OS are:

1. Give the program access to the com.ibm.db2.jcc.DB2Diagnosable interface and the com.ibm.db2.jcc.DB2Sqlca class. You can do that by importing them:

   ```
   com.ibm.db2.jcc.DB2Diagnosable
   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. Invoke the SQLException.getMessage method to retrieve the error description.
   c. Invoke the SQLException.getSQLState method to retrieve the SQLSTATE value.
   d. Invoke the SQLException.getErrorCode method to retrieve an SQL error code value.
   e. Check whether the current SQLException is an instance of a DB2Diagnosable object. If so:
      1) Invoke the DB2Diagnosable.getSqlca method to retrieve the DB2Sqlca object.
      2) Invoke the DB2Sqlca.getSqlCode method to retrieve an SQL error code value.
      3) 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.
      4) Invoke the DB2Sqlca.getSqlErrp method to retrieve the SQLERRP value.
      5) Invoke the DB2Sqlca.getSqlErrd method to retrieve the SQLERRD values in an array.
      6) Invoke the DB2Sqlca.getSqlWarn method to retrieve the SQLWARN values in an array.
      7) Invoke the DB2Sqlca.getSqlState method to retrieve the SQLSTATE value.
   f. 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 JDBC/SQLJ Driver for OS/390 and z/OS. 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


...
try {                                                                   2
  // Code that could generate SQLExceptions
 ...
} catch(SQLException sqle) {
    while(sqle != null) {             // Check whether there are more  3a
                                      // SQLExceptions to process
      System.out.println ("SQLException: " + sqle +
        ".  Message=" + sqle.getMessage() +                             3b
        ".  SQLSTATE=" + sqle.getSQLState() +                           3c
        " Error code=" + sqle.getErrorCode());                          3d
                                      // Print out the standard SQLException
      sqle.printStackTrace();
      //=====> Optional DB2-only error processing
      if (sqle instanceof DB2Diagnosable) {                             3e
                                      // Check if DB2-only information exists
        com.ibm.db2.jcc.DB2Diagnosable diagnosable =
        DB2Sqlca sqlca = diagnosable.getSqlca();                        3e1
                                        // Get DB2Sqlca object
        if (sqlca != null) {            // Check that DB2Sqlca is not null
          int sqlCode = sqlca.getSqlCode(); // Get the SQL error code    3e2
          String sqlErrmc = sqlca.getSqlErrmc();                        3e3
                                        // Get the entire SQLERRMC
          String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
                                        // You can also retrieve the
                                        // individual SQLERRMC tokens
          String sqlErrp = sqlca.getSqlErrp();                          3e4
                                        // Get the SQLERRP
          int[] sqlErrd = sqlca.getSqlErrd();                           3e5
                                        // Get SQLERRD fields
          char[] sqlWarn = sqlca.getSqlWarn();                          3e6
                                        // Get SQLWARN fields
          String sqlState = sqlca.getSqlState();                        3e7
                                        // Get SQLSTATE
          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 31. Processing an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS (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      3f
      }
}
```

Figure 31. Processing an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS (Part 2 of 2)

Internal errors in the JDBC/SQLJ Driver for OS/390 and z/OS: Internal errors in the DB2 JDBC drivers generate `SQLException` objects for which the value that is returned by `SQLException.getSQLState` is FFFFF, and the value that is returned by `SQLException.getErrorCode` is a value that is not documented in *DB2 Messages*. These error code values are not DB2 SQL error codes but are values that are generated by the JDBC driver. If `SQLException.getSQLState` returns FFFFF, contact your IBM service representative.

Handling an SQLWarning under the JDBC/SQLJ Driver for OS/390 and z/OS

Handling of an SQL warning under the JDBC/SQLJ Driver for OS/390 and z/OS is the same as handling an SQL warning under the DB2 Universal JDBC Driver. See “Handling an SQLWarning under the DB2 Universal JDBC Driver” on page 26.

Using LOBs in JDBC applications with the JDBC/SQLJ Driver for OS/390 and z/OS

The JDBC/SQLJ Driver for OS/390 and z/OS includes all of the LOB support in the JDBC 2.0 specification. See “Comparison of driver support for JDBC APIs” on page 107 for a list of supported methods. The JDBC/SQLJ Driver for OS/390 and z/OS also includes support for LOBs in additional methods and for additional data types.

LOB locator support: The JDBC/SQLJ Driver for OS/390 and z/OS does not use LOB locators for its support of LOB data types. This means that when you you use the JDBC/SQLJ Driver for OS/390 and z/OS, and you retrieve data from a LOB column, you retrieve the entire LOB. If you retrieve very large LOBs in your JDBC applications, you might need to increase the size of the JDBC application address space.

Additional methods supported by the JDBC/SQLJ Driver for OS/390 and z/OS: In addition to the methods in the JDBC 2.0 specification, the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS 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:
  - `getAsciiStream`
  - `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:
  - `setBinaryStream`
  - `setBytes`
- You can use the following `PreparedStatement` methods to set the values for parameters that correspond to CLOB columns:
  - `setAsciiStream`
  - `setCharacterStream`
  - `setString`
  - `setUnicodeStream`
- You can retrieve the value of a JDBC `CLOB` parameter using the following `CallableStatement` method:
  - `getString`

Using DBCLOBs with the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS: You can retrieve data from or store data in DBCLOB columns. However, because Java and JDBC do not have an equivalent to the DB2 DBCLOB data type, your JDBC programs need to use methods that are defined for Clob data to pass data to or from DBCLOB columns.

Restrictions on using LOBs with the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS:

- You cannot call a stored procedure that has LOB locator parameters or DBCLOB parameters.
- Inherited `PreparedStatement` methods `setAsciiStream` and `setUnicodeStream` cannot be used to set `CLOB` input parameters in a `CallableStatement`. Inherited `PreparedStatement` methods `setBinaryStream` and `setBytes` cannot be used to set `BLOB` input parameters in a `CallableStatement`. Method `getBytes` cannot be used to retrieve `BLOB` output parameters in a `CallableStatement`.
- For the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS, the maximum size for a LOB parameter of type OUT or INOUT in a `CallableStatement` is 1MB. IN parameters can be longer.

Using ROWIDs with the JDBC/SQLJ Driver for OS/390 and z/OS

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

- `getBytes`

You can use the following `PreparedStatement` method to store data in a ROWID column:

- `setBytes`

Using graphic string constants in JDBC applications

In EBCDIC environments, graphic string constants in JDBC applications have the following form:

```
G'\uxxxx\uxxxx...\uxxxx'
```

xxxx is the Unicode value in hexadecimal that corresponds to the desired EBCDIC graphic character.

For example, an EBCDIC double-byte G has the hexadecimal value 42C7. The corresponding Unicode hexadecimal value is FF27. Therefore, in JDBC methods, you represent the graphic string constant for an EBCDIC double-byte G as:

```
G'\uFF27'
```

The following code demonstrates using the `Statement.executeUpdate` method to execute an SQL statement that contains a graphic string constant:

```
Connection con;
Statement stmt;
int numUpd;
...
stmt = con.createStatement();               // Create a Statement object
// GRAPHIC_TABLE has one VARGRAPHIC(10) column named VGCOL.
// At least one row contains the string "GRAPHIC" in double-byte
// EBCDIC characters. The Unicode equivalent of "GRAPHIC" is
// G'\uFF27\uFF32\uFF21\uFF30\uFF28\uFF29\uFF23'.
// Update "GRAPHIC" in all rows to "graphic" in double-byte
// EBCDIC characters. The Unicode equivalent of "graphic" is
// G'\uFF47\uFF52\uFF41\uFF50\uFF48\uFF49\uFF43'
numUpd = stmt.executeUpdate(
  "UPDATE GRAPHIC_TABLE " +
  "SET VGCOL=G'\uFF47\uFF52\uFF41\uFF50\uFF48\uFF49\uFF43' " +
  "WHERE VGCOL=G'\uFF27\uFF32\uFF21\uFF30\uFF28\uFF29\uFF23'");
                                            // Perform the update
stmt.close();                               // Close Statement object
```

Figure 32. Using graphic string constants in a JDBC application

Chapter 3. SQLJ application programming

The following topics explain DB2 UDB for z/OS SQLJ application support:
- "Basic SQLJ application programming concepts"
- "Advanced SQLJ application programming concepts" on page 85

Basic SQLJ application programming concepts

The following topics contain basic information about writing SQLJ applications:
- "Basic steps in writing an SQLJ application"
- "Java packages for SQLJ support" on page 64
- "Variables in SQLJ applications" on page 64
- "Comments in an SQLJ application" on page 66
- "Connecting to a data source using SQLJ" on page 66
- "Setting the isolation level for an SQLJ transaction" on page 71
- "Committing or rolling back SQLJ transactions" on page 71
- "Savepoints in SQLJ applications" on page 72
- "Closing the connection to a data source in an SQLJ application" on page 73
- "SQL statements in an SQLJ application" on page 73
- "Creating and modifying DB2 objects in an SQLJ application" on page 73
- "How an SQLJ application retrieves data from DB2 tables" on page 74
- "Using a named iterator in an SQLJ application" on page 74
- "Using a positioned iterator in an SQLJ application" on page 76
- "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 78
- "Multiple open iterators for the same SQL statement in an SQLJ application" on page 81
- "Multiple open instances of an iterator in an SQLJ application" on page 83
- "Calling stored procedures in an SQLJ application" on page 83
- "Handling SQL errors in an SQLJ application" on page 84
- "Handling SQL warnings in an SQLJ application" on page 85

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 33 on page 62 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 33. 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 33. Simple SQLJ application (Part 2 of 2)

Notes to Figure 33 on page 62:

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 "Java packages for SQLJ support" on page 64.

2 `String` variables hvmgr and hvdeptno are *host identifiers*, which are equivalent to DB2 host variables. See "Variables in SQLJ applications" on page 64 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 "Connecting to a data source using SQLJ" on page 66 for more details.

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 "SQL statements in an SQLJ application" on page 73.

5
: This `try/catch` block demonstrates the use of the `SQLException` class for SQL error handling. For more information on handling SQL errors, see "Handling SQL errors in an SQLJ application" on page 84. For more information on handling SQL warnings, see "Handling SQL warnings in an SQLJ application" on page 85.

6
: This is an example of a comment. For rules on including comments in SQLJ programs, see "Comments in an SQLJ application" on page 66.

7
: This statement closes the connection to the data source. See "Closing the connection to a data source in an SQLJ application" on page 73.

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.

COM.ibm.db2os390.sqlj.jdbc
: Contains classes and interfaces that are specific to the JDBC/SQLJ Driver for OS/390.

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

javax.sql
: Contains JDBC 2.0 standard extensions.

Variables in SQLJ applications

In DB2® programs in other languages, you use host variables to pass data between the application program and DB2. In SQLJ programs, you use *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.

For the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS, a Java identifier in a host expression can have any of the data types listed in the Java data type column of "Java, JDBC, and SQL data types" on page 127, except for Blob or Clob. For the DB2 Universal JDBC Driver, a Java identifier can have any of the data types listed in the Java data type column of "Java, JDBC, and SQL data types" on page 127. Data types that are specified in an iterator can be any of the types in the Java data type column of "Java, JDBC, and SQL data types" on page 127.

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. The 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 the host expression from left to right before assigning its value to DB2.
- Evaluates side effects, such as operations with postfix operators, according to normal Java rules. All host expressions are fully evaluated before any of their values are passed to DB2.
- Uses Java rules for rounding and truncation.

Therefore, if the value of yearsEmployed is 6 before the UPDATE statement is executed, the value that is assigned to column BONUS by the UPDATE statement is ((int)6/5)*500, or 500. After 500 is assigned to BONUS, the value of yearsEmployed is incremented.

Restrictions on variable names: Two strings have special meanings in SQLJ programs. Observe the following restrictions when you use these strings in your SQLJ programs:

- The string __sJT_ is a reserved prefix for variable names that are generated by SQLJ. Do not begin the following types of names with __sJT_:
  - Host expression names
  - Java variable names that are declared in blocks that include executable SQL statements

  - Names of parameters for methods that contain executable SQL statements
  - Names of fields in classes that contain executable SQL statements, or in classes with subclasses or enclosed classes that contain executable SQL statements
- The string _SJ is a reserved suffix for resource files and classes that are generated by SQLJ. Avoid using the string _SJ in class names and input source file names.

Comments in an SQLJ application

To document your 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 only within host expressions.

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 DB2 Universal JDBC Driver, invoke `Class.forName` this way:

     ```
     Class.forName("com.ibm.db2.jcc.DB2Driver");
     ```
   - For the JDBC/SQLJ Driver for OS/390 and z/OS, invoke `Class.forName` this way:

     ```
     Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
     ```
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

"Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 10 or "Connecting to a data source using the DriverManager interface with a JDBC/SQLJ Driver for OS/390 and z/OS" on page 54. 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.

If the data source is a DB2 UDB for OS/390® or z/OS® system, and you do not specify these parameters, DB2 uses the external security environment, such as the RACF® security environment, that was previously established for the user. For a CICS® connection, you cannot specify a user ID or password.

info
Specifies an object of type `java.util.Properties` that contains a set of driver properties for the connection. For the JDBC/SQLJ driver for z/OS, you should specify only the `user` and `password` properties. For the DB2 Universal JDBC Driver, you can specify any of the properties listed in "Properties for the DB2 Universal JDBC Driver" on page 185.

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 34. 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 66 in connection technique 1.

2. Load the driver.

   This is the same as step 2 on page 66 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 "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 10.

   The meanings of the *url*, *user*, and *password* parameters are the same as the meanings of the parameters in step 3 on page 66 of connection technique 1.
4. Invoke the constructor for the connection context class that you created in step 1 on page 67.

   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 35. Using connection technique 2 to connect to a data source

Connection technique 3: This technique uses the JDBC `DataSource` interface for creating the connection. Use this technique only if your JDBC driver is a JDBC 2.0 driver.

1. Execute an SQLJ connection declaration clause.

   This is the same as step 1 on page 66 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 "Connecting to a data source using the DataSource interface" on page 12.

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 66 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);
   ```

   For environments other than the environments for CICS, stored procedures, and user-defined functions, the default autocommit mode for a JDBC connection is `true`. To disable autocommit, invoke `setAutoCommit(false)`.

5. Invoke the constructor for the connection context class that you created in step 1 on page 68.

   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 36. Using connection technique 3 to connect to a data source

Connection technique 4 (DB2 Universal JDBC Driver 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 66 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 37. 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. 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.

In a stored procedure that runs on DB2 UDB in the OS/390 or z/OS environment, or for a CICS or IMS application, when you use the default connection, DB2 uses the implicit connection.

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

Table 6. 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.

With the JDBC/SQLJ Driver for OS/390 and z/OS, you can change the isolation level only at the beginning of a transaction.

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};
```

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 DB2 Universal JDBC Driver, 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
#sql [ctxt] {CREATE DISTINCT TYPE SHOESIZE AS INTEGER WITH COMPARISONS};
                              // Create a distinct type
#sql [ctxt] {COMMIT};
                              // Commit the create
#sql [ctxt]
  {CREATE TABLE EMP_SHOE (EMPNO CHAR(6), EMP_SHOE_SIZE SHOESIZE)};
                              // Create table with distinct type
#sql [ctxt] {COMMIT};
                              // 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
ctxt.close();                  // Close the connection context
```

Figure 38. Setting, rolling back to, and releasing a savepoint in an SQLJ application

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 the `ConnectionContext.close()` method. This closes the connection context, as well as the connection to the data source. For example:

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

SQL statements in an SQLJ application

You execute SQL statements in a traditional SQL program to create tables, insert, update, and delete 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};
```

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 "Controlling the execution of SQL statements in SQLJ" on page 95 for information about when you need an execution context.

For complete information on SQLJ syntax, see Chapter 4, "JDBC and SQLJ reference," 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 UPDATE, and searched 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 "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 78.

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 "Using scrollable iterators in an SQLJ application" on page 102.

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.

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" on page 127 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 "Using SQLJ and JDBC in the same application" on page 86 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" on page 140 and "SQLJ context-clause" on page 138 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 39. Using a named iterator

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. See"Java, JDBC, and SQL data types" on page 127 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 "Using SQLJ and JDBC in the same application" on page 86 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 40. Using a positioned iterator

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 DB2 Universal JDBC Driver, the behavior is always DYNAMICRULES BIND. See the discussion of authorization IDs and dynamic SQL in *DB2 SQL Reference* for more information.

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 41. 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 42. 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 43. 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 44. Performing a positioned UPDATE with a named iterator

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

If you are using the DB2 Universal JDBC Driver, and your application connects to a DB2 UDB for z/OS® Version 8 server, or a DB2 UDB for Linux, UNIX®, and Windows® server at the FixPak 4 level or later, you 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 45.

```
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 45. Example of concurrent table operations using iterators with different SQL statements

Figure 46 on page 83 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 46. Example of concurrent table operations using iterators with the same SQL statement

Multiple open instances of an iterator in an SQLJ application

Multiple instances of an iterator can be open concurrently in a single SQLJ application. One application for this ability is to open several instances of an iterator that uses host expressions. Each instance can use a different set of host expression values.

The following example shows an application with two concurrently open instances of an iterator.

```
...
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 47. 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.

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 "Retrieving multiple result sets from a stored procedure in an SQLJ application" on page 95.

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 48. Calling a stored procedure in an SQLJ application

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());
}
```

For the JDBC/SQLJ driver for z/OS®, if your SQLJ or JDBC application runs only on DB2® UDB for OS/390® or z/OS, you can retrieve the contents of the SQLCA when an SQL statement generates an `SQLWarning` or `SQLException`. For information

on writing code to retrieve the SQLCA with the JDBC/SQLJ driver for z/OS, see "Handling an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 55.

With the DB2 Universal JDBC Driver, you can retrieve the SQLCA. For information on writing code to retrieve the SQLCA with the DB2 Universal JDBC Driver, see "Handling an SQLException under the DB2 Universal JDBC Driver" on page 22.

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 DB2 Universal JDBC Driver, follow the instructions in "Handling an SQLException under the DB2 Universal JDBC Driver" on page 22.

To retrieve DB2-specific information from the `SQLWarning` object with the JDBC/SQLJ driver for z/OS®, follow the instructions in "Handling an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 55.

Before you can execute `getWarnings` for an SQL clause, you need to set up an execution context for that SQL clause. See "Controlling the execution of SQL statements in SQLJ" on page 95 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);
```

Advanced SQLJ application programming concepts

The following topics contain more advanced information about writing SQLJ applications:

- "Retrieving multiple result sets from a stored procedure in an SQLJ application" on page 95
- "Making batch updates in SQLJ applications" on page 97
- "Iterators as passed variables for positioned UPDATE or DELETE operations in an SQLJ application" on page 100
- "Using scrollable iterators in an SQLJ application" on page 102

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 "Connecting to a data source using SQLJ" on page 66 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 49 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 49. Converting a JDBC result set to an SQLJ iterator

Notes to Figure 49:

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 50 on page 88 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 50. Converting an SQLJ iterator to a JDBC ResultSet

Notes to Figure 50:

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:

- Before you can access the columns of a remote table by name, through either a named iterator or an iterator that is converted to a JDBC `ResultSet` object, the DB2® DESCSTAT subsystem parameter must be set to YES. See "Setting DB2 subsystem parameters for SQLJ support" on page 280 for more information.
- 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 JDBC/SQLJ Driver for OS/390 and z/OS run-time property DB2CURSORHOLD or by the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver, 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.

LOBs in SQLJ applications with the DB2 Universal JDBC Driver

With the DB2 Universal JDBC Driver, 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`.

LOB locator support: The DB2 Universal JDBC Driver can use LOB locators to retrieve data. To cause JDBC to use LOB locators to retrieve data from LOB columns, you need to set the `fullyMaterializeLobData` property to `false`. Properties are discussed in "Properties for the DB2 Universal JDBC Driver" on page 185. `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, JDBC 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 DB2 subsystem. You cannot use a single LOB locator to move data between two different DB2 subsystems. To move LOB data between two DB2 subsystems, you need to materialize the LOB data when you retrieve it from a table in the first DB2 subsystem and then insert that data into the table in the second DB2 subsystem.

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

For Universal Driver type 2 connectivity to DB2® UDB for z/OS®, when the JDBC driver processes a CALL statement, the driver cannot determine the parameter data types.

When the deferPrepares property is set to true, and the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver, 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 DB2 Universal JDBC Driver, 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 our 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.

Using LOBs in SQLJ applications with the JDBC/SQLJ Driver for OS/390 and z/OS

With the JDBC/SQLJ Driver for OS/390 and z/OS, you cannot retrieve data into `Clob` or `Blob` host expressions. However, you can declare iterators with Clob or Blob data types to retrieve data from CLOB, BLOB, or DBCLOB columns, and retrieve the data into `String` host expressions. You can also use `String` host expressions to store data in CLOB, BLOB, or DBCLOB columns. The JDBC/SQLJ Driver for OS/390 and z/OS does not use LOB locators for processing data, so when you retrieve data from a LOB column you get the entire contents of the LOB.

Retrieving data from LOB columns: To retrieve data from a BLOB column, declare an iterator that includes a data type of Blob. To retrieve data from a CLOB or DBCLOB column, declare an iterator in which the corresponding column has a Clob data type. The following code fragment demonstrates how to retrieve data from a CLOB column.

```
#sql iterator ClobIter (int KEYCOL, Clob CLOBCOL);
                            // Declare named iterator
public static void main (String args[])
{
...
  ClobIter iter1 = null;    // Create iterator instance
  #sql [conn] iter1 = {SELECT KEYCOL, CLOBCOL from CLOBTABLE};
  while (iter1.next())
  {
    int key1 = iter1.KEYCOL();
                            // Retrieve KEYCOL value
    Clob clob1 = iter1.CLOBCOL();
                            // Retrieve CLOBCOL value
    String clobstring = clob1.getSubString((long)1,100);
                            // Use JDBC getSubString method
                            // to retrieve first 100 bytes of
                            // CLOBCOL value
    System.out.println("KEYCOL is: " + key1);
    System.out.println("First 100 chars of CLOBCOL is: " + clobstring );
  }
```

Figure 51. Retrieving CLOB data in an SQLJ program under the JDBC/SQLJ Driver for OS/390 and z/OS

Updating data in LOB columns: Under the JDBC/SQLJ Driver for OS/390 and z/OS, you cannot use host expressions with `Blob` or `Clob` data types to retrieve or update LOB data in SQLJ programs. Therefore, to update data in LOB columns, use String host expressions. The following code fragment demonstrates how to insert data into a CLOB column.

```
public static void main (String args[])
{
...
  int keycol = 45;
  String clobstr = new String("somereallybigstring");
                                // Declare object of class String
                                // and assign value that is to
                                // be passed to LOB column
  #sql [conn] {INSERT INTO CLOBTABLE
    (KEYCOL, CLOBCOL)           // Insert value from String
                                // host identifier into LOB column
                    VALUES(:keycol, :clobstr)};
}
```

Figure 52. Inserting data into a CLOB column

Using LOBs as stored procedure parameters: You cannot call a stored procedure that has LOB or LOB locator parameters.

ROWIDs in SQLJ with the DB2 Universal JDBC Driver

DB2® UDB 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 DB2 Universal JDBC Driver 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 53 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 53. Example of using an iterator to retrieve ROWID values

Figure 54 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 54. Example of calling a stored procedure with a ROWID parameter

Using graphic string constants in SQLJ applications

In EBCDIC environments, graphic string constants in SQLJ applications have the following form:

G'\u*xxxx*\u*xxxx*...\u*xxxx*'

xxxx is the Unicode value in hexadecimal that corresponds to the desired EBCDIC graphic character.

For example, an EBCDIC double-byte G has the hexadecimal value 42C7. The corresponding Unicode hexadecimal value is FF27. Therefore, in a SQLJ executable statement, you represent the graphic string constant for an EBCDIC double-byte G as:

```
G'\uFF27'
```

The following executable statement demonstrates a searched UPDATE that includes graphic string constants:

```
// GRAPHIC_TABLE has one VARGRAPHIC(10) column named VGCOL.
// At least one row contains the string "GRAPHIC" in double-byte
// EBCDIC characters. The Unicode equivalent of "GRAPHIC" is
// G'\uFF27\uFF32\uFF21\uFF30\uFF28\uFF29\uFF23'.
// Update "GRAPHIC" in all rows to "graphic" in double-byte
// EBCDIC characters. The Unicode equivalent of "graphic" is
// G'\uFF47\uFF52\uFF41\uFF50\uFF48\uFF49\uFF43'
#sql [myConnCtx] {UPDATE GRAPHIC_TABLE
  SET VGCOL=G'\uFF47\uFF52\uFF41\uFF50\uFF48\uFF49\uFF43'
  WHERE VGCOL=G'\uFF27\uFF32\uFF21\uFF30\uFF28\uFF29\uFF23'};
```

Figure 55. Using graphic string constants in an SQLJ application

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 56. Defining and using a distinct type

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");
   ```

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 second form of `getNextResultSet` requires JDK 1.4 or later.

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 57. Retrieving result sets from a stored procedure

Making batch updates in SQLJ applications

The DB2 Universal JDBC Driver 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 58. 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 "Making batch updates in JDBC applications" on page 41 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.

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. (See Chapter 6, "Preparing and running JDBC and SQLJ programs," on page 219 for more information on customization.) 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 59. 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 59 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 60 on page 102 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 60. 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.

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 DB2 Universal JDBC Driver 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`.

Important: Like static scrollable cursors in any other language, SQLJ static scrollable iterators use declared temporary tables for their internal processing. This

means that before you can execute any applications that contain static scrollable iterators, your database administrator needs to create a temporary database and temporary table spaces for those declared temporary tables. See *DB2 Installation Guide* for detailed information on creating the temporary database and temporary table spaces.

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 UDB 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 UDB for z/OS servers, at Version 8 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);
   ```
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 7 on page 104.

Table 7. 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
`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 8 lists the clauses that you can use to position the cursor.

Table 8. 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.

Table 8. FETCH clauses for positioning a scrollable cursor (continued)

Method	Positions the cursor
AFTER[4]	After the last row in the result table
BEFORE[4]	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.
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 61. Using scrollable iterators

Chapter 4. JDBC and SQLJ reference

The following topics contain reference information about JDBC and SQLJ:
- "Comparison of driver support for JDBC APIs"
- "Java, JDBC, and SQL data types" on page 127
- "Comparison of driver support for JDBC APIs"
- "SQLJ syntax" on page 132
- "sqlj.runtime reference" on page 142
- "DB2 Universal JDBC Driver reference information" on page 165
- "DataSource properties for the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS" on page 196

Comparison of 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 9. JDBC drivers for DB2 UDB

JDBC driver name	Associated DB2 UDB
DB2 Universal JDBC Driver	DB2 UDB for Linux, UNIX and Windows or DB2 UDB for z/OS
JDBC/SQLJ 2.0 Driver for OS/390	DB2 UDB for z/OS
DB2 JDBC Type 2 Driver for Linux, UNIX and Windows	DB2 UDB for Linux, UNIX and Windows

Table 10. DB2 JDBC support for Array methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getArray	No	No	No
getBaseType	No	No	No
getBaseTypeName	No	No	No
getResultSet	No	No	No

Table 11. DB2 JDBC support for BatchUpdateException methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getUpdateCounts	Yes	Yes	Yes

Table 12. DB2 JDBC support for Blob methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getBinaryStream	Yes	Yes	Yes

Table 12. DB2 JDBC support for Blob methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getBytes	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setBinaryStream[1,2]	Yes	No	No
setBytes[1,2]	Yes	No	No
truncate[1,2]	Yes	No	No

Notes:

1. This is a JDBC 3.0 method.
2. This method can be used only if the `fullyMaterializeLobData` property is set to true.

Table 13. DB2 JDBC support for CallableStatement methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
Methods inherited from java.sql.PreparedStatement	Yes	Yes	Yes
getArray	No	No	No
getBigDecimal	Yes	Yes	Yes
getBlob	Yes	Yes	Yes
getBoolean	Yes	Yes	Yes
getByte	Yes	Yes	Yes
getBytes	Yes	Yes	Yes
getClob	Yes	Yes	Yes
getDate	Yes[1]	Yes[2]	Yes[1]
getDouble	Yes	Yes	Yes
getFloat	Yes	Yes	Yes
getInt	Yes	Yes	Yes
getLong	Yes	Yes	Yes
getObject	Yes[3]	Yes[3]	Yes[3]
getRef	No	No	No
getShort	Yes	Yes	Yes
getString	Yes	Yes	Yes
getTime	Yes[1]	Yes[4]	Yes[1]
getTimestamp	Yes[1]	Yes[5]	Yes[1]
registerOutParameter[6]	Yes	Yes	Yes
wasNull	Yes	Yes	Yes

Table 13. DB2 JDBC support for CallableStatement methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support

Notes:

1. 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.
2. The following forms of `getDate` are *not* supported:

   ```
   getDate(int columnIndex, java.util.Calendar cal)
   getDate(String columnName, java.util.Calendar cal)
   ```

3. The following form of the `getObject` method is *not* supported:

   ```
   getObject(int parameterIndex, java.util.Map  map)
   ```

4. The following forms of `getTime` are *not* supported:

   ```
   getTime(int columnIndex, java.util.Calendar cal)
   getTime(String columnName, java.util.Calendar cal)
   ```

5. The following forms of `getTimestamp` are *not* supported:

   ```
   getTimestamp(int columnIndex, java.util.Calendar cal)
   getTimestamp(String columnName, java.util.Calendar cal)
   ```

6. The following form of the `registerOutParameter` method is *not* supported:

   ```
   registerOutParameter(int parameterIndex, int jdbcType, String typeName)
   ```

Table 14. DB2 JDBC support for Clob methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getAsciiStream	Yes	Yes	Yes
getCharacterStream	Yes	Yes	Yes
getSubString	Yes	Yes	Yes
length	Yes	Yes	Yes
position	Yes	Yes	Yes
setAsciiStream[1,2]	Yes	No	No
setCharacterStream[1,2]	Yes	No	No
setString[1,2]	Yes	No	No
truncate[1,2]	Yes	No	No

Notes:

1. This is a JDBC 3.0 method.
2. This method can be used only if the `fullyMaterializeLobData` property is set to true.

Table 15. DB2 JDBC support for Connection methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
commit	Yes	Yes	Yes

Table 15. DB2 JDBC support for Connection methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
createStatement	Yes[1]	Yes [2]	Yes
getAutoCommit	Yes	Yes	Yes
getCatalog	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getTransactionIsolation	Yes	Yes	Yes
getTypeMap	No	No	No
getWarnings	Yes	Yes	Yes
isClosed	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
nativeSQL	Yes	Yes	Yes
prepareCall	Yes	Yes[3]	Yes
prepareStatement	Yes[4]	Yes	Yes
releaseSavepoint	Yes[5]	No	No
rollback	Yes	Yes[6]	Yes[6]
setAutoCommit	Yes	Yes	Yes
setCatalog	Yes	Yes	Yes
setReadOnly	Yes[7]	Yes[7]	Yes
setSavepoint	Yes[5]	No	No
setTransactionIsolation	Yes	Yes	Yes
setTypeMap	No	No	No

Notes:

1. In addition to the JDBC 2.0 forms of `createStatement` statement, the following JDBC 3.0 form of `createStatement` is supported:

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

2. For the following form of `createStatement`, a *resultSetType* value of `TYPE_FORWARD_ONLY` and a *resultSetConcurrency* value of `CONCUR_READ_ONLY` are supported:

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

3. The following form of `prepareCall` is *not* supported:

   ```
   prepareCall(String sql, int resultSetType, int resultSetConcurrency)
   ```

4. In addition to the other forms of `prepareStatement`, the DB2 Universal JDBC Driver supports the following JDBC 3.0 form:

   ```
   prepareStatement(String sql, int autoGeneratedKeys)
   ```

5. This is a JDBC 3.0 method.
6. The JDBC 3.0 `rollback(Savepoint savepoint)` method is not supported.
7. The driver does not use the setting. For the DB2 Universal JDBC Driver, a connection can be set as read-only through the `readOnly` property for a `Connection` or `DataSource` object.

Table 16. DB2 JDBC support for ConnectionEvent methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.util.EventObject	Yes	Yes	Yes
getSQLException	Yes	Yes	Yes

Table 17. DB2 JDBC support for ConnectionEventListener methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
connectionClosed	Yes	Yes	Yes
connectionErrorOccurred	Yes	Yes	Yes

Table 18. DB2 JDBC support for ConnectionPoolDataSource methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getLoginTimeout	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
getPooledConnection	Yes	Yes	Yes
setLoginTimeout	Yes[1]	Yes	Yes
setLogWriter	Yes	Yes	Yes

Note:

1. This method is not supported for Universal Driver type 2 connectivity on DB2 UDB in the OS/390 or z/OS environment.

Table 19. DB2 JDBC support for DatabaseMetaData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
allProceduresAreCallable	Yes	Yes	Yes
allTablesAreSelectable	Yes	Yes	Yes
dataDefinitionCausesTransactionCommit	Yes	Yes	Yes
dataDefinitionIgnoredInTransactions	Yes	Yes	Yes
deletesAreDetected	Yes	Yes	Yes
doesMaxRowSizeIncludeBlobs	Yes	Yes	Yes
getAttributes	Yes	No	No
getBestRowIdentifier	Yes	Yes	Yes
getCatalogs	Yes	Yes	Yes
getCatalogSeparator	Yes	Yes	Yes
getCatalogTerm	Yes	Yes	Yes
getColumnPrivileges	Yes	Yes	Yes

Table 19. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getColumns	Yes[1]	Yes	Yes
getConnection	Yes	Yes	Yes
getCrossReference	Yes	Yes	Yes
getDatabaseMajorVersion	Yes	No	No
getDatabaseMinorVersion	Yes	No	No
getDatabaseProductName	Yes	Yes	Yes
getDatabaseProductVersion	Yes	Yes	Yes
getDefaultTransactionIsolation	Yes	Yes	Yes
getDriverMajorVersion	Yes	Yes	Yes
getDriverMinorVersion	Yes	Yes	Yes
getDriverName	Yes	Yes	Yes
getDriverVersion	Yes	Yes	Yes
getExportedKeys	Yes	Yes	Yes
getExtraNameCharacters	Yes	Yes	Yes
getIdentifierQuoteString	Yes	Yes	Yes
getImportedKeys	Yes	Yes	Yes
getIndexInfo	Yes	Yes	Yes
getJDBCMajorVersion	Yes	No	No
getJDBCMinorVersion	Yes	No	No
getMaxBinaryLiteralLength	Yes	Yes	Yes
getMaxCatalogNameLength	Yes	Yes	Yes
getMaxCharLiteralLength	Yes	Yes	Yes
getMaxColumnNameLength	Yes	Yes	Yes
getMaxColumnsInGroupBy	Yes	Yes	Yes
getMaxColumnsInIndex	Yes	Yes	Yes
getMaxColumnsInOrderBy	Yes	Yes	Yes
getMaxColumnsInSelect	Yes	Yes	Yes
getMaxColumnsInTable	Yes	Yes	Yes
getMaxConnections	Yes	Yes	Yes
getMaxCursorNameLength	Yes	Yes	Yes
getMaxIndexLength	Yes	Yes	Yes
getMaxProcedureNameLength	Yes	Yes	Yes
getMaxRowSize	Yes	Yes	Yes
getMaxSchemaNameLength	Yes	Yes	Yes
getMaxStatementLength	Yes	Yes	Yes
getMaxStatements	Yes	Yes	Yes

Table 19. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getMaxTableNameLength	Yes	Yes	Yes
getMaxTablesInSelect	Yes	Yes	Yes
getMaxUserNameLength	Yes	Yes	Yes
getNumericFunctions	Yes	Yes	Yes
getPrimaryKeys	Yes	Yes	Yes
getProcedureColumns	Yes	Yes	Yes
getProcedures	Yes	Yes	Yes
getProcedureTerm	Yes	Yes	Yes
getResultSetHoldability	Yes	No	No
getSchemas	Yes[1]	Yes	Yes
getSchemaTerm	Yes	Yes	Yes
getSearchStringEscape	Yes	Yes	Yes
getSQLKeywords	Yes	Yes	Yes
getSQLStateType	Yes	No	No
getStringFunctions	Yes	Yes	Yes
getSuperTables	Yes[2]	No	No
getSuperTypes	Yes[2]	No	No
getSystemFunctions	Yes	Yes	Yes
getTablePrivileges	Yes	Yes	Yes
getTables	Yes[1]	Yes	Yes
getTableTypes	Yes	Yes	Yes
getTimeDateFunctions	Yes	Yes	Yes
getTypeInfo	Yes	Yes	Yes
getUDTs	No	No	Yes[2]
getURL	Yes	Yes	Yes
getUserName	Yes	Yes	Yes
getVersionColumns	Yes	Yes	Yes
insertsAreDetected	Yes	Yes	Yes
isCatalogAtStart	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
nullPlusNonNullIsNull	Yes	Yes	Yes
nullsAreSortedAtEnd	Yes	Yes	Yes
nullsAreSortedAtStart	Yes	Yes	Yes
nullsAreSortedHigh	Yes	Yes	Yes
nullsAreSortedLow	Yes	Yes	Yes
othersDeletesAreVisible	Yes	Yes	Yes

Table 19. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
othersInsertsAreVisible	Yes	Yes	Yes
othersUpdatesAreVisible	Yes	Yes	Yes
ownDeletesAreVisible	Yes	Yes	Yes
ownInsertsAreVisible	Yes	Yes	Yes
ownUpdatesAreVisible	Yes	Yes	Yes
storesLowerCaseIdentifiers	Yes	Yes	Yes
storesLowerCaseQuotedIdentifiers	Yes	Yes	Yes
storesMixedCaseIdentifiers	Yes	Yes	Yes
storesMixedCaseQuotedIdentifiers	Yes	Yes	Yes
storesUpperCaseIdentifiers	Yes	Yes	Yes
storesUpperCaseQuotedIdentifiers	Yes	Yes	Yes
supportsAlterTableWithAddColumn	Yes	Yes	Yes
supportsAlterTableWithDropColumn	Yes	Yes	Yes
supportsANSI92EntryLevelSQL	Yes	Yes	Yes
supportsANSI92FullSQL	Yes	Yes	Yes
supportsANSI92IntermediateSQL	Yes	Yes	Yes
supportsBatchUpdates	Yes	Yes	Yes
supportsCatalogsInDataManipulation	Yes	Yes	Yes
supportsCatalogsInIndexDefinitions	Yes	Yes	Yes
supportsCatalogsInPrivilegeDefinitions	Yes	Yes	Yes
supportsCatalogsInProcedureCalls	Yes	Yes	Yes
supportsCatalogsInTableDefinitions	Yes	Yes	Yes
SupportsColumnAliasing	Yes	Yes	Yes
supportsConvert	Yes	Yes	Yes
supportsCoreSQLGrammar	Yes	Yes	Yes
supportsCorrelatedSubqueries	Yes	Yes	Yes
supportsDataDefinitionAndDataManipulationTransactions	Yes	Yes	Yes
supportsDataManipulationTransactionsOnly	Yes	Yes	Yes
supportsDifferentTableCorrelationNames	Yes	Yes	Yes
supportsExpressionsInOrderBy	Yes	Yes	Yes
supportsExtendedSQLGrammar	Yes	Yes	Yes
supportsFullOuterJoins	Yes	Yes	Yes
supportsGetGeneratedKeys	Yes	No	No
supportsGroupBy	Yes	Yes	Yes
supportsGroupByBeyondSelect	Yes	Yes	Yes
supportsGroupByUnrelated	Yes	Yes	Yes

Table 19. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
supportsIntegrityEnhancementFacility	Yes	Yes	Yes
supportsLikeEscapeClause	Yes	Yes	Yes
supportsLimitedOuterJoins	Yes	Yes	Yes
supportsMinimumSQLGrammar	Yes	Yes	Yes
supportsMixedCaseIdentifiers	Yes	Yes	Yes
supportsMixedCaseQuotedIdentifiers	Yes	Yes	Yes
supportsMultipleOpenResults	Yes	Yes	No
supportsMultipleResultSets	Yes	Yes	Yes
supportsMultipleTransactions	Yes	Yes	Yes
supportsNamedParameters	Yes	No	No
supportsNonNullableColumns	Yes	Yes	Yes
supportsOpenCursorsAcross Commit	Yes	Yes	Yes
supportsOpenCursorsAcross Rollback	Yes	Yes	Yes
supportsOpenStatementsAcrossCommit	Yes	Yes	Yes
supportsOpenStatementsAcrossRollback	Yes	Yes	Yes
supportsOrderByUnrelated	Yes	Yes	Yes
supportsOuterJoins	Yes	Yes	Yes
supportsPositionedDelete	Yes	Yes	Yes
supportsPositionedUpdate	Yes	Yes	Yes
supportsResultSetConcurrency	Yes	Yes	Yes
supportsResultSetHoldability	Yes	No	No
supportsResultSetType	Yes	Yes	Yes
supportsSavepoints	Yes	No	No
supportsSchemasInDataManipulation	Yes	Yes	Yes
supportsSchemasInIndexDefinitions	Yes	Yes	Yes
supportsSchemasInPrivilegeDefinitions	Yes	Yes	Yes
supportsSchemasInProcedureCalls	Yes	Yes	Yes
supportsSchemasInTableDefinitions	Yes	Yes	Yes
supportsSelectForUpdate	Yes	Yes	Yes
supportsStoredProcedures	Yes	Yes	Yes
supportsSubqueriesInComparisons	Yes	Yes	Yes
supportsSubqueriesInExists	Yes	Yes	Yes
supportsSubqueriesInIns	Yes	Yes	Yes
supportsSubqueriesInQuantifieds	Yes	Yes	Yes
supportsSuperTables	Yes	No	No
supportsSuperTypes	Yes	No	No

Table 19. DB2 JDBC support for DatabaseMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
supportsTableCorrelationNames	Yes	Yes	Yes
supportsTransactionIsolationLevel	Yes	Yes	Yes
supportsTransactions	Yes	Yes	Yes
supportsUnion	Yes	Yes	Yes
supportsUnionAll	Yes	Yes	Yes
updatesAreDetected	Yes	Yes	Yes
usesLocalFilePerTable	Yes	Yes	Yes
usesLocalFiles	Yes	Yes	Yes

Notes:

1. The JDBC 3.0 version of this method is supported.
2. The method can be executed, but it returns an empty `ResultSet`.

Table 20. DB2 JDBC support for DataSource methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getConnection	Yes	Yes	Yes
getLoginTimeout	Yes	Yes	Yes[1]
getLogWriter	Yes	Yes	Yes
setLoginTimeout	Yes[2]	Yes	Yes[1]
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for Universal Driver type 2 connectivity on DB2 UDB in the OS/390 or z/OS environment.

Table 21. DB2 JDBC support for DataTruncation methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Throwable	Yes	Yes	Yes
Methods inherited from java.sql.SQLException	Yes	Yes	Yes
Methods inherited from java.sql.SQLWarning	Yes	Yes	Yes
getDataSize	Yes	Yes	Yes
getIndex	Yes	Yes	Yes
getParameter	Yes	Yes	Yes
getRead	Yes	Yes	Yes

Table 21. DB2 JDBC support for DataTruncation methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getTransferSize	Yes	Yes	Yes

Table 22. DB2 JDBC support for Driver methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
acceptsURL	Yes	Yes	Yes
connect	Yes	Yes	Yes
getMajorVersion	Yes	Yes	Yes
getMinorVersion	Yes	Yes	Yes
getPropertyInfo	Yes	Yes	Yes
jdbcCompliant	Yes	Yes	Yes

Table 23. DB2 JDBC support for DriverManager methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
deregisterDriver	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
getDriver	Yes	Yes	Yes
getDrivers	Yes	Yes	Yes
getLoginTimeout	Yes	Yes	Yes[1]
getLogStream	Yes	Yes	Yes
getLogWriter	Yes	Yes	Yes
println	Yes	Yes	Yes
registerDriver	Yes	Yes	Yes
setLoginTimeout	Yes[2]	Yes	Yes[1]
setLogStream	Yes	Yes	Yes
setLogWriter	Yes	Yes	Yes

Notes:

1. The DB2 JDBC Type 2 Driver does not use this setting.
2. This method is not supported for Universal Driver type 2 connectivity on DB2 UDB in the OS/390 or z/OS environment.

Table 24. DB2 JDBC support for ParameterMetaData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getParameterClassName	No	No	No
getParameterCount	Yes	No	No
getParameterMode	Yes	No	No

Table 24. DB2 JDBC support for ParameterMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getParameterType	Yes	No	No
getParameterTypeName	Yes	No	No
getPrecision	Yes	No	No
getScale	Yes	No	No
isNullable	Yes	No	No
isSigned	Yes	No	No

Table 25. DB2 JDBC support for PooledConnection methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
addConnectionEventListener	Yes	Yes	Yes
close	Yes	Yes	Yes
getConnection	Yes	Yes	Yes
removeConnectionEventListener	Yes	Yes	Yes

Table 26. DB2 JDBC support for PreparedStatement methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.sql.Statement	Yes	Yes	Yes
addBatch	Yes	Yes	Yes
clearParameters	Yes	Yes	Yes
execute	Yes	Yes	Yes
executeQuery	Yes	Yes	Yes
executeUpdate	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
setArray	No	No	No
setAsciiStream	Yes	Yes	Yes
setBigDecimal	Yes	Yes	Yes
setBinaryStream	Yes	Yes	Yes
setBlob	Yes	Yes	Yes
setBoolean	Yes	Yes	Yes
setByte	Yes	Yes	Yes
setBytes	Yes	Yes	Yes
setCharacterStream	Yes	Yes	Yes
setClob	Yes	Yes	Yes
setDate	Yes[1]	Yes[2]	Yes[1]
setDouble	Yes	Yes	Yes

Table 26. DB2 JDBC support for PreparedStatement methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
setFloat	Yes	Yes	Yes
setInt	Yes	Yes	Yes
setLong	Yes	Yes	Yes
setNull	Yes[3]	Yes[3]	Yes[3]
setObject	Yes	Yes	Yes
setRef	No	No	No
setShort	Yes	Yes	Yes
setString	Yes[4]	Yes[4]	Yes[4]
setTime	Yes[1]	Yes[5]	Yes[1]
setTimestamp	Yes[1]	Yes[6]	Yes[1]
setUnicodeStream	Yes	Yes	Yes
setURL	Yes	No	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 `setDate` is *not* supported:
   `setDate(int parameterIndex, java.sql.Date x, java.util.Calendar cal)`
3. The following form of `setNull` is *not* supported:
   `setNull(int parameterIndex, int jdbcType, String typeName)`
4. setString is not supported if the column has the FOR BIT DATA attribute or the data type is BLOB.
5. The following form of `setTime` is *not* supported:
   `setTime(int parameterIndex, java.sql.Time x, java.util.Calendar cal)`
6. The following form of `setTimestamp` is *not* supported:
   `setTimestamp(int parameterIndex, java.sql.Timestamp x, java.util.Calendar cal)`

Table 27. DB2 JDBC support for Ref methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
get BaseTypeName	No	No	No

Table 28. DB2 JDBC support for ResultSet methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
absolute	Yes	No	Yes
afterLast	Yes	No	Yes
beforeFirst	Yes	No	Yes
cancelRowUpdates	Yes	No	No

Table 28. DB2 JDBC support for ResultSet methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
deleteRow	Yes	No	No
findColumn	Yes	Yes	Yes
first	Yes	No	Yes
getArray	No	No	No
getAsciiStream	Yes	Yes	Yes
getBigDecimal	Yes	Yes	Yes
getBinaryStream	Yes[1]	Yes[1]	Yes
getBlob	Yes	Yes	Yes
getBoolean	Yes	Yes	Yes
getByte	Yes	Yes	Yes
getBytes	Yes	Yes	Yes
getCharacterStream	Yes	Yes	Yes
getClob	Yes	Yes	Yes
getConcurrency	Yes	Yes	Yes
getCursorName	Yes	Yes	Yes
getDate	Yes[2]	Yes[3]	Yes[2]
getDouble	Yes	Yes	Yes
getFetchDirection	Yes	Yes	Yes
getFetchSize	Yes	Yes	Yes
getFloat	Yes	Yes	Yes
getInt	Yes	Yes	Yes
getLong	Yes	Yes	Yes
getMetaData	Yes	Yes	Yes
getObject	Yes[4]	Yes[4]	Yes[4]
getRef	No	No	No
getRow	Yes	No	Yes
getShort	Yes	Yes	Yes
getStatement	Yes	Yes	Yes
getString	Yes	Yes	Yes
getTime	Yes[2]	Yes[5]	Yes[2]
getTimestamp	Yes[2]	Yes[6]	Yes[2]
getType	Yes	Yes	Yes
getUnicodeStream	Yes	Yes	Yes
getURL	Yes	No	Yes
getWarnings	Yes	Yes	Yes
insertRow	No	No	No

Table 28. DB2 JDBC support for ResultSet methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
isAfterLast	Yes	No	Yes
isBeforeFirst	Yes	No	Yes
isFirst	Yes	No	Yes
isLast	Yes	No	Yes
last	Yes	No	Yes
moveToCurrentRow	Yes	No	No
moveToInsertRow	No	No	No
next	Yes	Yes	Yes
previous	Yes	No	Yes
refreshRow	Yes	No	No
relative	Yes	No	Yes
rowDeleted	Yes	No	No
rowInserted	No	No	No
rowUpdated	Yes	No	No
setFetchDirection	Yes	Yes[7]	Yes
setFetchSize	Yes	Yes	Yes
updateAsciiStream	Yes	No	No
updateBigDecimal	Yes	No	No
updateBinaryStream	Yes	No	No
updateBoolean	Yes	No	No
updateByte	Yes	No	No
updateBytes	Yes	No	No
updateCharacterStream	Yes	No	No
updateDate	Yes	No	No
updateDouble	Yes	No	No
updateFloat	Yes	No	No
updateInt	Yes	No	No
updateLong	Yes	No	No
updateNull	Yes	No	No
updateObject	Yes	No	No
updateRow	Yes	No	No
updateShort	Yes	No	No
updateString	Yes	No	No
updateTime	Yes	No	No
updateTimestamp	Yes	No	No
wasNull	Yes	Yes	Yes

Table 28. DB2 JDBC support for ResultSet methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support

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 forms of `getDate` are *not* supported:

   ```
   getDate(int columnIndex, java.util.Calendar cal)
   getDate(String columnName, java.util.Calendar cal)
   ```

4. The following form of the `getObject` method is *not* supported:

   ```
   getObject(int parameterIndex, java.util.Map  map)
   ```

5. The following forms of `getTime` are *not* supported:

   ```
   getTime(int columnIndex, java.util.Calendar cal)
   getTime(String columnName, java.util.Calendar cal)
   ```

6. The following forms of `getTimestamp` are *not* supported:

   ```
   getTimestamp(int columnIndex, java.util.Calendar cal)
   getTimestamp(String columnName, java.util.Calendar cal)
   ```

7. Supported only if *direction* is `ResultSet.FETCH_FORWARD`.

Table 29. DB2 JDBC support for ResultSetMetaData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getCatalogName	Yes	Yes	Yes
getColumnClassName	No	No	Yes
getColumnCount	Yes	Yes	Yes
getColumnDisplaySize	Yes	Yes	Yes
getColumnLabel	Yes	Yes	Yes
getColumnName	Yes	Yes	Yes
getColumnType	Yes	Yes	Yes
getColumnTypeName	Yes	Yes	Yes
getPrecision	Yes	Yes	Yes
getScale	Yes	Yes	Yes
getSchemaName	Yes	Yes	Yes
getTableName	Yes	Yes	Yes
isAutoIncrement	Yes	Yes	Yes
isCaseSensitive	Yes	Yes	Yes
isCurrency	Yes	Yes	Yes
isDefinitelyWritable	Yes	Yes	Yes
isNullable	Yes	Yes	Yes
isReadOnly	Yes	Yes	Yes
isSearchable	Yes	Yes	Yes
isSigned	Yes	Yes	Yes

Table 29. DB2 JDBC support for ResultSetMetaData methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
isWritable	Yes	Yes	Yes

Table 30. DB2 JDBC support for SQLData methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getSQLTypeName	No	No	No
readSQL	No	No	No
writeSQL	No	No	No

Table 31. DB2 JDBC support for SQLException methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from java.lang.Exception	Yes	Yes	Yes
getSQLState	Yes	Yes	Yes
getErrorCode	Yes	Yes	Yes
getNextException	Yes	Yes	Yes
setNextException	Yes	Yes	Yes

Table 32. DB2 JDBC support for SQLInput methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
readArray	No	No	No
readAsciiStream	No	No	No
readBigDecimal	No	No	No
readBinaryStream	No	No	No
readBlob	No	No	No
readBoolean	No	No	No
readByte	No	No	No
readBytes	No	No	No
readCharacterStream	No	No	No
readClob	No	No	No
readDate	No	No	No
readDouble	No	No	No
readFloat	No	No	No
readInt	No	No	No
readLong	No	No	No
readObject	No	No	No

Table 32. DB2 JDBC support for SQLInput methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
readRef	No	No	No
readShort	No	No	No
readString	No	No	No
readTime	No	No	No
readTimestamp	No	No	No
wasNull	No	No	No

Table 33. DB2 JDBC support for SQLOutput methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
writeArray	No	No	No
writeAsciiStream	No	No	No
writeBigDecimal	No	No	No
writeBinaryStream	No	No	No
writeBlob	No	No	No
writeBoolean	No	No	No
writeByte	No	No	No
writeBytes	No	No	No
writeCharacterStream	No	No	No
writeClob	No	No	No
writeDate	No	No	No
writeDouble	No	No	No
writeFloat	No	No	No
writeInt	No	No	No
writeLong	No	No	No
writeObject	No	No	No
writeRef	No	No	No
writeShort	No	No	No
writeString	No	No	No
writeStruct	No	No	No
writeTime	No	No	No
writeTimestamp	No	No	No

Table 34. DB2 JDBC support for Statement methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
addBatch	Yes	Yes	Yes
cancel	Yes[1,2]	No	Yes

Table 34. DB2 JDBC support for Statement methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
clearBatch	Yes	Yes	Yes
clearWarnings	Yes	Yes	Yes
close	Yes	Yes	Yes
execute	Yes[3]	Yes	Yes
executeBatch	Yes	Yes	Yes
executeQuery	Yes	Yes	Yes
executeUpdate	Yes[3]	Yes	Yes
getConnection	Yes	No	Yes
getFetchDirection	Yes	No	Yes
getFetchSize	Yes	No	Yes
getGeneratedKeys	Yes	No	No
getMaxFieldSize	Yes	Yes	Yes
getMaxRows	Yes	Yes	Yes
getMoreResults	Yes[4]	Yes	Yes
getQueryTimeout	Yes[2]	Yes	Yes
getResultSet	Yes	Yes	Yes
getResultSetConcurrency	Yes	Yes	Yes
getResultSetType	Yes	Yes	Yes
getUpdateCount[5]	Yes	Yes	Yes
getWarnings	Yes	Yes	Yes
setCursorName	Yes	Yes	Yes
setEscapeProcessing	Yes	Yes	Yes
setFetchDirection	Yes	Yes	Yes
setFetchSize	Yes	No	Yes
setMaxFieldSize	Yes	Yes	Yes
setMaxRows	Yes	Yes	Yes
setQueryTimeout	Yes[6]	Yes[6]	Yes

Table 34. DB2 JDBC support for Statement methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support

Notes:

1. With Universal Driver type 4 connectivity, you can execute Statement.cancel() only if the database server supports the DRDA INTRDBRQS (interrupt relational database request) command. Only DB2 UDB for z/OS servers at the Version 8 or later level have this support. Therefore, with Universal Driver type 4 connectivity, you can execute Statement.cancel() only for connections to DB2 UDB for z/OS at Version 8 or later.
2. This method is supported only for:
   - Universal Driver type 2 connectivity to DB2 UDB Linux, UNIX, and Windows server at Version 8.1 or later
   - Universal Driver type 4 connectivity to DB2 UDB for z/OS Version 8 or later
3. In addition to the other forms of `execute` or `executeUpdate`, the DB2 Universal JDBC Driver supports the following JDBC 3.0 forms:

   ```
   executeUpdate(String sql, int autoGeneratedKeys)
   execute(String sql, int autoGeneratedKeys)
   ```
4. In addition to `getMoreResults()`, the DB2 Universal JDBC Driver supports the following JDBC 3.0 forms:
   - `getMoreResults(java.sql.Statement.CLOSE_CURRENT_RESULT)`
   - `getMoreResults(java.sql.Statement.KEEP_CURRENT_RESULT)`
   - `getMoreResults(java.sql.Statement.CLOSE_ALL_RESULTS)`
5. Not supported for stored procedure `ResultSets`.
6. For Universal Driver type 2 connectivity in the OS/390 or z/OS environment, this method is supported only for a *seconds* value of 0.

Table 35. DB2 JDBC support for Struct methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getSQLTypeName	No	No	No
getAttributes	No	No	No

Table 36. DB2 JDBC support for XAConnection methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
Methods inherited from javax.sql.PooledConnection	Yes[1]	No	Yes
getXAResource	Yes[1]	No	Yes

Notes:

1. This method is supported for DB2 Universal JDBC Driver type 2 connectivity to a DB2 UDB for Linux, UNIX and Windows server or DB2 Universal JDBC Driver type 4 connectivity to a DB2 UDB for z/OS server.

Table 37. DB2 JDBC support for XADataSource methods

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
getLoginTimeout	Yes	No	Yes
getLogWriter	Yes	No	Yes
getXAConnection	Yes	No	Yes
setLoginTimeout	Yes	No	Yes

Table 37. DB2 JDBC support for XADataSource methods (continued)

JDBC method	DB2 Universal JDBC Driver support	JDBC/SQLJ 2.0 Driver for OS/390 support	DB2 JDBC Type 2 Driver for Linux, UNIX and Windows support
setLogWriter	Yes	No	Yes

Java, JDBC, and SQL data types

The following tables summarize the mappings of Java data types to JDBC and SQL data types for a DB2 UDB for OS/390 or z/OS system.

Table 38 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 38. 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	DECIMAL(19,0)[2]
long, java.lang.Long	BIGINT[3]
float, java.lang.Float	REAL
double, java.lang.Double	DOUBLE
java.math.BigDecimal	DECIMAL(*p*,*s*)[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]
byte[]	CHAR(*n*) FOR BIT DATA[5]
byte[]	VARCHAR(*n*) FOR BIT DATA[7]
byte[]	BLOB(*n*)[9,10]
byte[]	ROWID
java.sql.Blob	BLOB(*n*)[10]
java.sql.Clob	CLOB(*n*)[10]
java.sql.Clob	DBCLOB(*m*)[11]
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]
com.ibm.db2.jcc.DB2RowID	ROWID
java.net.URL	DATALINK[12]

Table 38. Mappings of Java data types to DB2 data types for updating DB2 tables (continued)

Java data type	SQL data type

Notes:

1. DB2 has no exact equivalent for the Java boolean or byte data types, but the best fit is SMALLINT.
2. DB2 UDB in the OS/390 or z/OS environment has no exact equivalent for the Java long or java.lang.Long data types, but the best fit is DECIMAL(19,0).
3. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows.
4. *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 precision and performance than using other combinations of data types.
5. *n*<=255.
6. *m*<=127.
7. *n*<=32704.
8. *m*<=16352.
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 39 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 39. 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
INTEGER	int	short, byte, long, float, double, java.math.BigDecimal, boolean, java.lang.String
BIGINT[1]	long	int, short, byte, float, double, java.math.BigDecimal, boolean, java.lang.String
DECIMAL(*p*,*s*) or NUMERIC(*p*,*s*)[2]	java.math.BigDecimal	long, int, short, byte, float, double, 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

Table 39. 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
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
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[][3]
DBCLOB(*m*)	No exact equivalent. Use java.sql.Clob.	
ROWID	com.ibm.db2.jcc.DB2RowID	byte[]
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[4]	

Notes:

1. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows.
2. You should design financial applications so that DECIMAL columns map to java.math.BigDecimal columns. If you know the precision and scale of a DECIMAL column, retrieving data from that column into a java.math.BigDecimal variable results in better precision and performance than using other combinations of data types.
3. This mapping is valid only if DB2 can determine the data type of the column.
4. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 40 on page 130 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 40 on page 130, the first data type is the **recommended** data type.

Table 40. 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[2]
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*)
java.lang.String	CLOB	CLOB(*n*)
byte[]	BINARY	CHAR FOR BIT DATA
byte[]	VARBINARY	VARCHAR FOR BIT DATA
byte[]	LONGVARBINARY	VARCHAR FOR BIT DATA
byte[]	VARBINARY	BLOB(*n*)[3]
byte[]	LONGVARBINARY	BLOB(*n*)[3]
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
java.net.URL	DATALINK	DATALINK[4]

Table 40. 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

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. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows servers.
3. This mapping is valid only if DB2 can determine the data type of the column.
4. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

Table 41 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 UDB 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 UDB in the OS/390 or z/OS environment, 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 41. 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
SMALLINT	short, java.lang.Integer
INTEGER	int, java.lang.Integer
BIGINT[1]	long
REAL	float, java.lang.Float
DOUBLE	double, java.lang.Double
DECIMAL	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[]
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
DATALINK	java.net.URL[2]

| *Table 41. Mappings of SQL data types in a CREATE PROCEDURE or CREATE FUNCTION statement to data types in the corresponding Java stored procedure or user-defined function program (continued)*

SQL data type in CREATE PROCEDURE or CREATE FUNCTION	Data type in Java stored procedure or user-defined function method

Notes:

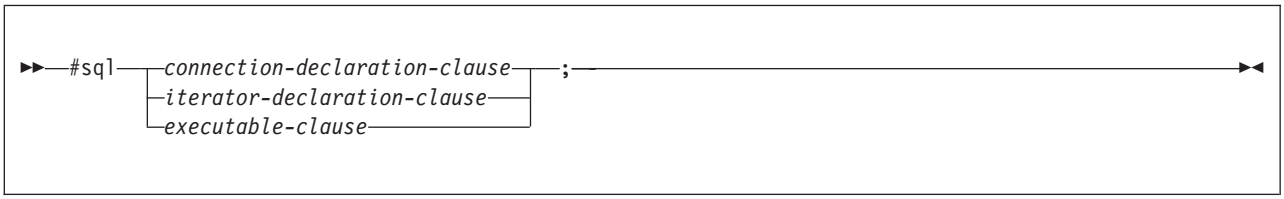
1. The BIGINT SQL type is available only on DB2 UDB for Linux, UNIX and Windows servers.
2. The DATALINK data type is supported only by the DB2 JDBC Type 2 Driver for Linux, UNIX and Windows.

SQLJ syntax

The following topics contain information about the syntax of SQLJ clauses:

SQLJ clause

The SQL statements in an SQLJ program are in SQLJ clauses. The general syntax of an SQLJ clause is:

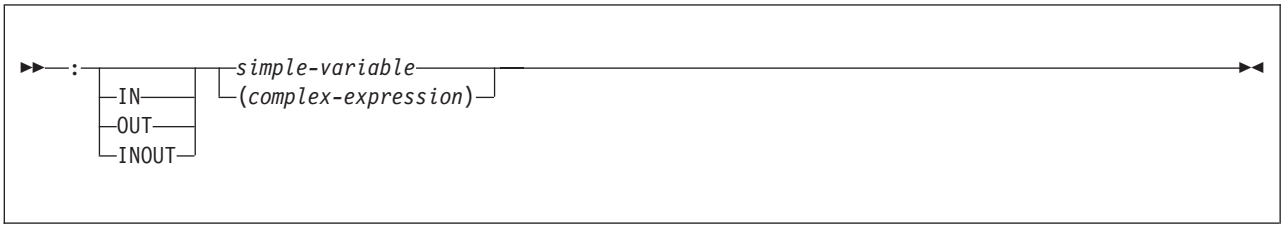


Keywords in an SQLJ clause are case sensitive, unless those keywords are part of an SQL statement in an executable clause.

SQLJ host-expression

A host expression is a Java variable or expression that is referenced by SQLJ clauses in an SQLJ application program.

Syntax:



Description:

: Indicates that the variable or expression that follows is a host expression. The colon must immediately precede the variable or expression.

IN | OUT | INOUT
For a host expression that is used as a parameter in a stored procedure call, identifies whether the parameter provides data to the stored procedure (IN), retrieves data from the stored procedure (OUT), or does both (INOUT). The default is IN.

simple-variable
Specifies a Java unqualified identifier.

complex-expression
Specifies a Java expression that results in a single value.

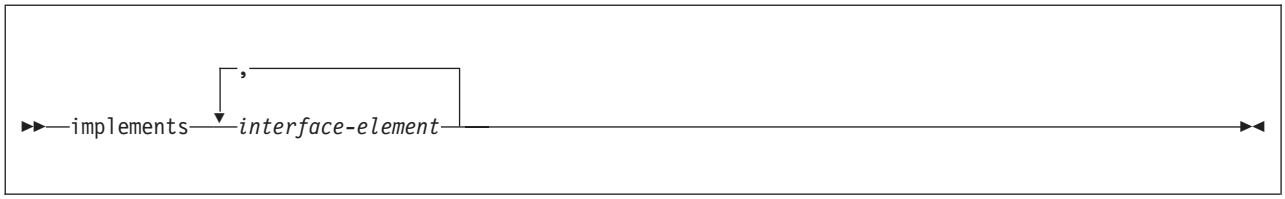
Usage notes:

- A complex expression must be enclosed in parentheses.
- ANSI/ISO rules govern where a host expression can appear in a static SQL statement.

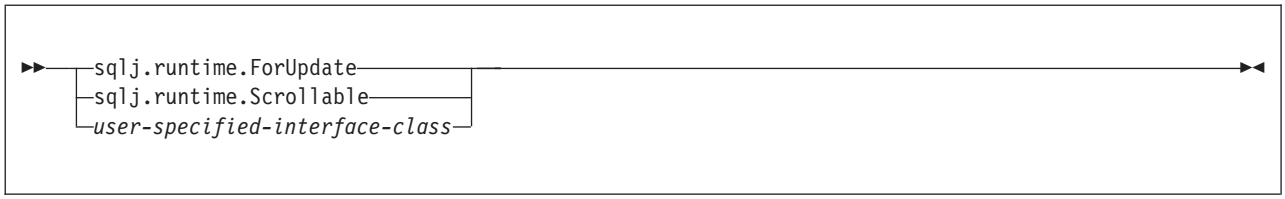
SQLJ implements-clause

The implements clause derives one or more classes from a Java interface.

Syntax:



interface-element:



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 "Performing positioned UPDATE and DELETE operations in an SQLJ application" on page 78 for information on performing a positioned UPDATE or positioned DELETE operation in SQLJ.

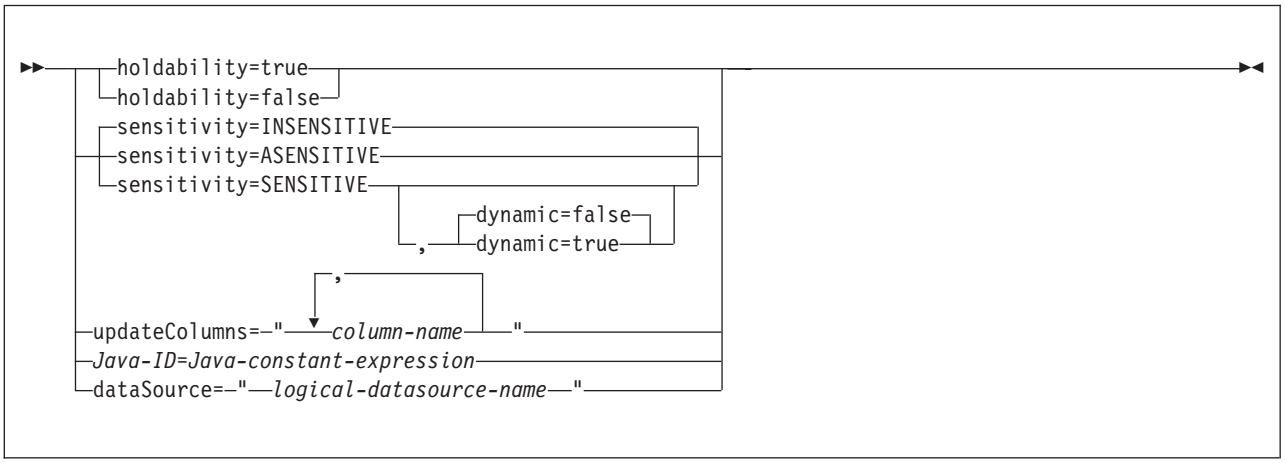
You need to implement `sqlj.runtime.Scrollable` when you declare a scrollable iterator. See "Using scrollable iterators in an SQLJ application" on page 102 for information on scrollable iterators.

SQLJ with-clause

The with clause specifies a set of one or more attributes for an iterator or a connection context.

Syntax:

with-element:



Description:

holdability
For an iterator, specifies whether an iterator keeps its position in a table after a COMMIT is executed. The value for holdability must be `true` or `false`.

sensitivity
For an iterator, specifies whether changes that are made to the underlying table can be visible to the iterator after it is opened. The value must be INSENSITIVE, SENSITIVE, or ASENSITIVE. The default is INSENSITIVE.

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`.

If the value of dynamic is `true`, the data source must support dynamic scrollable cursors.

DB2 UDB for Linux, UNIX and Windows servers do not support dynamic scrollable cursors. Specify `true` only if your application accesses data on DB2 UDB for z/OS servers, at Version 8 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 DB2 Universal JDBC Driver.

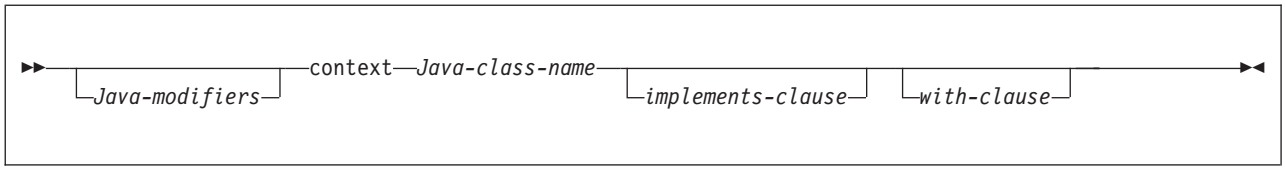
Usage notes:

- The value on the left side of a with element must be unique within its with clause.
- If you specify updateColumns in a with element of an iterator declaration clause, the iterator declaration clause must also contain an implements clause that specifies the sqlj.runtime.ForUpdate interface.
- If you do not customize your SQLJ program, the JDBC driver ignores the value of holdability that is in the with clause. Instead, the driver uses the JDBC driver setting for holdability.

SQLJ connection-declaration-clause

The connection declaration clause declares a connection to a data source in an SQLJ application program.

Syntax:



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" on page 133 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" on page 134 for a description of this clause.

Usage notes:

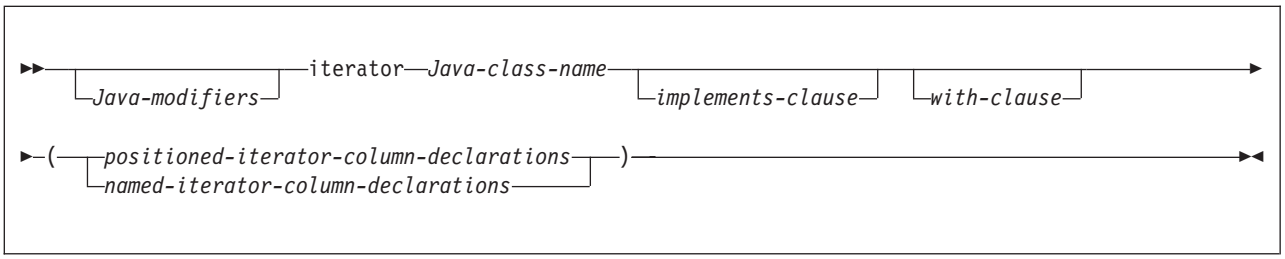
- SQLJ generates a connection class declaration for each connection declaration clause you specify. SQLJ data source connections are objects of those generated connection classes.
- You can specify a connection declaration clause anywhere that a Java class definition can appear in a Java program.

SQLJ iterator-declaration-clause

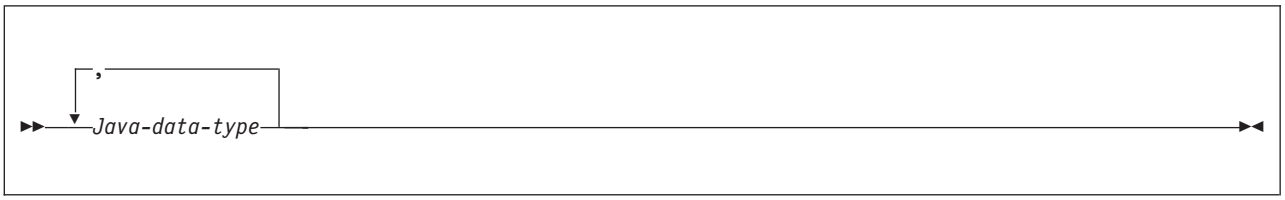
An iterator declaration clause declares a positioned iterator class or a named iterator class in an SQLJ application program. An iterator contains the result table from a query. SQLJ generates an iterator class for each iterator declaration clause you specify. An iterator is an object of an iterator class.

An iterator declaration clause has a form for a positioned iterator and a form for a named iterator. The two kinds of iterators are distinct and incompatible Java types that are implemented with different interfaces.

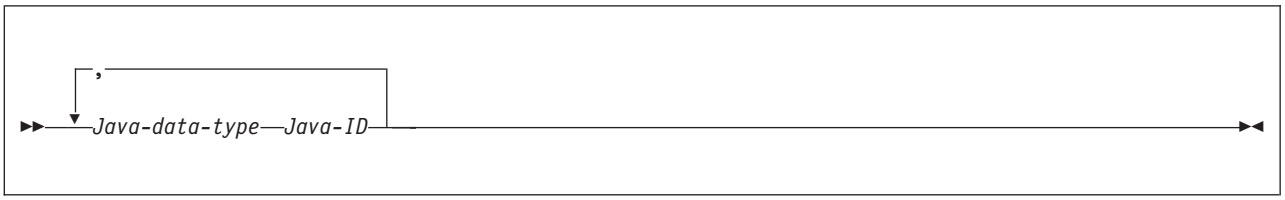
Syntax:



positioned-iterator-column declarations:



named-iterator-column-declarations:



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" on page 133 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" on page 134 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" on page 127 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" on page 127 for a list of compatible data types.

Usage notes:

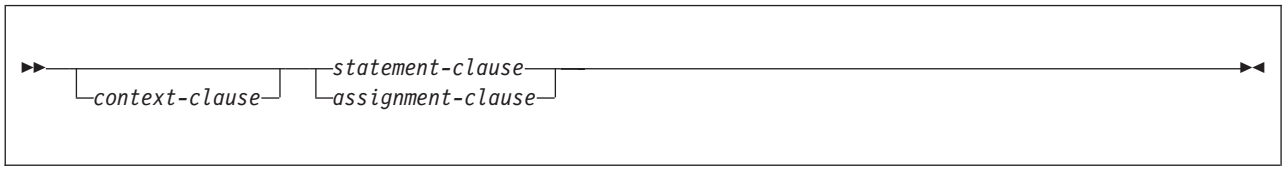
- An iterator declaration clause can appear anywhere in a Java program that a Java class declaration can appear.
- When a named iterator declaration contains more than one pair of Java data types and Java IDs, all Java IDs within the list must be unique.

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:

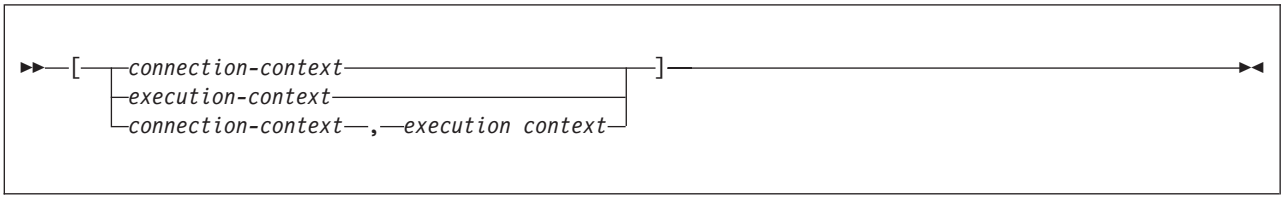
- An executable clause can appear anywhere in a Java program that a Java statement can appear.
- SQLJ reports negative SQL codes from executable clauses through class `java.sql.SQLException`.

  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.

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:



Description:

connection-context
Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of the connection context class that SQLJ generates for a connection declaration clause.

execution-context
Specifies a valid Java identifier that is declared earlier in the SQLJ program. That identifier must be declared as an instance of class `sqlj.runtime.ExecutionContext`.

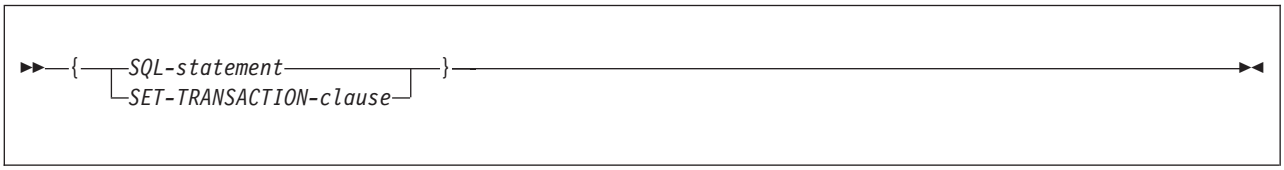
Usage notes:

- If you do not specify a connection context in an executable clause, SQLJ uses the default connection context.
- If you do not specify an execution context, SQLJ obtains the execution context from the connection context of the statement.

SQLJ statement-clause

A statement clause contains an SQL statement or a SET TRANSACTION clause.

Syntax:



Description:

SQL-statement
You can include the DB2 UDB for z/OS SQL statements in Table 42 on page 139 in a statement clause.

For information on individual SQL statements, see *DB2 SQL Reference*.

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" on page 140 for more information.

Table 42. 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
RENAME (JDBC/SQLJ driver for z/OS only)
REVOKE
ROLLBACK
SAVEPOINT
SELECT INTO
SET CURRENT APPLICATION ENCODING SCHEME

Table 42. Valid SQL statements in an SQLJ statement clause (continued)

SET CURRENT DEGREE
SET CURRENT LOCALE LC_CTYPE
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
SET CURRENT OPTIMIZATION HINT
SET CURRENT PACKAGE PATH
SET CURRENT PACKAGESET (USER is not supported)
SET CURRENT PRECISION
SET CURRENT REFRESH AGE
SET CURRENT RULES
SET CURRENT SQLID
SET PATH
SIGNAL SQLSTATE (JDBC/SQLJ driver for z/OS only)
UPDATE

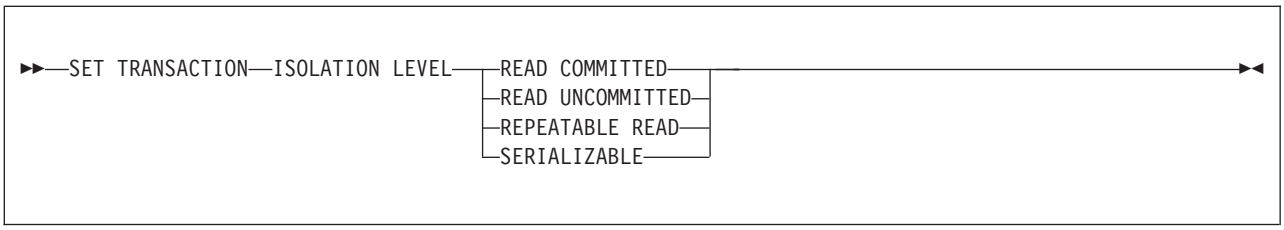
Usage notes:

- 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.

SQLJ SET-TRANSACTION-clause

The SET TRANSACTION clause sets the isolation level for the current unit of work.

Syntax:



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.

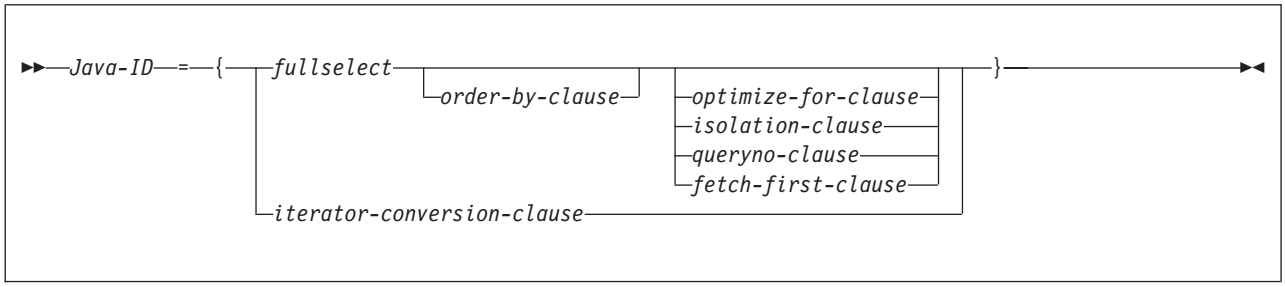
Usage notes:

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:



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.

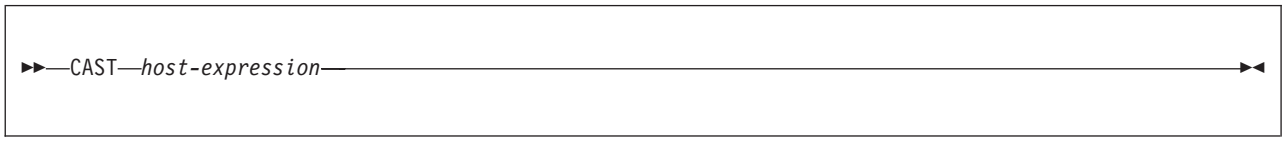
Usage notes:

- 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" on page 127 for a list of compatible Java and SQL data types.
- If the object that is identified by *Java-ID* is a named iterator, the name of each accessor method must match, except for case, the name of a column in the result set. In addition, the data type of the object that an accessor method returns must be compatible with the data type of the corresponding column in the result set.
- You can put an assignment clause anywhere in a Java program that a Java assignment statement can appear. However, you cannot put an assignment clause where a Java assignment expression can appear. For example, you cannot specify an assignment clause in the control list of a for statement.

SQLJ iterator-conversion-clause

The iterator conversion clause converts a JDBC `ResultSet` to an iterator.

Syntax:



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.

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 43 summarizes the interfaces in sqlj.runtime.

Table 43. 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 44 summarizes the classes in sqlj.runtime.

Table 44. Summary of sqlj.runtime classes

Class name	Purpose
AsciiStream	A class for handling an input stream whose bytes should be interpreted as ASCII.
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.

Table 44. Summary of sqlj.runtime classes (continued)

Class name	Purpose
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.

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 DB2 Universal JDBC Driver 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.

sqlj.runtime.ForUpdate interface

SQLJ implements the `sqlj.runtime.ForUpdate` interface in SQLJ programs that contain an iterator declaration clause with `implements sqlj.runtime.ForUpdate`. An SQLJ program that does positioned UPDATE or DELETE operations (UPDATE...WHERE CURRENT OF or DELETE...WHERE CURRENT OF) must include an iterator declaration clause with `implements sqlj.runtime.ForUpdate`.

Methods:

getCursorName

Format:

```
public abstract String getCursorName() throws SQLException
```

This method is used by code that is generated by the SQLJ translator. It is not intended for direct use by application programs.

sqlj.runtime.NamedIterator interface

The `sqlj.runtime.NamedIterator` interface is implemented when an SQLJ application executes an iterator declaration clause for a named iterator. A named iterator includes result table column names, and the order of the columns in the iterator is not important.

An implementation of the `sqlj.runtime.NamedIterator` interface includes an accessor method for each column in the result table. An accessor method returns

the data from its column of the result table. The name of an accessor method matches the name of the corresponding column in the named iterator.

Methods (inherited from the `ResultSetIterator` interface):

close
Format:
```
public abstract void close() throws SQLException
```

Releases database resources that the iterator uses.

isClosed
Format:
```
public abstract boolean isClosed() throws SQLException
```

Returns a value of `true` if the close method has been invoked. Returns `false` if the close method has not been invoked.

next
Format:
```
public abstract boolean next() throws SQLException
```

Advances the iterator to the next row. Before an instance of the `next` method is invoked for the first time, the iterator is positioned before the first row of the result table. `next` returns a value of `true` when a next row is available and `false` when all rows have been retrieved.

sqlj.runtime.PositionedIterator interface

The `sqlj.runtime.PositionedIterator` interface is implemented when an SQLJ application executes an iterator declaration clause for a positioned iterator. The order of columns in a positioned iterator must be the same as the order of columns in the result table, and a positioned iterator does not include result table column names.

Methods: `sqlj.runtime.PositionedIterator` inherits all **ResultSetIterator** methods, and includes the following additional method:

endFetch
Format:
```
public abstract boolean endFetch() throws SQLException
```

Returns a value of `true` if the iterator is not positioned on a row. Returns a value of `false` if the iterator is positioned on a row.

sqlj.runtime.ResultSetIterator interface

The `sqlj.runtime.ResultSetIterator` interface is implemented by SQLJ for all iterator declaration clauses.

An untyped iterator can be generated by declaring an instance of the `sqlj.runtime.ResultSetIterator` interface directly. In general, use of untyped iterators is not recommended.

Variables:

ASENSITIVE
Format:
```
public static final int ASENSITIVE
```

A constant that can be returned by the getSensitivity method. It indicates that the iterator is defined as ASENSITIVE.

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.

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 the iterator is positioned after the last row. If *n*<0 and *x*+*n*<1, the iterator is positioned before the first row.

Returns `true` if the iterator is on a row. Otherwise, returns `false`.

setFetchDirection(int)

Format:

```
public abstract void setFetchDirection (int) throws SQLException
```

Gives the SQLJ runtime environment a hint as to the direction in which rows of this iterator object are processed. Possible values are:

sqlj.runtime.ResultSetIterator.FETCH_FORWARD
Rows are processed in a forward direction, from first to last.

sqlj.runtime.ResultSetIterator.FETCH_REVERSE
Rows are processed in a backward direction, from last to first.

sqlj.runtime.ResultSetIterator.FETCH_UNKNOWN
The order of processing is not known.

sqlj.runtime.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.

`getNextResultSet(int current)` requires JDK 1.4 or later.

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 character,

graphic, or varying-length binary 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.

sqlj.runtime.SQLNullException class

The `sqlj.runtime.SQLNullException` class is derived from the `java.sql.SQLException` class. An `sqlj.runtime.SQLNullException` is thrown when an SQL NULL value is fetched into a host identifier with a Java primitive type. The SQLSTATE value for an instance of `SQLNullException` is '22002'.

sqlj.runtime.StreamWrapper class

The `sqlj.runtime.StreamWrapper` class wraps a `java.io.InputStream` instance and extends the `java.io.InputStream` class. 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.

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.

DB2 Universal JDBC Driver reference information

The following topics contain information that is specific to the DB2 Universal JDBC Driver:

- "DB2 Universal JDBC Driver extensions to JDBC"
- "JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 179
- "SQLJ differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers" on page 182
- "Error codes issued by the DB2 Universal JDBC Driver" on page 183
- "SQLSTATEs issued by the DB2 Universal JDBC Driver" on page 183
- "How to find DB2 Universal JDBC Driver version and environment information" on page 184
- "Properties for the DB2 Universal JDBC Driver" on page 185

DB2 Universal JDBC Driver extensions to JDBC

This topic describes the JDBC APIs that are specific to the DB2 Universal JDBC Driver.

To use any of the methods that are described in this topic, you must 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");
```

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 DB2 Universal JDBC Driver. See "Properties for the DB2 Universal JDBC Driver" on page 185 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 45 lists the DB2 Universal JDBC Driver properties and their data types. See "Properties for the DB2 Universal JDBC Driver" on page 185 for definitions of these properties.

Table 45. DB2 Universal JDBC Driver properties and their data types

	Property name	Data type
#	com.ibm.db2.jcc.DB2BaseDataSource.accountingInterval	String
	com.ibm.db2.jcc.DB2BaseDataSource.clientAccountingInformation	String
	com.ibm.db2.jcc.DB2BaseDataSource.clientApplicationInformation	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.cliSchema	String
	com.ibm.db2.jcc.DB2BaseDataSource.currentFunctionPath	String
#	com.ibm.db2.jcc.DB2BaseDataSource.currentMaintainedTableTypesForOptimization	String
	com.ibm.db2.jcc.DB2BaseDataSource.currentPackagePath	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.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.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.logWriter	PrintWriter

Table 45. DB2 Universal JDBC Driver properties and their data types (continued)

	Property name	Data type
	com.ibm.db2.jcc.DB2BaseDataSource.portNumber	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.securityMechanism	int
#	com.ibm.db2.jcc.DB2BaseDataSource.sendDataAsIs	boolean
	com.ibm.db2.jcc.DB2BaseDataSource.serverName	String
#	com.ibm.db2.jcc.DB2BaseDataSource.supportsAsynchronousXARollback	int
	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.user	String
#	com.ibm.db2.jcc.DB2BaseDataSource.useTargetColumnEncoding	boolean

DB2BaseDataSource methods:

In addition to the `getXXX` and `setXXX` methods for the DB2BaseDataSource properties, the following methods are defined only for the DB2 Universal JDBC Driver.

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`

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 DB2 Universal JDBC Driver.

getDB2ClientProgramId

Format:

```
# public String getDB2ClientProgramId()
#   throws 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 SQLException
```

Returns application information for the current client.

getDB2ClientUser

Format:

```
public String getDB2ClientUser()
  throws 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 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 SQLException
```

Returns the list of DB2 package collections that are searched for the DB2 Universal JDBC Driver packages.

getDB2CurrentPackageSet

Format:

```
public String getDB2CurrentPackageSet()
  throws SQLException
```

Returns the collection ID for the connection.

| **getDB2SystemMonitor**

| Format:

```
| public abstract DB2SystemMonitor getDB2SystemMonitor()
|   throws SQLException
```

| Returns the system monitor object for the connection. Each DB2 Universal | JDBC Driver connection can have a single system monitor. See | "DB2SystemMonitor interface" on page 176 for more information.

getJccLogWriter

Format:

```
public PrintWriter getJccLogWriter()
  throws SQLException
```

Returns the current trace destination for the DB2 Universal JDBC Driver trace.

installDB2JavaStoredProcedure

Format:

```
void DB2Connection.installDB2JavaStoredProcedure(java.io.InputStream jarFile,
  int jarFileLength, String jarId)
```

Invokes the SQLJ.DB2_INSTALL_JAR stored procedure on a DB2 UDB for z/OS server to create a new definition of a JAR file in the DB2 catalog for that server.

isDB2GatewayConnection

Format:

```
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.

replaceDB2JavaStoredProcedure

Format:

```
void DB2Connection.replaceDB2JavaStoredProcedure(java.io.InputStream jarFile,
  int jarFileLength, String jarId)
  throws SQLException
```

Invokes the SQLJ.DB2_REPLACE_JAR stored procedure on a DB2 UDB for z/OS server to replace a definition of a JAR file in the DB2 catalog for that server.

resetDB2Connection

Format:

```
public void resetDB2Connection(String user, String password,
  DB2BaseDataSource ds)
  throws SQLException
void DB2Connection.resetDB2Connection()
  throws SQLException
```

Reopens a physical connection for immediate reuse. The first form of `resetDB2Connection` reopens the connection with a new user ID, password, and connection properties. The second form of `resetDB2Connection` the connection with the same user ID, password, and connection properties.

setDB2ClientAccountingInformation

Format:

```
public void setDB2ClientAccountingInformation(String info)
  throws SQLException
```

Specifies accounting information for the connection. This information is for client accounting purposes. This value can change during a connection.

`setDB2ClientAccountingInformation` sets the value in the CLIENT ACCTNG special register.

Parameter description:

info
User-specified accounting information. The maximum length depends on the server. For a DB2 UDB for OS/390 or 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.

setDB2ClientApplicationInformation
Format:

```
public void setDB2ClientApplicationInformation(String info)
  throws SQLException
```

Specifies application information for the connection. This information is for client accounting purposes. This value can change during a connection.

`setDB2ClientApplicationInformation` sets the value in the CLIENT APPLNAME special register.

Parameter description:

info
User-specified application information. The maximum length depends on the server. For a DB2 UDB for OS/390 or z/OS server, the maximum length is 32 bytes. A Java empty string ("") is valid for this parameter value, but a Java `null` value is not valid.

setDB2ClientProgramId
Format:

```
public abstract void setDB2ClientProgramId(String program-ID)
  throws java.sql.SQLException
```

Sets a user-defined program identifier for the client. The program identifier can be used to identify the application at the database server.

setDB2ClientUser
Format:

```
public void setDB2ClientUser(String user)
  throws 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.

`setDB2ClientUser` sets the value in the CLIENT USERID special register.

Parameter description:

user
The user ID for the current client.The maximum length depends on the server. For a DB2 UDB for OS/390 or 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 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.

`setDB2ClientWorkstation` sets the value in the CLIENT WRKSTNNAME special register.

Parameter description:

name
The workstation name for the current client.The maximum length depends on the server. For a DB2 UDB for OS/390 or 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 SQLException
```

Specifies a list of collection IDs that DB2 searches for the DB2 Universal JDBC Driver DB2 packages.

Parameter description:

packagePath
A comma-separated list of collection IDs.

setDB2CurrentPackageSet
Format:

```
public void setDB2CurrentPackageSet(String packageSet)
  throws SQLException
```

Specifies the collection ID for the connection. When you set this value, you also set the collection ID of the DB2 Universal JDBC Driver 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 SQLException

public void setJccLogWriter(PrintWriter logWriter, int traceLevel)
  throws SQLException
```

Enables or disables the DB2 Universal JDBC Driver trace, or changes the trace destination during an active connection.

Parameter descriptions:

logWriter
An object of type `java.io.PrintWriter` to which the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver" on page 185 for valid values.

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 DB2 Universal JDBC Driver.

getSqlca
Format:

```
public DB2Sqlca getSqlca()
```

Returns a `DB2Sqlca` object from a `java.sql.Exception` that is produced under a DB2 Universal JDBC Driver.

getThrowable
Format:

```
public Throwable getThrowable()
```

Returns a `java.lang.Throwable` object from a `java.sql.Exception` that is produced under a DB2 Universal JDBC Driver.

printTrace
Format:

```
static public void printTrace(java.io.PrintWriter printWriter,
  String header)
```

Prints diagnostic information after a `java.sql.Exception` is thrown under a DB2 Universal JDBC Driver.

Parameter descriptions:

printWriter
The destination for the diagnostic information.

header
User-defined information that is printed at the beginning of the output.

DB2ExceptionFormatter class:

The `com.ibm.db2.jcc.DB2ExceptionFormatter` class contains methods for printing diagnostic information to a stream.

DB2ExceptionFormatter methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

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.

DB2JccDataSource interface:

The `com.ibm.db2.jcc.DB2JccDataSource` interface can be used to distinguish between `com.ibm.db2.jcc.DB2BaseDataSource` instances for the DB2 Universal JDBC Driver and `com.ibm.db2.jcc.DB2BaseDataSource` instances for the JDBC/SQLJ Driver for OS/390 and z/OS. If a `DataSource` instance implements `com.ibm.db2.jcc.DB2JccDataSource`, it is a DB2 Universal JDBC Driver `DataSource` instance. Otherwise, it is an JDBC/SQLJ Driver for OS/390 and z/OS `DataSource` instance.

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 DB2 Universal JDBC Driver.

executeDB2QueryBatch

Format:

```
public void executeDB2QueryBatch()
  throws java.sql.SQLException
```

Executes a statement batch that contains queries with parameters.

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 DB2 Universal JDBC Driver.

getBytes

Format:

```
public byte[] getBytes()
```

Converts a com.ibm.jcc.DB2RowID object to bytes.

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 DB2 Universal JDBC Driver-only properties.

DB2SimpleDataSource properties:

The following property is defined only for the DB2 Universal JDBC Driver. See "Properties for the DB2 Universal JDBC Driver" on page 185 for an explanation of this property.

String com.ibm.db2.jcc.DB2SimpleDataSource.password

DB2SimpleDataSource methods:

The following method is defined only for the DB2 Universal JDBC Driver.

setPassword

Format:

```
public void setPassword(String password)
```

Sets the password for the `DB2SimpleDataSource` object. There is no corresponding `getPassword` method. Therefore, the password cannot be encrypted because there is no way to retrieve the password so that you can decrypt it.

DB2Sqlca class:

The `com.ibm.db2.jcc.DB2Sqlca` class is an encapsulation of the DB2 SQLCA. For an explanation of the SQLCA fields, see *DB2 SQL Reference*.

DB2Sqlca methods:

The following methods are defined only for the DB2 Universal JDBC Driver.

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.

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 DB2 Universal JDBC Driver.

getDB2ClientProgramId
Format:

```
# public String getDB2ClientProgramId()
#   throws SQLException
```

Returns the user-defined program identifier for the client. The program identifier can be used to identify the application at the database server.

setDB2ClientProgramId

Format:

```
public abstract void setDB2ClientProgramId(String program-ID)
  throws java.sql.SQLException
```

Sets a user-defined program identifier for the client. The program identifier can be used to identify the application at the database server.

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 DB2 Universal JDBC Driver.

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 DB2 Universal JDBC Driver.

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 UDB 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`.

JDBC differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers

The DB2 Universal JDBC Driver differs from the JDBC/SQLJ Driver for OS/390 and z/OS in the following areas:

Supported methods:

The DB2 Universal JDBC Driver supports a number of JDBC methods that the other drivers do not support, and does not support several methods that the other drivers support. For details, see "Comparison of driver support for JDBC APIs" on page 107.

Support for scrollable and updatable ResultSets:

The DB2 Universal JDBC Driver supports scrollable and updatable `ResultSets`.

The JDBC/SQLJ driver for z/OS support only non-scrollable and non-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:

- "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 10
- "Connecting to a data source using the DriverManager interface with a JDBC/SQLJ Driver for OS/390 and z/OS" on page 54

Difference in error codes and SQLSTATEs returned for driver errors:

The DB2 Universal JDBC Driver does not use existing SQLCODEs or SQLSTATEs for internal errors, as the other drivers do. See "Error codes issued by the DB2 Universal JDBC Driver" on page 183 and "SQLSTATEs issued by the DB2 Universal JDBC Driver" on page 183.

The JDBC/SQLJ driver for z/OS return SQLSTATE FFFFF when internal errors occur.

Security mechanisms:

The JDBC drivers have different security mechanisms.

For information on DB2 Universal JDBC Driver security mechanisms, see"Security under the DB2 Universal JDBC Driver" on page 289.

For information on security mechanisms for the JDBC/SQLJ driver for z/OS, see "Security under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 298.

How connection properties are set:

With Universal Driver type 4 connectivity, you set properties for a connection by setting the properties for the associated `DataSource` or `Connection` object.

With Universal Driver type 2 connectivity, you set properties for a connection in one of these ways:

- You can set properties only for a connection by setting the properties for the associated `DataSource` or `Connection` object.
- You can set driver-wide properties through an optional run-time properties file.

For the JDBC/SQLJ driver for z/OS, you set properties through the JDBC/SQLJ run-time properties file.

Support for read-only connections:

With the DB2 Universal JDBC Driver, you can make a connection read-only through the `readOnly` property for a `Connection` or `DataSource` object.

The JDBC/SQLJ driver for z/OS does not support read-only connections.

Results returned from ResultSet.getString for a BIT DATA column:

The DB2 Universal JDBC Driver 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 JDBC/SQLJ 2.0 Driver for OS/390 and z/OS returns the data in the encoding scheme of the caller.

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 DB2 Universal JDBC Driver does not throw an exception until the subsequent `PreparedStatement.executeUpdate` method executes. Therefore, for the DB2 Universal JDBC Driver, 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 JDBC/SQLJ 2.0 Driver for OS/390 and z/OS throws an exception after the `PreparedStatement.setBinaryStream`, `PreparedStatement.setCharacterStream`, or `PreparedStatement.setUnicodeStream` method executes.

Default mappings for PreparedStatement.setXXXStream:

With the DB2 Universal JDBC Driver, 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 JDBC/SQLJ driver for z/OS, 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 DB2 Universal JDBC Driver, 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 DB2 Universal JDBC Driver supports are limited to those that are supported by the underlying JRE implementation.

A DB2 Universal JDBC Driver client using type 4 connectivity sends data to the database server as Unicode UTF-8.

The conversions that the JDBC/SQLJ driver for z/OS and the DB2 Universal JDBC Driver with type 2 connectivity support are also limited to those that are supported by the underlying JRE implementation.

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 DB2 Universal JDBC Driver, 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 JDBC/SQLJ driver for z/OS, 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.

Result of using getBoolean to retrieve a value from a CHAR column:

With the DB2 Universal JDBC Driver, 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 JDBC/SQLJ driver for z/OS, when you execute `ResultSet.getBoolean` or `CallableStatement.getBoolean` to retrieve a Boolean value from a CHAR column, and the column contains the value "0", the value `false` is returned. If the column contains any other value, `true` is returned.

Internal use of LOB locators by the JDBC drivers:

The DB2 Universal JDBC Driver uses LOB locators internally under the following circumstances:
- Always, for fetching data from scrollable cursors
- Never, for fetching data from stored procedure output parameters
- If the `fullyMaterializeLobData` connection property is set to `false`, in all other cases

The JDBC/SQLJ 2.0 Driver for OS/390 and z/OS does not use LOB locators.

SQLJ differences between the DB2 Universal JDBC Driver and other DB2 JDBC drivers

SQLJ support in the DB2 Universal JDBC Driver differs from SQLJ support in the other DB2 JDBC drivers in the following areas:

Connection associated with the default connection context:

If you are using the `DataSource` interface to connect to a data source, before you can use a default connection context, the logical name `jdbc/defaultDataSource` must be registered with JNDI. The JDBC/SQLJ 2.0 Driver for OS/390 and z/OS creates a connection to the local data source for the default connection context.

Production of DBRMs during SQLJ program preparation:

The SQLJ program preparation process for the DB2 Universal JDBC Driver does not produce DBRMs. Therefore, with the DB2 Universal JDBC Driver, you can produce DB2 packages only by using the DB2 Universal JDBC Driver utilities.

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 "Connecting to a data source using SQLJ" on page 66 for more information.

Support for scrollable and updatable iterators:

SQLJ with the DB2 Universal JDBC Driver supports scrollable and updatable iterators.

The JDBC/SQLJ driver for z/OS support only non-scrollable and non-updatable iterators.

Dynamic execution of SQL statements under WebSphere Application Server:

With the DB2 Universal JDBC Driver, if you are using a version of WebSphere Application Server for z/OS and OS/390 before version 5.0.2, all SQL statements in an SQLJ program are executed dynamically, regardless of whether you customize

the SQLJ program. For WebSphere Application Server for z/OS and OS/390 Version 5.0.2 and above, if you customize your SQLJ program, SQL statements are executed statically.

Error codes issued by the DB2 Universal JDBC Driver

Error codes in the ranges +4200 to +4299, +4450 to +4499, -4200 to -4299, and -4450 to -4499 are reserved for the DB2 Universal JDBC Driver. Currently, the DB2 Universal JDBC Driver issues the following error codes:

Table 46. Error codes issued by the DB2 Universal JDBC Driver

Error Code	Explanation
-4200	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	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.
-4203	An error occurred on an XA connection during execution of an SQL statement. For network optimization, the DB2 Universal JDBC Driver 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.
-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 was reestablished. The host name or IP address is *host-name*\ and the service name or port number is *port*. Any DB2 special registers that were modified during the original connection are reestablished.
-4499	A fatal error occurred that resulted in a disconnect.
-99999	The DB2 Universal JDBC Driver issued an error that does not yet have an error code.

SQLSTATEs issued by the DB2 Universal JDBC Driver

SQLSTATEs in the range 46600 to 466ZZ are reserved for the DB2 Universal JDBC Driver. Currently, the DB2 Universal JDBC Driver returns a null SQLSTATE value for an internal error, unless the error is a DRDA error. The following SQLSTATEs are issued for DRDA errors:

08004 The application server rejected establishment of the connection.

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.

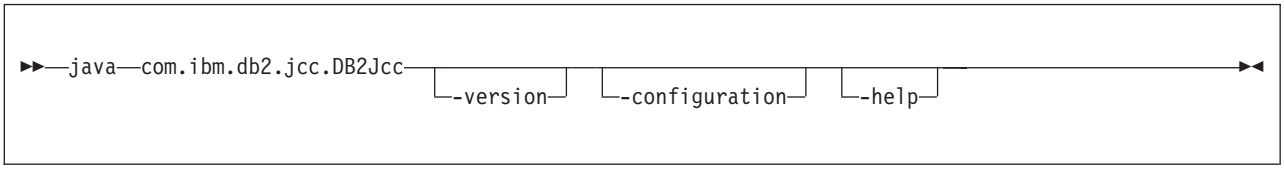
58016 The DDM parameter is not supported.

58017 The DDM parameter value is not supported.

How to find DB2 Universal JDBC Driver version and environment information

To determine the version of the DB2 Universal JDBC Driver, as well as information about the environment in which the driver is running, run the DB2Jcc utility on the UNIX System Services command line.

DB2Jcc syntax:



DB2Jcc option descriptions:

-version
Specifies that the DB2 Universal JDBC Driver displays its name and version.

-configuration
Specifies that the DB2 Universal JDBC Driver 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 JDBC Universal Driver 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 JDBC Universal Driver Architecture 2.1.29 Test Build
[ibm][db2][jcc] Compatible JRE versions: { 1.3, 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 1.3.1
[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/v81/bldsupp/AIX/jdk1.3.1/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 = 1.3.1
[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 = 1.3
[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/v81/bldsupp/AI
X/jdk1.3.1/jdbc2.0_stdext/jdbc2_0-stdext.jar:/wsdb/v81/bldsupp/AIX/jdk1.3.1/jta1.0.1/jta-spec1_0_1.j
ar:/wsdb/v81/bldsupp/AIX/jdk1.3.1/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/v81/bldsupp/AIX/jdk1.3.1/jre/bin:/wsdb/v81/bldsupp/
AIX/jdk1.3.1/jre/bin/classic:/home/myid/sqllib/lib:/local/cobol:/usr/lib
[ibm][db2][jcc] Path of extension directory or directories = /wsdb/v81/bldsupp/AIX/jdk1.3.1/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 62. Sample DB2Jcc output

Properties for the DB2 Universal JDBC Driver

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 "DB2 Universal JDBC Driver extensions to JDBC" on page 165 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 "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 10.

- In a `java.lang.String` value in the *url* parameter of a DriverManager.getConnection call, as shown in "Connecting to a data source using the DriverManager interface with the DB2 Universal JDBC Driver" on page 10.

The properties are:

accountingInterval
Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. The data type of this property is String. If the value of accountingInterval is "COMMIT", DB2 accounting records are produced at commit points. Otherwise, accounting records are produced on termination of the physical connection to the data source.

accountingInterval applies only to Universal Driver type 2 connectivity on DB2 UDB for z/OS. accountingInterval is not applicable to connections under CICS or IMS, or for Java stored procedures.

The accountingInterval property overrides the db2.jcc.accountingInterval configuration property.

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 UDB for OS/390 or 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 UDB for z/OS server. Database administrators can use this property to correlate work on a DB2 UDB 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 DB2 Universal JDBC Driver supplies a value of db2jcc*thread-name*.

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 UDB for OS/390 or 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 UDB for OS/390 or 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.

cliSchema
Specifies the schema of the DB2 shadow catalog tables or views that are searched when an application invokes a `DatabaseMetaData` method.

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 UDB for OS/390 or z/OS server, the

maximum length is 2048 bytes. The value is a comma-separated list of schema names. Those names can be ordinary or delimited identifiers.

currentMaintainedTableTypesForOptimization
Specifies a value that identifies the types of objects that can be considered when 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 the DB2 packages for the DB2 Universal JDBC Driver.

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 DB2 packages for the DB2 Universal JDBC Driver. The data type of this property is String. The default is NULLID for Universal Driver type 4 connectivity. For Universal Driver type 2 connectivity, if a value for currentPackageSet is not specified, the property value is not set. If currentPackageSet is set, its value overrides the value of jdbcCollection.

Multiple instances of the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver instance. To choose an instance of the DB2 Universal JDBC Driver for a connection, you specify a currentPackageSet value that matches the collection ID for one of the DB2 Universal JDBC Driver 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.

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. currentSchema is available only in DB2 Version 8 new-function mode.

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 UDB 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 Universal Driver type 4 connectivity or Universal Driver type 2 connectivity is used.

For Universal Driver 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 UDB 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 Universal Driver type 2 connectivity:

- The databaseName value is the location name for the data source. The location name is defined in the SYSIBM.LOCATIONS catalog table.

If the databaseName property is not set, the connection location depends on the type of environment in which the connection is made. If the connection is made in an environment such as a stored procedure, CICS, or IMS environment, where a DB2 connection to a location is previously established, that connection is used. The connection URL for this case is jdbc:default:connection:. If a connection to DB2 is not previously established, the connection is to the local site. The connection URL for this case is jdbc:db2os390: or jdbc:db2os390sqlj:.

deferPrepares
Specifies whether to defer prepare operations until run time. The data type of this property is boolean. The default is `true` for Universal Driver type 4 connectivity. The property is not applicable to Universal Driver 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 DB2 Universal # JDBC Driver is enabled. The connection concentrator function is available only # for connections to DB2 UDB 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 DB2 # Universal JDBC Driver is enabled. The Sysplex workload balancing function is # available only for connections to DB2 UDB 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.

fullyMaterializeLobData
Indicates whether the driver retrieves LOB locators for FETCH operations. The data type of this property is boolean. If the value is `true`, LOB data is fully materialized within the JDBC driver when a row is fetched. If this 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`.

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.

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 Universal Driver type 4 connectivity.

For more information on using Kerberos security with the DB2 Universal JDBC Driver, see "Kerberos security under the DB2 Universal JDBC Driver" on page 293.

jdbcCollection
Specifies the collection ID for the packages that are used by an instance of the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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 Universal Driver type 2 connectivity on DB2 UDB for z/OS.

keepDynamic
Specifies whether the DB2 server keeps already prepared dynamic SQL statements in the dynamic statement cache after commit points so that those prepared statements can be reused. The data type of this property is int. Valid values that you can specify are `YES` (1) and `NO` (2). keepDynamic is applicable only for connections to DB2 for z/OS database servers.

If the keepDynamic property is not specified, the keepDynamic value is `NOT_SET` (0). If the connection is to a DB2 UDB for z/OS server, caching of dynamic statements for a connection is not done. If the connection is to a DB2 UDB for Linux, UNIX, and Windows server, caching of dynamic statements for a connection is done.

Dynamic statement caching can be done only if the EDM dynamic statement cache is enabled on the database server. The CACHEDYN subsystem parameter must be set to YES to enable the dynamic statement cache.

keepDynamic is used with the DB2Binder -keepdynamic option. The keepDynamic property value that is specified must match the -keepdynamic value that was specified when DB2Binder was run.

See *DB2 Application Programming and SQL Guide* for more information on dynamic statement caching.

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 Universal Driver type 4 connectivity.

loginTimeout
The maximum time in seconds to wait for a connection to a data source, or for SQL requests to that 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 Universal Driver type 2 connectivity on DB2 UDB in the z/OS or OS/390 environment.

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.

maxTransportObjects
Specifies the maximum number of transport objects that can be used for all connections with the associated `DataSource` object. Transport objects are used for the connection concentrator and Sysplex workload balancing. 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.

The data type of this property is int.

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);
```

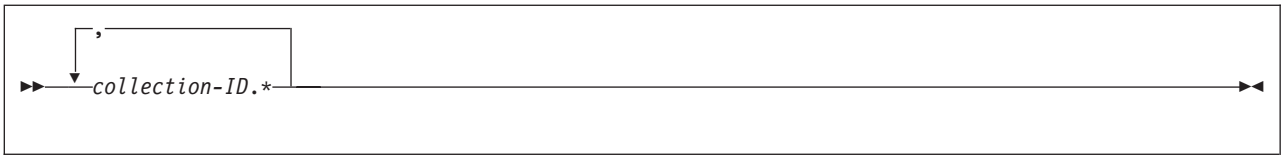
pkList
Specifies a package list that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established.

pkList is applicable only to Universal Driver type 2 connectivity.

Specify this property if you do not bind plans for your SQLJ programs or for the JDBC driver. If you specify this property, **do not specify planName.**

Recommendation: Use pkList instead of planName.

The format of the package list is:



pkList overrides the value of the db2.jcc.pkList configuration property. If pkList, planName, and db2.jcc.pkList are not specified, the value of pkList is NULLID.*.

planName
Specifies a DB2 plan name that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established.

Specify this property if you bind plans for your SQLJ programs and for the JDBC driver packages. If you specify this property, **do not specify pkList.**

planName is applicable only to Universal Driver type 2 connectivity.

planName overrides the value of the db2.jcc.planName configuration property. If pkList, planName, and db2.jcc.planName are not specified, NULLID.* is used as the package list for the underlying CREATE THREAD call.

portNumber
The port number where the DRDA® server is listening for requests. The data type of this property is int.

This property is applicable only to Universal Driver type 4 connectivity.

queryCloseImplicit
Specifies whether cursors are closed immediately after all rows are fetched. queryCloseImplicit applies only to Universal Driver type 4 connectivity to DB2 UDB 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 DB2 Universal JDBC Driver to invoke a DB2 UDB for OS/390 or 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.

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.

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 DB2 Universal JDBC Driver 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.

This property is applicable only to Universal Driver type 4 connectivity.

sendDataAsIs
Specifies that the DB2 Universal JDBC Driver 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.

This property is applicable only to Universal Driver type 4 connectivity.

supportsAsynchronousXARollback
Specifies whether the DB2 Universal JDBC Driver 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).

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 DB2 Universal JDBC Driver 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.

For Universal Driver type 2 connectivity, the db2.jcc.override.traceFile configuration property value overrides the `traceFile` property value.

Recommendation: Set the db2.jcc.override.traceFile configuration property, rather than setting the traceFile property for individual connections.

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_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
  ```

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 host 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);
```

DataSource properties for the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS

A `DB2DataSource` or `DB2SimpleDataSource` class provides a set of properties that define how the connection to a particular data source should be made. Those properties are usually set when a `DataSource` object is created and deployed. Those properties are:

databaseName
Specifies the location name to be used when establishing connections using the `DataSource` object. If the location name is not the local site (see the description of the DB2SQLJSSID property in "The SQLJ/JDBC run-time properties file" on page 281), the location name must be defined in SYSIBM.LOCATIONS. If the location name is the local site, the location name must have been specified in field DB2 LOCATION NAME of the DISTRIBUTED DATA FACILITY panel during the DB2 installation. If you do not set the `databaseName` property, connections that are established using this data source object are to the local site. This property has data type String. The default value is null.

description
Describes the data source object. This property has data type String. The default value is null.

user
Specifies the z/OS user ID to be used when using the `DataSource` object to establish a connection to the data source. DB2 validates the user ID and password. You can override this property by calling the `DataSource.getConnection` method with the *user* parameter. If you set the `user` property, or specify *user* parameter in the `DataSource.getConnection` method call, you must also set the `password` property, or specify the *password* parameter in the `DataSource.getConnection` method call. This property has data type String.

password
Specifies a corresponding password for the `user` property. You can override this property by calling the `DataSource.getConnection` method with the *password* parameter. This property has data type String. The default value is null.

planName
Specifies the name of the plan that DB2 allocates for connections that are established using the data source object. This property has data type String. The default value is DSNJDBC.

loginTimeout
Specifies the maximum time in seconds to wait for the `DataSource` object to connect to a data source. A value of 0 means that the timeout value is the default system timeout value, which is specified by the `db2.connpool.connect.create.timeout` property in the db2sqljjdbc.properties file. This property has data type int. The default value is 0.

Table 47 on page 197 lists the methods that you use to set and retrieve the property values.

Table 47. getXXX and setXXX methods for DataSource properties under the JDBC/SQLJ 2.0 Driver for OS/390 and z/OS

Property	getXXX method	setXXX method
databaseName	String getDatabaseName()	void setDatabaseName(String *location-name*)
description	String getDescription()	void setDescription(String *description*)
loginTimeout	int getLoginTimeout()	void setLoginTimeout(int *timeout*)
password	None	void setPassword(String *password*)
planName	String getPlanName()	void setPlanName(String *plan-name*)
user	String getUser()	void setUser(String *user-name*)

Chapter 5. Creating Java stored procedures and user-defined functions

Stored procedures and user-defined functions are programs that can contain SQL statements. You invoke a stored procedure from a client program by executing the SQL CALL statement. You invoke a user-defined function by specifying the user-defined function name, followed by its arguments, in an SQL statement. This topic contains information that is specific to defining and writing Java user-defined functions and stored procedures. For general information on stored procedures, see Part 6 of *DB2 Application Programming and SQL Guide*. For general information on user-defined functions, see Part 3 of *DB2 Application Programming and SQL Guide*. For information on preparing Java stored procedures or user-defined functions for execution, see "Preparing Java routines for execution" on page 245.

In this topic, the following terminology is used:

- The word *routine* refers to either a stored procedure or a user-defined function.
- The term *interpreted Java stored routine* refers to a stored procedure or a user-defined function that runs in a JVM.

The following topics provide additional information:

- "Setting up the environment for interpreted Java routines"
- "Defining a Java routine to DB2" on page 206
- "Defining a JAR file for a Java routine to DB2" on page 210
- "Writing a Java routine" on page 214
- "Testing a Java routine" on page 218

Setting up the environment for interpreted Java routines

This topic discusses the setup tasks for preparing and running interpreted Java routines. If you plan to use DB2 Development Center to prepare and run your interpreted Java routines, see the following URL for complete instructions:

`http://www.ibm.com/software/db2zos/sqlproc`

To set up the environment for running interpreted Java routines, you need to perform these tasks:

1. Ensure that your operating system and the Java SDK are at the correct levels, and that you have installed all prerequisite products. See "Prerequisites for interpreted Java routines" for a list of requirements.
2. Install DB2 UDB for z/OS Java support. See Chapter 7, "Installing the DB2 Universal JDBC Driver," on page 251.
3. Create the Workload Manager for z/OS (WLM) application environment for running the routines. See "Setting up the WLM application environment for interpreted Java routines" on page 200.
4. Set environment variables that are required by Java routines. See "Setting the run-time environment for interpreted Java stored procedures" on page 202.

Prerequisites for interpreted Java routines

In addition to DB2 UDB for z/OS with Java support, you need to install the following products for interpreted Java stored procedures:

- z/OS with z/OS UNIX System Services, WLM, and RRS

- IBM Developer Kit for z/OS, Java 2 Technology Edition, SDK 1.3.1 level, SDK 1.4.1 level or later

Setting up the WLM application environment for interpreted Java routines

To set up WLM application environments for stored procedures or user-defined functions, you need to define a JCL startup procedure for each WLM environment, and define the application environment to WLM. You need different WLM application environments for interpreted Java routines from the WLM application environments you use for other routines.

Creating the WLM address space startup procedure

The address space startup procedure for Java routines requires extra DD statements that other routines do not need. Figure 63 shows an example of a startup procedure for an address space in which Java routines can run. The JAVAENV DD statement indicates to DB2 that the WLM environment is for Java routines.

```
//DSNWLM PROC RGN=0K,APPLENV=WLMIJAV,DB2SSN=DSN,NUMTCB=5   1
//IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
// PARM='&DB2SSN,&NUMTCB,&APPLENV'

//STEPLIB DD DISP=SHR,DSN=DSN810.RUNLIB.LOAD
// DD DISP=SHR,DSN=CEE.SCEERUN
// DD DISP=SHR,DSN=DSN810.SDSNEXIT
// DD DISP=SHR,DSN=DSN810.SDSNLOAD
// DD DISP=SHR,DSN=DSN810.SDSNLOD2
//JAVAENV DD DISP=SHR,DSN=WLMIJAV.JSPENV                   2
//JSPDEBUG DD SYSOUT=A                                     3
//CEEDUMP DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
```

Figure 63. Startup procedure for a WLM address space in which an interpreted Java routine runs

Notes to Figure 63:

1. In this line, change the DB2SSN value to your DB2 UDB for z/OS subsystem name. Change the APPLENV value to the name of the application environment that you set up for Java stored procedures. The maximum value of NUMTCB should be between 5 and 8. For testing a Java stored procedure, NUMTCB=1 is recommended. With NUMTCB=1, only one JVM is started, so refreshing the WLM environment after you change the stored procedure takes less time.
2. JAVAENV specifies a data set that contains Language Environment® run-time options for Java stored procedures. The presence of this DD statement indicates to DB2 that the WLM environment is for Java routines. For an interpreted Java routine, this data set must contain the environment variable JAVA_HOME. This environment variable indicates to DB2 that the WLM environment is for interpreted Java routines. JAVA_HOME also specifies the highest-level directory in the set of directories that containing the Java SDK.
3. Specifies a data set into which DB2 puts information that you can use to debug your stored procedure. The information that DB2 collects can be very helpful in debugging setup problems, and also contains key information that you need to provide when you submit a problem to IBM Service. You should comment out this DD statement during production.

Defining the WLM application environment

To define the application environment to WLM, specify the values shown on the following WLM panels.

```
  File  Utilities  Notes  Options  Help
------------------------------------------------------------------------------
                    Definition Menu     WLM Appl
Command ===> ____________________________________________________________

Definition data set . :  none
Definition name  . . . . WLMENV
Description . . . . . .  Environment for Development Center
Select one of the
following options. . .   9  1. Policies
                            2. Workloads
                            3. Resource Groups
                            4. Service Classes
                            5. Classification Groups
                            6. Classification Rules
                            7. Report Classes
                            8. Service Coefficients/Options
                            9. Application Environments
                           10. Scheduling Environments
```

Definition name
Specify the name of the WLM application environment that you are setting up for stored procedures.

Description
Specify any value.

Options
Specify 9 (Application Environments).

```
  Application-Environment  Notes  Options  Help
------------------------------------------------------------------------------
                    Create an Application Environment
Command ===> ____________________________________________________________

Application Environment Name . : WLMENV
Description  . . . . . . . . . . Environment for Development Center
Subsystem Type . . . . . . . . . DB2
Procedure Name . . . . . . . . . DSN8WLMP
Start Parameters . . . . . . . . DB2SSN=DB2T,NUMTCB=3,APPLENV=WLMENV
                                 ____________________________________
                                 ___________________________________

Limit on starting server address spaces for a subsystem instance:
1   1.  No limit.
    2.  Single address space per system.
    3.  Single address spaces per sysplex.
```

Subsystem Type
Specify DB2.

Procedure Name
This name must match the name of the JCL startup procedure for the stored procedure address spaces that are associated with this application environment.

Start Parameters
If the DB2 subsystem in which the stored procedure runs is not in a Sysplex, the DB2SSN value must match the name of that DB2 subsystem. If the same JCL is used for multiple DB2 subsystems, specify DB2SSN=&IWMSSNM.

The NUMTCB value depends on the type of stored procedure you are running. For running Java routines, the maximum value that you specify should be between 5 and 8.

The APPLENV value must match the value that you specify on the CREATE PROCEDURE or CREATE FUNCTION statement for the routines that run in this application environment.

Limit on starting server address spaces for a subsystem instance
Specify 1 (no limit).

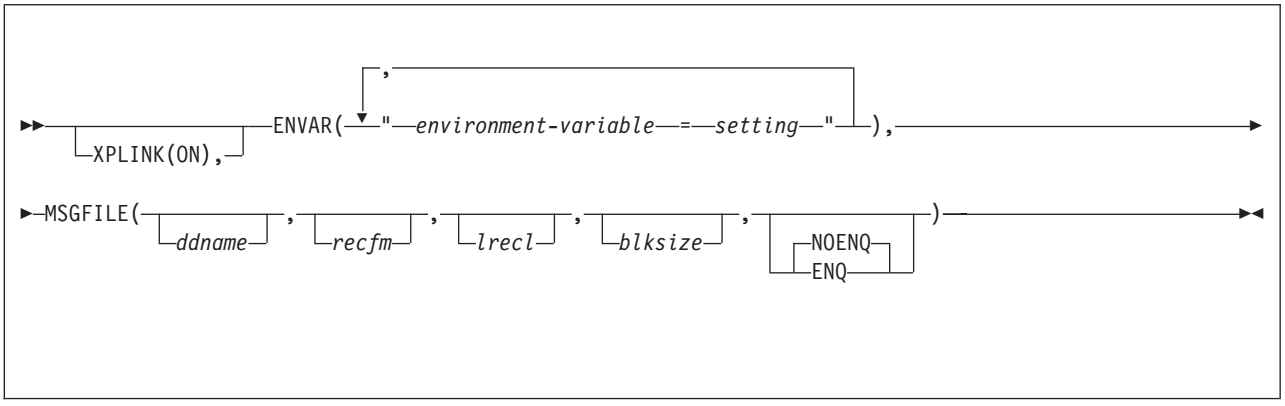
Setting the run-time environment for interpreted Java stored procedures

For Java routines, the startup procedure for the stored procedure address space contains a JAVAENV DD statement. This statement specifies a data set that contains Language Environment run-time options for the routines that run in the stored procedure address space. Create the data set with the characteristics that are listed in Table 48.

Table 48. Data set characteristics for the JAVAENV data set

Primary space allocation	1 block
Secondary space allocation	1 block
Record format	VB
Record length	255
Block size	4096

After you create the data set, edit it to insert a Language Environment options string, which has this form:



The maximum length of the Language Environment run-time options string in a JAVAENV data set for interpreted Java stored procedures is 245 bytes. If you exceed the maximum length, DB2 truncates the contents but does not issue a message. If you enter the contents of the JAVAENV data set on more than one line, DB2 concatenates the lines to form the run-time options string. The run-time options string can contain no leading or trailing blanks. Within the string, only blanks that are valid within an option are permitted.

If your environment variable list is long enough that the JAVAENV content is greater than 245 bytes, you can put the environment variable list in a separate data set in a separate file, and use the environment variable _CEE_ENVFILE to point to that file.

Parameter descriptions:

XPLINK(ON)
Causes the initialization of the XPLINK environment. This parameter is **required** for Java 2 Technology Edition, SDK 1.4.1. This parameter must **not** be specified for Java 2 Technology Edition, SDK 1.3.1.

ENVAR
Sets the initial values for specified environment variables. The environment variables that you might need to specify are:

CLASSPATH
When you prepare your Java routines, if you do not put your routine classes into JAR files, include the directories that contain those classes. For example:

```
CLASSPATH=.:/U/DB2RES3/ACMEJOS
```

DB2_HOME or JCC_HOME
The value of DB2_HOME or JCC_HOME is the highest-level directory in the set of directories that contain the JDBC driver. Specify only one of these environment variables. Use DB2_HOME if your Java routines run under the JDBC/SQLJ Driver for OS/390 and z/OS. Use JCC_HOME if your Java routines run under the DB2 Universal JDBC Driver. For example:

```
JCC_HOME=/usr/lpp/db2810
```

JAVA_HOME
This environment variable indicates to DB2 that the WLM environment is for interpreted Java routines. The value of JAVA_HOME is the highest-level directory in the set of directories that contain the Java SDK. For example:

```
JAVA_HOME=/usr/lpp/java/IBM/J1.3
```

JVMPROPS
This environment variable specifies the name of a z/OS UNIX System Services file that contains startup options for the JVM in which the stored procedure runs. For example:

```
JVMPROPS=/usr/lpp/java/properties/jvmsp
```

The following example shows the contents of a startup options file that you might use for a JVM in which Java stored procedures run:

```
# Properties file for JVM for Java stored procedures
# Sets the initial size of middleware heap within non-system heap
-Xms64M

# Sets the maximum size of nonsystem heap
-Xmx128M

#initial size of application class system heap
-Xinitacsh512K

#initial size of system heap
-Xinitsh512K

#initial size of transient heap
-Xinitth32M
```

For information about JVM startup options, see *Persistent Reusable Java Virtual Machine User's Guide*, available at:

```
http://www.ibm.com/servers/eserver/zseries/software/java
```

Click the Reference Information link.

LC_ALL
Modify LC_ALL to change the locale to use for the locale categories when the individual locale environment variables specify locale information. This value needs to match the CCSID for the DB2 subsystem on which the stored procedures run. For example:

```
LC_ALL=En_US.IBM-037
```

RESET_FREQ
Specifies the frequency of JVM reset operations and indicates whether the JVM is run in resettable mode.

The reset frequency and resettable mode depend on the RESET_FREQ value:

>0 The JVM is run in resettable mode. A reset operation is performed after the number of stored procedure invocations that is specified by the RESET_FREQ value.

For example, RESET_FREQ=3 indicates that the JVM is reset after three stored procedure invocations.

=0 The JVM is run in resettable mode, with a reset frequency of 256 stored procedure invocations. This is the default.

<0 The JVM is not run in resettable mode.

Non-resettable mode is supported only with the DB2 Universal JDBC Driver. If RESET_FREQ is less than zero and JCC_HOME is not specified in the JAVAENV data set, the JVM does not start, and an error is generated.

When the JVM runs in resettable mode, a garbage collection request is made after every ten resets. In non-resettable mode, no garbage collection request is made. If garbage collection is necessary when the JVM is in non-resettable mode, request garbage collection by specifying the `-Xgcpolicy` JVM option in the JVMPROPS environment variable.

Running the JVM In resettable mode protects future users of a Java routine from corruption that is caused by the current user of the routine. Run the JVM in non-resettable mode only if you know that a routine does not corrupt the JVM.

For information about the resettable JVM and garbage collection, see "Using static and non-final variables in a Java routine" on page 215 and *Persistent Reusable Java Virtual Machine User's Guide*.

TMSUFFIX
Specifies a list of directories and JAR files that contain classes that are to be included in the trusted middleware classes for the JVM that is used to execute the routine. The list is in the same format as a CLASSPATH list. Specify TMSUFFIX under either of the following circumstances:

- When a class needs control over its static members, and those members cannot be re-initialized when the JVM is reset. In this case, you can define a tidy-up method that is executed each time the JVM is reset.
- When the following conditions are true:
  - Java routines that use certain classes fail with an SQLSTATE of 38503 and an error code -430.
  - The associated DSNX961I console message indicates that the JVM cannot be reset.

- The action that prevents the JVM from being reset cannot be avoided. The only way to be able to reset the JVM is to designate the classes that contain the needed methods as trusted middleware.

If the value that is specified for RESET_FREQ is less than zero, any value that is specified for TMSUFFIX is appended to the effective CLASSPATH. The TMSUFFIX value follows the specified CLASSPATH and the JARs for the JDBC driver. Any class that is specified for TMSUFFIX is treated as a normal class. Its tidy-up method is not automatically invoked because the JVM is not reset.

For information about trusted middleware, static data, tidy-up methods, and when a JVM cannot be reset, see *Persistent Reusable Java Virtual Machine User's Guide*.

TZ

Modify TZ to change the local timezone. For example:

```
TZ=PST08
```

The default is GMT.

_CEE_ENVFILE

Specifies a z/OS UNIX System Services data set that contains some or all of the settings for environment variables.

Use the _CEE_ENVFILE parameter if the length of environment variable string causes the total length of the JAVAENV content to exceed 245 bytes, which is the DB2 limit for the JAVAENV content.

The data set must be variable-length.The format for environment variable settings in this data set is:

```
environment-variable-1=setting-1
environment-variable-2=setting-2
...
environment-variable-n=setting-n
```

You can specify some of your environment variable settings as arguments of ENVAR and put some of the settings in this data set, or you can put all of your environment variable settings in this data set.

For example, to use file /u/db281/javasp/jspnolimit.txt for environment variable settings, specify:

```
_CEE_ENVFILE=/u/db281/javasp/jspnolimit.txt
```

USE_LIBJVM_G

Specifies whether the debug version of the JVM is used instead of the default, non-debug version of the JVM. The debug version of the JVM is in dynamic link library libjvm_g. If USE_LIBJVM_G is not specified, or its value is anything other than the capitalized string YES, the non-debug version of the JVM is used. For example, USE_LIBJVM_G=NO causes the non-debug version of the JVM to be used.

If USE_LIBJVM_G=YES, the JVMPROPS environment variable must specify a file that contains JVM startup options. That file must contain the startup option -Djava.execsuffix=_g.

Specify USE_LIBJVM_G=YES only under the direction of IBM Software Support.

MSGFILE
Specifies the DD name of a data set in which Language Environment puts run-time diagnostics. All subparameters in the MSGFILE parameter are optional. The default is

```
MSGFILE(SYSOUT,FBA,121,0,NOENQ)
```

If you specify a data set name in the JSPDEBUG statement of your stored procedure address space startup procedure, you need to specify JSPDEBUG as the first parameter. If the NUMTCB value in the stored procedure address space startup procedure is greater than 1, you need to specify ENQ as the fifth subparameter. *z/OS Language Environment Programming Reference* contains complete information about MSGFILE.

The following example shows the contents of a JAVAENV data set.

```
ENVAR("JCC_HOME=/usr/lpp/db2810",
"JAVA_HOME=/usr/lpp/javas13/J1.3",
"WORK_DIR=/u/db281/tmp"),
MSGFILE(JSPDEBUG)
```

For information on environment variables that are related to locales, see *z/OS C/C++ Programming Guide*.

Defining a Java routine to DB2

Defining a Java routine to DB2 involves one or two steps, depending on where the routine resides:

- For interpreted Java routines that you store in JAR files, you need to define the JAR files to DB2.

  If you prepare the Java routine for execution without DB2 Development Center, use the SQLJ.INSTALL_JAR built-in stored procedure to define the JAR files to DB2. To replace or delete the JAR file, use the SQLJ.REPLACE_JAR or SQLJ.REMOVE_JAR stored procedure. These stored procedures are discussed in detail in "Defining a JAR file for a Java routine to DB2" on page 210.
- For all types of Java routines, you need to define the routine to DB2.

  If you prepare the Java routine for execution without DB2 Development Center, execute the CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. To alter the routine definition, use the ALTER PROCEDURE or ALTER FUNCTION statement. For information on these statements, see Chapter 5 of *DB2 SQL Reference*.

If you use the DB2 Development Center to prepare your Java stored procedures for execution, the DB2 Development Center defines the Java routine and the JAR file to DB2 for you.

The definition for a Java routine is much like the definition for a routine in any other language. However, the following parameters have different meanings for Java routines.

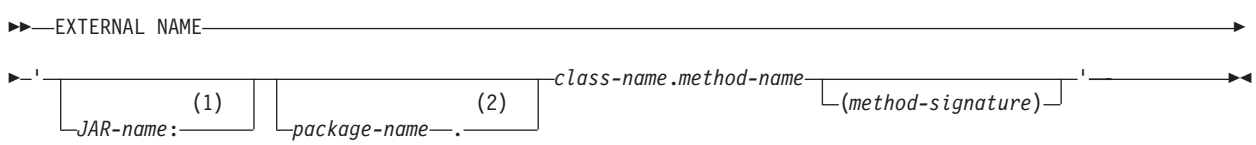
LANGUAGE
Specifies the application programming language in which the routine is written.

Specify LANGUAGE JAVA.

You cannot specify LANGUAGE JAVA for a user-defined table function.

EXTERNAL NAME

Specifies the program that runs when the procedure name is specified in a CALL statement or the user-defined function name is specified in an SQL statement. For Java routines, the argument of EXTERNAL NAME is a string that is enclosed in single quotation marks. The EXTERNAL NAME clause for a Java routine has the following syntax:



Notes:

1 For compatibility with DB2 UDB for Linux, UNIX and Windows, you can use an exclamation point (!) after *JAR-name* instead of a colon.

2 For compatibility with previous versions of DB2, you can use a slash (/) after *package-name* instead of a period.

Whether you include *JAR-name* depends on where the Java code for the routine resides. If you create a JAR file from the class file for the routine (the output from the javac command), you need to include *JAR-name*. You must create the JAR file and define the JAR file to DB2 before you execute the CREATE PROCEDURE or CREATE FUNCTION statement. If some other user executes the CREATE PROCEDURE or CREATE FUNCTION statement, you need to grant the USAGE privilege on the JAR to them.

If you use a JAR file, that JAR file must be self-contained. That is, if a class within the JAR file references another class, the referenced class must be also be in the JAR file. The exception to this rule is that classes that are in directories that are referenced in CLASSPATH, DB2_HOME or JCC_HOME,, and JAVA_HOME do not need to be included in the JAR file.

Whether you include *(method-signature)* depends on the following factors:

- The way that you define the parameters in your routine method

  Each SQL data type has a corresponding default Java data type. If your routine method uses data types other than the default types, you need to include a method signature in the EXTERNAL NAME clause. A method signature is a comma-separated list of data types.
- Whether you overload a Java routine

  If you have several Java methods in the same class, with the same name and different parameter types, you need to specify the method signature to indicate which version of the program is associated with the Java routine.

If your stored procedure returns result sets, you also need to include a parameter in the method signature for each result set. The parameter can be in one of the following forms:

- `java.sql.ResultSet[]`
- An array of an SQLJ iterator class

You do *not* include these parameters in the parameter list of the SQL CALL statement when you invoke the stored procedure.

Table 41 on page 131 shows the SQL data types that you can specify in the parameter definition and the corresponding Java data types that you can specify in the method signature.

Example: EXTERNAL NAME clause for a Java user-defined function: Suppose that you write a Java user-defined function as method getSals in class S1Sal and package s1. You put S1Sal in a JAR file named sal_JAR and install that JAR in DB2. The EXTERNAL NAME parameter is :

```
EXTERNAL NAME 'sal_JAR:s1.S1Sal.getSals'
```

Example: EXTERNAL NAME clause for a Java stored procedure: Suppose that you write a Java stored procedure as method getSals in class S1Sal. You put S1Sal in a JAR file named sal_JAR and install that JAR in DB2. The stored procedure has one input parameter of type INTEGER and returns one result set. The Java method for the stored procedure receives one parameter of type java.lang.Integer, but the default Java data type for an SQL type of INTEGER is int, so the EXTERNAL NAME clause requires a signature clause. The EXTERNAL NAME parameter is :

```
EXTERNAL NAME 'sal_JAR:S1Sal.getSals(java.lang.Integer,java.sql.ResultSet[])'
```

NO SQL
Indicates that the routine does not contain any SQL statements.

For a Java routine that is stored in a JAR file, you cannot specify NO SQL.

PARAMETER STYLE
Identifies the linkage convention that is used to pass parameters to the routine.

For a Java routine, the only value that is valid is PARAMETER STYLE JAVA.

You cannot specify PARAMETER STYLE JAVA for a user-defined table function.

WLM ENVIRONMENT
Identifies the MVS workload manager (WLM) environment in which the routine is to run when the DB2 stored procedure address space is WLM-established.

If you do not specify this parameter, the routine runs in the default WLM environment that was specified when DB2 was installed.

PROGRAM TYPE
Specifies whether Language Environment runs the routine as a main routine or a subroutine.

This parameter value must be PROGRAM TYPE SUB.

RUN OPTIONS
Specifies the Language Environment run-time options to be used for the routine.

This parameter has no meaning for a Java routine. If you specify this parameter with LANGUAGE JAVA, DB2 issues an error.

SCRATCHPAD
Specifies that when the user-defined function is invoked for the first time, DB2 allocates memory for a scratchpad.

You cannot use a scratchpad in a Java user-defined function. Do not specify SCRATCHPAD when you create or alter a Java user-defined function.

FINAL CALL
Specifies that a final call is made to the user-defined function, which the function can use to free any system resources that it has acquired.

You cannot perform a final call when you call a Java user-defined function. Do not specify FINAL CALL when you create or alter a Java user-defined function.

DBINFO
Specifies that when the routine is invoked, an additional argument is passed that contains environment information.

You cannot pass the additional argument when you call a Java routine. Do not specify DBINFO when you call a Java routine.

SECURITY
Specifies how the routine interacts with an external security product, such as RACF, to control access to non-SQL resources. The values of the SECURITY parameter are the same for a Java routine as for any other routine. However, the value of the SECURITY parameter determines the authorization ID that must have authority to access z/OS UNIX System Services. The values of SECURITY and the IDs that must have access to z/OS UNIX System Services are:

DB2 The user ID that is defined for the stored procedure address space in the RACF started-procedure table.

EXTERNAL
The invoker of the routine.

DEFINER
The definer of the routine.

For a complete explanation of the parameters in a CREATE PROCEDURE, CREATE FUNCTION, ALTER PROCEDURE or ALTER FUNCTION statement, see Chapter 5 of *DB2 SQL Reference*.

Example: Defining a Java stored procedure: Suppose that you have written and prepared a stored procedure that has these characteristics:

Fully-qualified procedure name	SYSPROC.S1SAL
Parameters	DECIMAL(10,2) INOUT
Language	Java
Collection ID for the stored procedure package	DSNJDBC
Package, class, and method name	s1.S1Sal.getSals
Type of SQL statements in the program	Statements that modify DB2 tables
WLM environment name	WLMIJAV
Maximum number of result sets returned	1

This CREATE PROCEDURE statement defines the stored procedure to DB2:

```
CREATE PROCEDURE SYSPROC.S1SAL
 (DECIMAL(10,2) INOUT)
  FENCED
  MODIFIES SQL DATA
  COLLID DSNJDBC
  LANGUAGE JAVA
  EXTERNAL NAME  's1.S1Sal.getSals'
```

```
  WLM ENVIRONMENT WLMIJAV
  DYNAMIC RESULT SETS 1
  PROGRAM TYPE SUB
  PARAMETER STYLE JAVA;
```

Example: Defining a Java user-defined function: Suppose that you have written and prepared a user-defined function that has these characteristics:

Fully-qualified function name	MYSCHEMA.S2SAL
Input parameter	INTEGER
Data type of returned value	VARCHAR(20)
Language	Java
Collection ID for the function package	DSNJDBC
Package, class, and method name	s2.S2Sal.getSals
Java data type of the method input parameter	java.lang.Integer
JAR file that contains the function class	sal_JAR
Type of SQL statements in the program	Statements that modify DB2 tables
Function is called when input parameter is null?	Yes
WLM environment name	WLMIJAV

This CREATE FUNCTION statement defines the user-defined function to DB2:

```
CREATE FUNCTION MYSCHEMA.S2SAL(INTEGER)
  RETURNS VARCHAR(20)
  FENCED
  MODIFIES SQL DATA
  COLLID DSNJDBC
  LANGUAGE JAVA
  EXTERNAL NAME  'sal_JAR:s2.S2Sal.getSals(java.lang.Integer)'
  WLM ENVIRONMENT WLMIJAV
  CALLED ON NULL INPUT
  PROGRAM TYPE SUB
  PARAMETER STYLE JAVA;
```

In this function definition, you need to specify a method signature in the EXTERNAL NAME clause because the data type of the method input parameter is different from the default Java data type for an SQL type of INTEGER.

Defining a JAR file for a Java routine to DB2

One way to organize the classes for a Java routine is to collect those classes into a JAR file, as described in "Creating JAR files for Java routines" on page 247. If you do this, you need to install the JAR file into the DB2 catalog. DB2 provides five built-in stored procedures that perform the following functions for the JAR file:

SQLJ.INSTALL_JAR
Installs a JAR file into the local DB2 catalog.

SQLJ.REPLACE_JAR
Replaces an existing JAR file in the local DB2 catalog.

SQLJ.REMOVE_JAR
Deletes a JAR file from the local DB2 catalog or a remote DB2 catalog.

SQLJ.DB2_INSTALL_JAR
Installs a JAR file into the local DB2 catalog or a remote DB2 catalog.

SQLJ.DB2_REPLACE_JAR
Replaces an existing JAR file in the local DB2 catalog or a remote DB2 catalog.

You can use the DB2 Development Center to install JAR files into the DB2 catalog, or you can write a client program that executes SQL CALL statements to invoke the stored procedures. The following information describes how to call the stored procedures.

Calling SQLJ.INSTALL_JAR

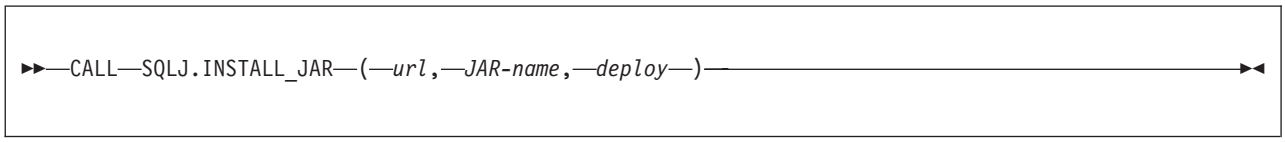
Use SQLJ.INSTALL_JAR to create a new definition of a JAR file in the local DB2 catalog.

SQLJ.INSTALL_JAR authorization

To call SQLJ.INSTALL_JAR, you need the following privileges:

- The EXECUTE privilege on SQLJ.INSTALL_JAR.
- If the SQL authorization ID of the process under which SQLJ.INSTALL_JAR is invoked is not the same as the schema for the JAR, you need one of the following authorizations:
  - SYSADM or SYSCTRL authority
  - The CREATEIN privilege on the designated schema for the JAR.

SQLJ.INSTALL_JAR syntax



SQLJ.INSTALL_JAR parameters

url A VARCHAR(1024) input parameter that identifies the z/OS UNIX System Services full path name for the JAR file that is to be installed in the DB2 catalog. The format is `file://`*path-name* or `file:/`*path-name*.

JAR-name
A VARCHAR(257) input parameter that contains 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 SQLID special register.

deploy
An INTEGER input parameter that indicates whether additional actions should be performed after the JAR file is installed. Additional actions are not supported, so this value should always be 0.

Calling SQLJ.REPLACE_JAR

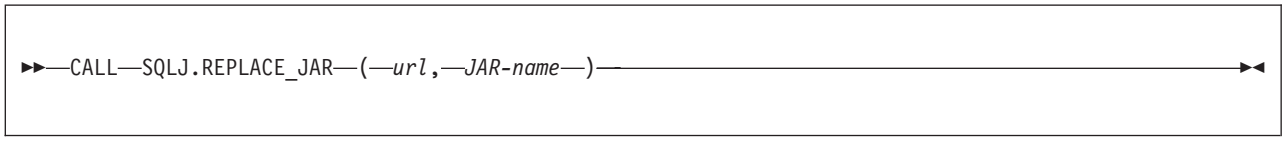
Use SQLJ.REPLACE_JAR to replace an existing JAR file in the local DB2 catalog.

SQLJ.REPLACE_JAR authorization

To call SQLJ.REPLACE_JAR, you need the following privileges:

- The EXECUTE privilege on SQLJ.REPLACE_JAR.
- If the SQL authorization ID of the process under which SQLJ.REPLACE_JAR is invoked is not the same as the schema for the JAR, you need one of the following authorizations:
  - SYSADM or SYSCTRL authority
  - The DROPIN and CREATEIN privileges on the designated schema for the JAR.

SQLJ.REPLACE_JAR syntax



SQLJ.REPLACE_JAR parameters

url A VARCHAR(1024) input parameter that identifies the z/OS UNIX System Services full path name for the JAR file that replaces the existing JAR file in the DB2 catalog. The format is `file://`*path-name* or `file:/`*path-name*.

JAR-name
A VARCHAR(257) input parameter that contains 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 SQLID special register.

Calling SQLJ.REMOVE_JAR

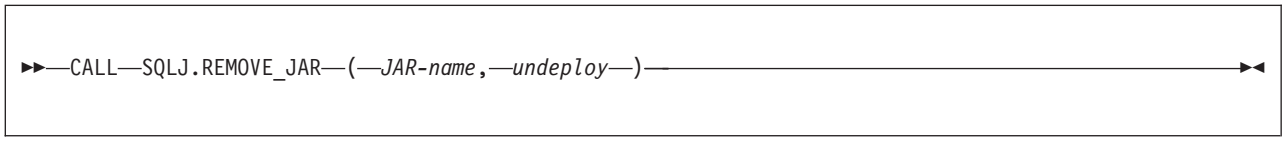
Use SQLJ.REMOVE_JAR to delete a JAR file from the local DB2 catalog or a remote DB2 catalog. To delete a JAR file at a remote location, you need to execute a CONNECT statement to connect to that location before you call SQLJ.REMOVE_JAR.

SQLJ.REMOVE_JAR authorization

To call SQLJ.REMOVE_JAR, you need the following privileges:

- The EXECUTE privilege on SQLJ.REMOVE_JAR.
- If the SQL authorization ID of the process under which SQLJ.REMOVE_JAR is invoked is not the same as the schema for the JAR, you need one of the following authorizations:
  - SYSADM or SYSCTRL authority
  - The DROPIN privilege on the designated schema for the JAR.

SQLJ.REMOVE_JAR syntax



SQLJ.REMOVE_JAR parameters

JAR-name
A VARCHAR(257) input parameter that contains the DB2 name of the JAR that is to be removed from the catalog, 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 SQLID special register.

undeploy
An INTEGER input parameter that indicates whether additional actions should be performed before the JAR file is removed. Additional actions are not supported, so this value should always be 0.

Calling SQLJ.DB2_INSTALL_JAR

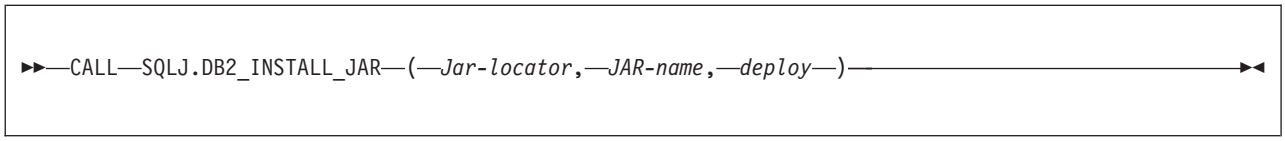
Use SQLJ.DB2_INSTALL_JAR to create a new definition of a JAR file in the local DB2 catalog or a remote DB2 catalog. To install a JAR file at a remote location, you need to execute a CONNECT statement to connect to that location before you call SQLJ.DB2_INSTALL_JAR.

SQLJ.DB2_INSTALL_JAR authorization

To call SQLJ.DB2_INSTALL_JAR, you need the following privileges:

- The EXECUTE privilege on SQLJ.DB2_INSTALL_JAR.
- If the SQL authorization ID of the process under which SQLJ.DB2_INSTALL_JAR is invoked is not the same as the schema for the JAR, you need one of the following authorizations:
  - SYSADM or SYSCTRL authority
  - The CREATEIN privilege on the designated schema for the JAR.

SQLJ.DB2_INSTALL_JAR syntax



SQLJ.DB2_INSTALL_JAR parameters

JAR-locator
A BLOB locator input parameter that points to the JAR file that is to be installed in the DB2 catalog.

JAR-name
A VARCHAR(257) input parameter that contains 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 SQLID special register.

deploy
An INTEGER input parameter that indicates whether additional actions should be performed after the JAR file is installed. Additional actions are not supported, so this value should always be 0.

Calling SQLJ.DB2_REPLACE_JAR

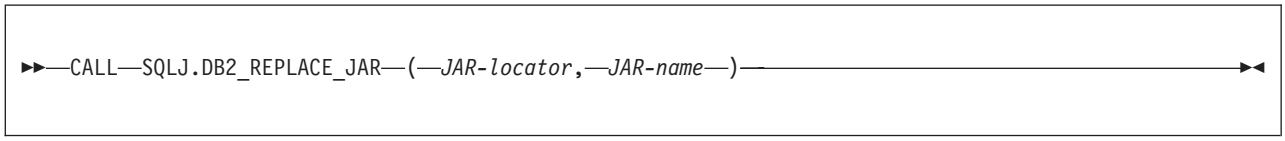
Use SQLJ.DB2_REPLACE_JAR to replace an existing JAR file in the local DB2 catalog or in a remote DB2 catalog. To replace a JAR file at a remote location, you need to execute a CONNECT statement to connect to that location before you call SQLJ.DB2_REPLACE_JAR.

SQLJ.DB2_REPLACE_JAR authorization

To call SQLJ.DB2_REPLACE_JAR, you need the following privileges:

- The EXECUTE privilege on SQLJ.REPLACE_JAR.
- If the SQL authorization ID of the process under which SQLJ.DB2_REPLACE_JAR is invoked is not the same as the schema for the JAR, you need one of the following authorizations:
  - SYSADM or SYSCTRL authority
  - The DROPIN and CREATEIN privileges on the designated schema for the JAR.

SQLJ.DB2_REPLACE_JAR syntax



SQLJ.DB2_REPLACE_JAR parameters

JAR-locator
A BLOB locator input parameter that points to the JAR file that is to be replaced in the DB2 catalog.

JAR-name
A VARCHAR(257) input parameter that contains 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 SQLID special register.

Writing a Java routine

A Java routine is a Java application program that runs in a stored procedure address space. It can include JDBC methods or SQLJ clauses. A Java routine is much like any other Java program and follows the same rules as routines in other languages. It receives input parameters, executes Java statements, optionally executes SQLJ clauses, JDBC methods, or a combination of both, and returns output parameters.

Differences between Java routines and stand-alone Java programs

A Java routine differs from a stand-alone Java program in the following ways:

- In a Java routine, a JDBC connection or an SQLJ connection context can use the connection to the data source that processes the CALL statement or the user-defined function invocation. The URL that identifies this default connection is jdbc:default:connection.
- The top-level method for a Java routine must be declared as static and public.

  Although you can use static and final variables in a Java routine without problems, you might encounter problems when you use static and non-final variables. You cannot guarantee that a static and non-final variable retains its value in the following circumstances:
  - Across multiple invocations of the same routine
  - Across invocations of different routines that reference that variable

  See "Using static and non-final variables in a Java routine" on page 215 for more information on how to use static and non-final variables.
- As in routines in other languages, the SQL statements that you can execute in the routine depend on whether you specify an SQL access level of NO SQL, CONTAINS SQL, READS SQL DATA, or MODIFIES SQL DATA. See Appendix C of *DB2 SQL Reference* for a list of the SQL statements that you can execute for each access level.

Differences between Java routines and other routines

A Java routine differs from stored procedures that are written in other languages in the following ways:

- A Java routine must be defined with PARAMETER STYLE JAVA. PARAMETER STYLE JAVA specifies that the routine uses a parameter-passing convention that conforms to the Java language and SQLJ specifications. DB2 passes INOUT and OUT parameters as single-entry arrays. This means that in your Java routine, you must declare OUT or INOUT parameters as arrays. For example, suppose that stored procedure sp_one_out has one output parameter of type int. You declare the parameter like this:

  ```
  public static void routine_one_out (int[] out_parm)
  ```

- Java routines that are Java main methods have these restrictions:
  - The method must have a signature of String[]. It must be possible to map all the parameters to Java variables of type java.lang.String.
  - The routine can have only IN parameters.
- You cannot make instrumentation facility interface (IFI) calls in Java routines.
- As in other Java programs, you cannot include the following statements in a Java routine:
  - CONNECT
  - RELEASE
  - SET CONNECTION
- The mappings between data types for routine parameters and host data types follow the rules for mappings between SQL and SQLJ data types shown in "Java, JDBC, and SQL data types" on page 127.
- The technique for returning result sets from Java stored procedures is different from the technique for returning result sets in other stored procedures. See "Writing a Java stored procedure to return result sets" on page 216 for information on how to cause a Java stored procedure to return result sets.

Using static and non-final variables in a Java routine

Using static and non-final variables can cause problems for Java routines for the following reasons:

- Use of variables that are static and non-final reduces portability.

  Because the ANSI/ISO standard does not include support for static and non-final variables, different database products might process those variables differently.

- A sequence of routine invocations is not necessarily processed by the same JVM, and static variable values are not shared among different JVMs.

  For example, suppose that two stored procedures, INITIALIZE and PROCESS, use the same static variable, sv1. INITIALIZE sets the value of sv1, and PROCESS depends on the value of sv1. If INITIALIZE runs in one JVM, and then PROCESS runs in another JVM, sv1 in PROCESS does not contain the value that INTIALIZE set.

  Specifying NUMTCB=1 in the WLM-established stored process space startup procedure is not sufficient to guarantee that a sequence of routine invocations go to the same JVM. Under load, multiple stored procedure address spaces are initiated, and each address space has its own JVM. Multiple invocations might be directed to multiple address spaces.

- By default, on z/OS systems, static variables are reset whenever the JVM goes through a reset cycle.

  The default number of stored procedure invocations or user-defined function references before static variables are reset is 256. You can change this value for the WLM address space. However, the routine has no control over this value.

In certain cases, you need to declare variables as static and non-final. In those cases, you can use one of the following techniques to make your routines work correctly with static variables. For the JVM information that is discussed in these descriptions, see *Persistent Reusable Java Virtual Machine User's Guide*.

Static variable technique 1 (simpler):

Set up the JVM to load classes as shareable application classes. This happens automatically for classes that are in the CLASSPATH, and for classes that are loaded from an installed JAR. In this case, static variables in Java routines are stored in the application-class system heap, and might be impacted by JVM resets. The application-class system heap is a segregated part of the system heap, and contains shareable application-class objects that persist for the lifetime of the JVM.

To determine whether the values of static data in a routine have persisted across routine invocations, define a static boolean variable in the class that contains the routine. Initially set the variable to false, and then set it to true when you set the value of static data. Check the value of the boolean variable at the beginning of the routine. If the value is true, the static data has persisted. Otherwise, the data values need to be set again. With this technique, static data values are not set for most routine invocations, but are set more than once during the lifetime of the JVM. Also, with this technique, it is not a problem for a routine to execute on different JVMs for different invocations.

Static variable technique 2 (more complex):

Set up the JVM to load classes as trusted middleware classes. These classes go on the middleware heap. The middleware heap contains objects that persist across JVM resets. During A JVM reset, tidy-up and reinitialize methods can be used to reset the classes to a known initialization state ready for their next use.

The advantage of this technique is that a class can be aware of the reset events and is not subject to the default reinitialization of static variables when the JVM is reset. The class must manage its own static data. In addition, trusted middleware can perform some operations that shareable application classes cannot.

The disadvantages of this technique are:
- This technique does not address the case in which initialization occurs in one JVM and use occurs in another.
- A trusted class cannot be stored in an installed JAR.
- Errors in trusted middleware classes can be hard to diagnose.
- Because trusted middleware classes can be more powerful than application classes, errors in their coding can cause larger problems than errors in application classes.

To make a class a trusted middleware class, specify the directory or JAR that contains that class in the TMSUFFIX environment variable in the JAVAENV data set, but not in the CLASSPATH. See "Setting the run-time environment for interpreted Java stored procedures" on page 202.

Writing a Java stored procedure to return result sets

Your stored procedure can return multiple query result sets to a client program if the following conditions are satisfied:

- The client supports the DRDA code points that are used to return query result sets.

- The value of DYNAMIC RESULT SETS in the stored procedure definition is greater than 0.

For each result set you want to be returned, your Java stored procedure must perform the following actions:

- For each result set, include an object of type java.sql.ResultSet[] or an array of an SQLJ iterator class in the parameter list for the stored procedure method. If the stored procedure definition includes a method signature, for each result set, include java.sql.ResultSet[] or the fully-qualified name of an array of a class that is declared as an SQLJ iterator in the method signature. These result set parameters must be the *last* parameters in the parameter list or method signature. Do *not* include a `java.sql.ResultSet` array or an iterator array in the SQL parameter list of the stored procedure definition.
- Execute a SELECT statement to obtain the contents of the result set.
- Retrieve any rows that you do *not* want to return to the client.
- Assign the contents of the result set to element 0 of the `java.sql.ResultSet[]` object or array of an SQLJ iterator class that you declared in step 217.
- Do not close the `ResultSet`, the statement that generated the `ResultSet`, or the connection that is associated with the statement that generated the `ResultSet`.

  DB2 does not return result sets for `ResultSets` that are closed before the stored procedure terminates.

Figure 64 shows an example of a Java stored procedure that uses an SQLJ iterator to retrieve a result set.

```
package s1;

import sqlj.runtime.*;
import java.sql.*;
import java.math.*;
#sql iterator NameSal(String LastName, BigDecimal Salary);                  1
public class S1Sal
{
  public static void getSals(BigDecimal[] AvgSalParm,
                             java.sql.ResultSet[] rs)                       2
    throws SQLException
  {
    NameSal iter1;
    try
    {
      #sql iter1 = {SELECT LASTNAME, SALARY FROM EMP                        3
                    WHERE SALARY>0 ORDER BY SALARY DESC};
      #sql {SELECT AVG(SALARY) INTO :(AvgSalParm[0]) FROM EMP};             4
    }
    catch (SQLException e)
    {
      System.out.println("SQLCODE returned: " + e.getErrorCode());
      throw(e);
    }
    rs[0] = iter1.getResultSet();                                           5
  }
}
```

Figure 64. Java stored procedure that returns a result set

Notes to Figure 64:

1 This SQLJ clause declares the iterator named NameSal, which is used to retrieve the rows that will be returned to the stored procedure caller in a result set.

2 The declaration for the stored procedure method contains declarations for a single passed parameter, followed by the declaration for the result set object.

3 This SQLJ clause executes the SELECT to obtain the rows for the result set, constructs an iterator object that contains those rows, and assigns the iterator object to variable iter1.

4 This SQLJ clause retrieves a value into the parameter that is returned to the stored procedure caller.

5 This statement uses the `getResultSet` method to assign the contents of the iterator to the result set that is returned to the caller.

Testing a Java routine

Before you invoke your Java routines from SQL applications, it is a good idea to run the routines as stand-alone programs, which are easier to debug. A Java program that runs as a routine requires only a DB2 package. However, before you can run the program as a stand-alone program, you need to bind a DB2 plan for it.

When you are ready to test your programs as Java routines, include a JSPDEBUG DD statement in your startup procedure for the stored procedure address space. This DD statement specifies a data set to which DB2 writes debug information as the Java routines execute.

Another technique that you can use for debugging is to include `System.out.println` and `System.err.println` calls in your program to write messages to the STDERR and STDOUT files. If you are using the Java SDK 1.3.1, you need to include JAVAOUT and JAVAERR DD statements in the WLM address space startup procedure to indicate the z/OS UNIX System Services data sets to which STDOUT and STDERR map. The DD statements look like these:

```
//JAVAOUT DD PATH='/u/db281/javasp/JAVAOUT.TXT',
// PATHOPTS=(ORDWR,OCREAT,OAPPEND),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
//JAVAERR DD PATH='/u/db281/javasp/JAVAERR.TXT',
// PATHOPTS=(ORDWR,OCREAT,OAPPEND),
// PATHMODE=(SIRUSR,SIWUSR,SIRGRP,SIWGRP,SIROTH,SIWOTH)
```

The z/OS UNIX System Services directories that are specified by the PATH parameter must exist on your system. The PATHOPTS options OCREAT and OAPPEND cause the files that are specified in the PATH parameter to be created if they do not exist, or to be appended if they exist.

If you are using the Java SDK 1.4.1 or later, and you do not include JAVAOUT and JAVAERR DD statements in your WLM address space startup procedure, STDERR output is written to the directory that is specified by the WORK_DIR parameter in the JAVAENV data set. If no WORK_DIR parameter is specified, output goes to the default directory.

Chapter 6. Preparing and running JDBC and SQLJ programs

DB2 UDB for z/OS Java programs run in the z/OS UNIX System Services environment. The following topics contain information about preparing and running Java programs:

- "Preparing JDBC programs for execution"
- "Preparing SQLJ programs for execution under the DB2 Universal JDBC Driver"
- "Preparing SQLJ programs for execution under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 239
- "Preparing Java routines for execution" on page 245
- "Running JDBC and SQLJ programs" on page 249

Preparing JDBC programs for execution

Preparing a Java program that contains only JDBC methods is the same as preparing any other Java program. You compile the program using the `javac` command. No precompile or bind steps are required. For example, to prepare the Sample01.java program for execution, execute this command from the /usr/lpp/db2810/ directory:

```
javac Sample01.java
```

Preparing SQLJ programs for execution under the DB2 Universal JDBC Driver

To prepare an SQLJ application to run in a JVM, and with the DB2 Universal JDBC Driver, follow these steps:

1. Translate the source code to produce generated Java source code and serialized profiles, and compile the generated source code to product Java bytecodes.
2. Customize the serialized profiles to produce customized serialized profiles and DB2 packages.

Figure 65 on page 220 shows the steps of the program preparation process for a program that uses the DB2 Universal JDBC Driver.

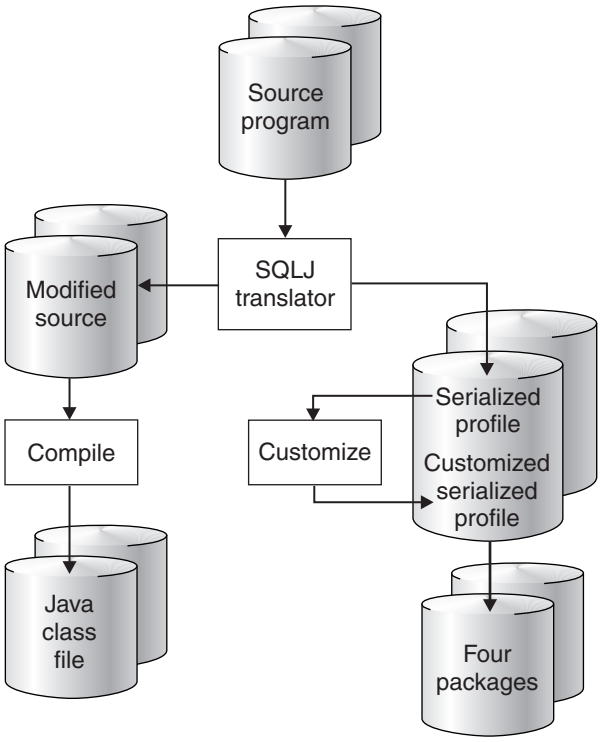
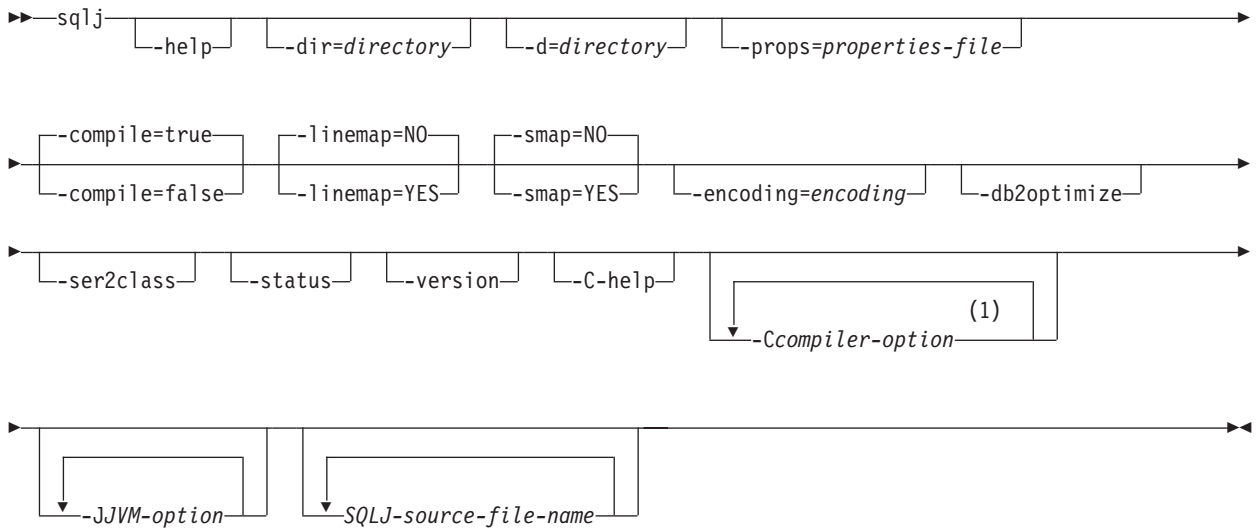


Figure 65. The SQLJ program preparation process for the DB2 Universal JDBC Driver

Translating and compiling SQLJ source code under the DB2 Universal JDBC Driver

The first steps in preparing an executable SQLJ program are to use the SQLJ translator to generate a Java source program, compile the Java source program, and produce zero or more serialized profiles. You issue the `sqlj` command from the z/OS UNIX System Services command line to invoke the SQLJ translator. The SQLJ translator runs without connecting to DB2.

sqlj syntax



Notes:

1 The -C-classpath and -C-sourcepath options are used by the SQLJ translator as well as by the Java compiler.

sqlj parameter descriptions

-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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver. 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-Xinitacsh512
```

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.

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.

Customizing an SQLJ serialized profile under the DB2 Universal JDBC Driver

After you use the SQLJ translator to generate serialized profiles for an SQLJ program, you need to customize each serialized profile.

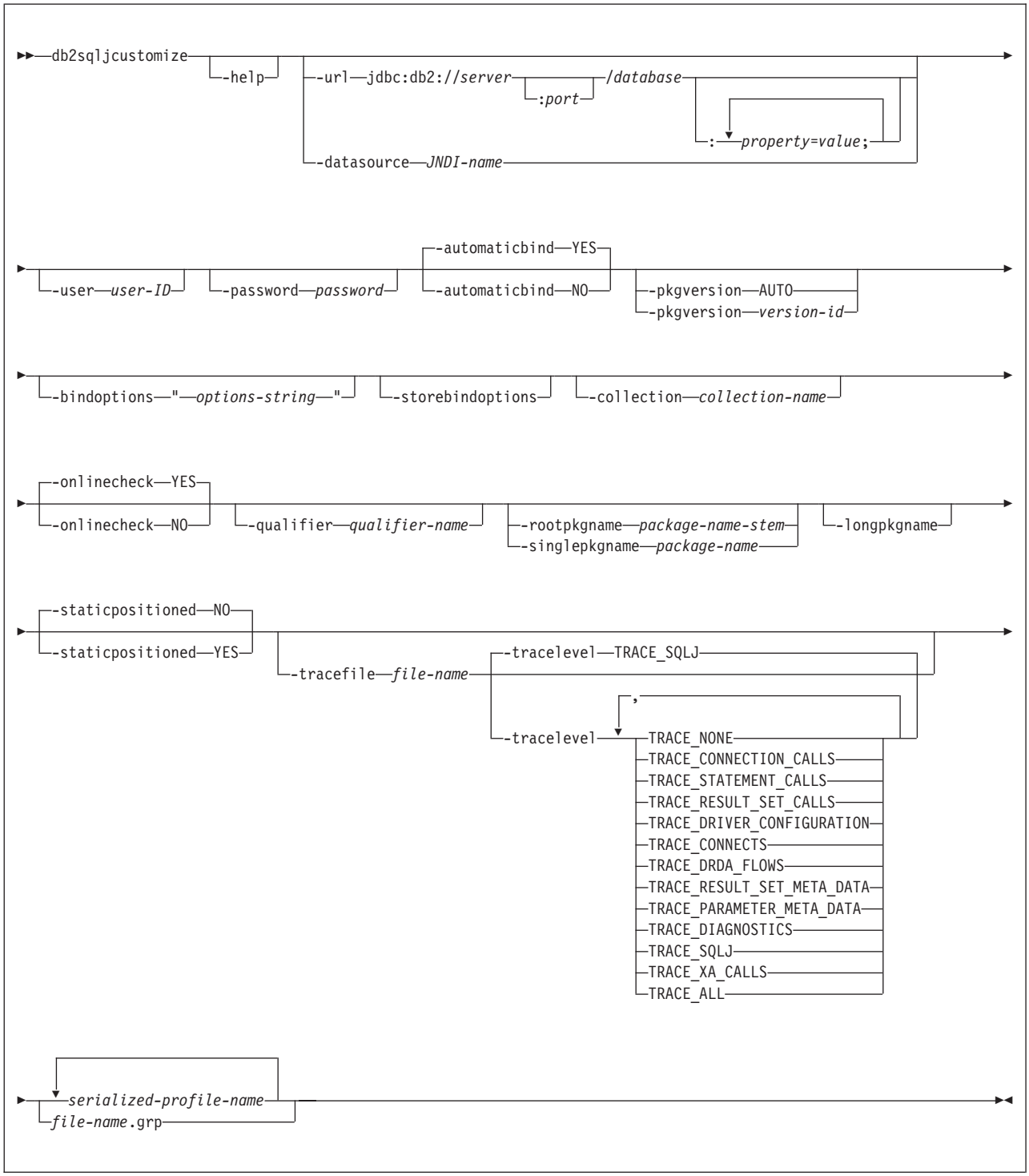
Execute the `db2sqljcustomize` command on the z/OS UNIX System Services command line to produce a customized serialized profile, and optionally, to produce DB2 packages at a specified data source. You can produce the customized serialized profile and DB2 packages on any data source against which a DB2 Universal JDBC Driver runs.You can also use the `db2sqljcustomize` command to do online checking.

By default, `db2sqljcustomize` binds DB2 packages. However, you can disable automatic creation of packages and use the `db2sqljbind` utility to bind packages later. See "Binding the packages for the DB2 Universal JDBC Driver" on page 264 for a description of `db2sqljbind`.

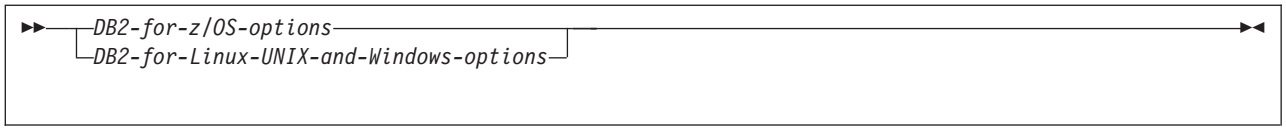
db2sqljcustomize authorization

To execute this command, you must use a privilege set of the process that includes 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
  - PACKADM authority on the collection or on all collections
- If the package exists, the BIND privilege on the package

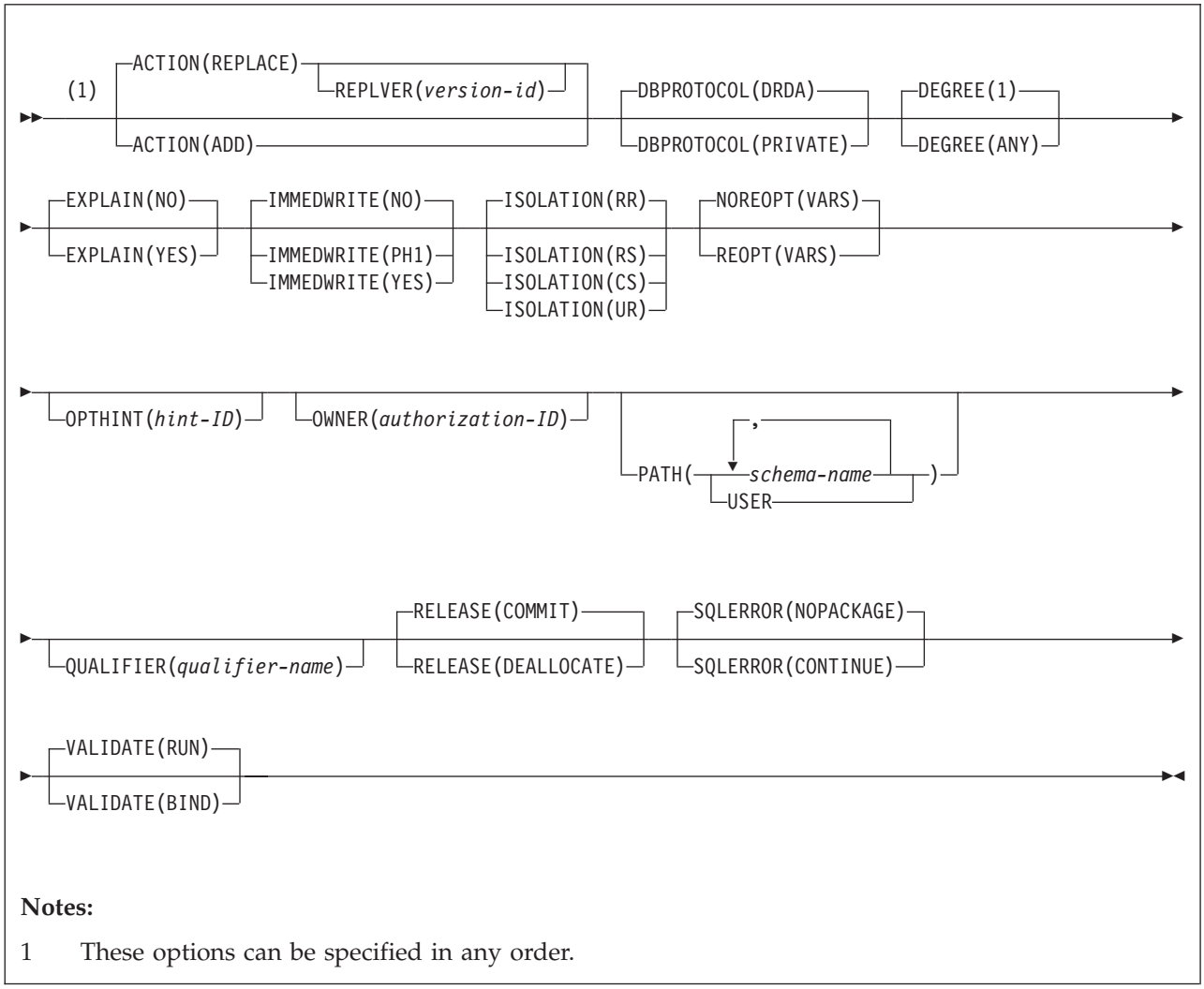
db2sqljcustomize syntax



options-string:

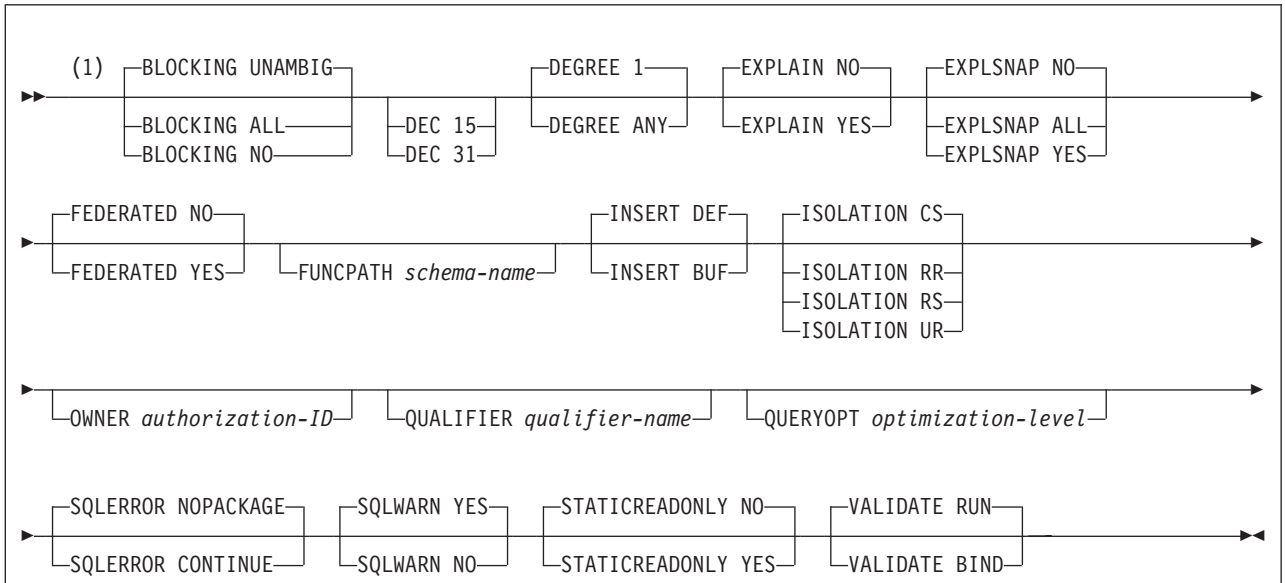

DB2-for-z/OS-options
DB2-for-Linux-UNIX-and-Windows-options

DB2 UDB for z/OS options:



Notes:

1 These options can be specified in any order.



Notes:

1 These options can be specified in any order.

db2sqljcustomize parameter descriptions

-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;
```

property=value;
A property for the JDBC connection. For the definitions of these properties, see "Properties for the DB2 Universal JDBC Driver" on page 185.

-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 Universal Driver 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. See the Authorization topic under BIND PACKAGE Chapter 2 of *DB2 Command Reference* for the authorization that is needed to bind a package on DB2 UDB for z/OS.

-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 UDB for z/OS system, specify DB2 UDB for z/OS options. If you are preparing your program to run on a DB2 UDB for Linux, UNIX and Windows system, specify DB2 UDB for Linux, UNIX and Windows options.

Notes on bind options:

- Specify ISOLATION only if you also specify the -singlepkgname option.

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 DB2 Universal JDBC Driver are different from the values and defaults for DB2. Check the preparation options under "Customizing an SQLJ serialized profile under the DB2 Universal JDBC Driver" on page 224 to determine which options you can use. For information on the meanings of DB2 UDB for z/OS bind options, see *DB2 Command Reference*. For information on the VERSION precompile option, see *DB2 Application Programming and SQL Guide*. For information on precompiler or bind options for DB2 UDB for Linux, UNIX and Windows, see DB2 Universal Database Command Reference.

-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. See "JDBC and SQLJ problem diagnosis with the DB2 Universal JDBC Driver" on page 317 for more information about db2sqljprint.

-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 option determines the data source that is to be used for online checking. The default is YES if the -url 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`_SJProfile`*IDNumber*`.ser`

The four generated package names are of the following form:

*Bytes-from-program-name**IDNumber**PkgIsolation*

Table 49 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 49. 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 50.

Table 50 shows the values of the *PkgIsolation* portion of a package name that is generated by db2sqljcustomize.

Table 50. 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 your IBM service representative.

-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 your IBM service representative.

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*_SJProfile*IDNumber*.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.

db2sqljcustomize output

When `db2sqljcustomize` runs, it creates a customized serialized profile. It also creates DB2 packages, if the automaticbind value is YES.

db2sqljcustomize 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.
- CHAR columns in the result table of an SQLJ SELECT statement at a remote server (db2profc only):

  The JDBC driver cannot describe a SELECT statement that is run at a remote server. Therefore, without online checking, the driver cannot determine the exact data types and lengths of the result table columns. For character columns, the driver assigns a data type and length of VARCHAR(512). Therefore, if you do not perform online checking, and you select data from a CHAR column, the result is a character string of length 512, which is not the desired result.
- Column names in the result table of an SQLJ SELECT statement at a remote server (db2sqljcustomize only):

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.

For example, if you have an SQLJ program that has been running on the JDBC/SQLJ Driver for OS/390 and z/OS, and you want a version of the program that runs on the DB2 Universal JDBC Driver, as well as the existing version, you need to follow these steps:

1. Change the code in your source program that connects to the data source to point to the new driver.
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.

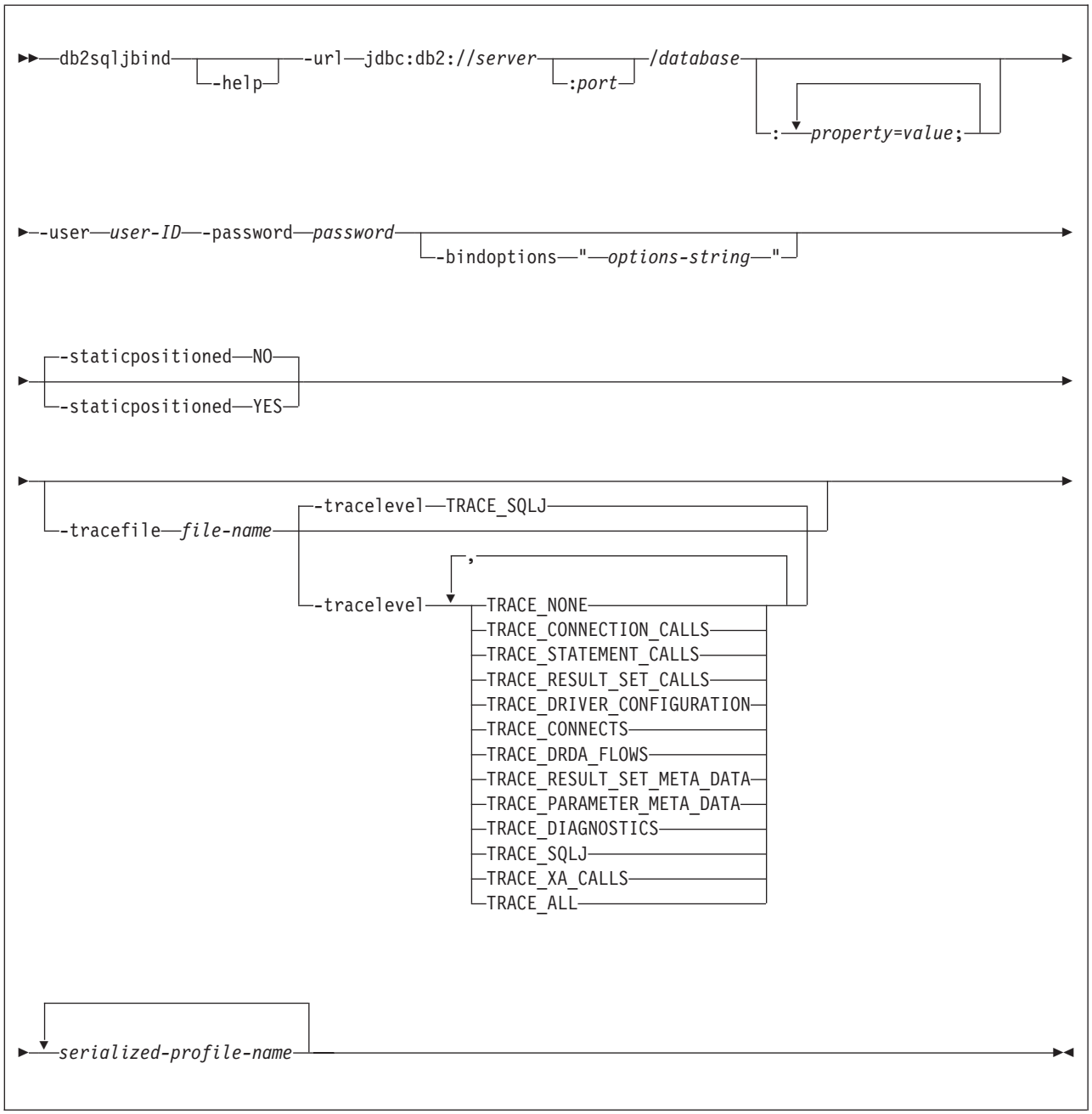
It is essential that you specify the VERSION option when you perfom this step. If you do not, you overwrite your original packages.

When you run the old version of the program that uses the JDBC/SQLJ Driver for OS/390 and z/OS, DB2 loads the old versions of the packages. When you run the new version of the program that uses the DB2 Universal JDBC Driver, DB2 loads the new versions of the packages.

Binding packages after running db2sqljcustomize

Applications that run with the DB2 Universal JDBC Driver require packages but no plans. If the `db2sqljcustomize` -automaticbind option is specified as YES or defaults to YES, `db2sqljcustomize` binds packages for you at the data source that you specify in the -url parameter. However, if automaticbind is NO, if a bind fails when `db2sqljcustomize` runs, or if you want to create identical packages at multiple locations for the same serialized profile, you can use the `db2sqljbind` utility to bind packages.

db2sqljbind syntax



db2sqljbind parameter descriptions

-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;
```

property=value;
A property for the JDBC connection. For the definitions of these properties, see "Properties for the DB2 Universal JDBC Driver" on page 185.

-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 UDB for z/OS system, specify DB2 UDB for z/OS options. If you are preparing your program to run on a DB2 UDB for Linux, UNIX and Windows system, specify DB2 UDB 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 UDB 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.

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 DB2 Universal JDBC Driver are different from the values and defaults for DB2. Check the preparation options in the previous syntax diagram to determine which options you can use. For information on the meanings of DB2 UDB for z/OS bind options, see *DB2 Command Reference*. For information on the VERSION precompile option, see *DB2 Application Programming and SQL Guide*. For information on precompiler or bind options for DB2 UDB for Linux, UNIX and Windows, see DB2 Universal Database Command Reference.

-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 your IBM service representative.

-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 your IBM service representative.

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*_SJProfile*IDNumber*.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.

db2sqljbind 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 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. See "db2sqljcustomize parameter descriptions" on page 228 for information on how to ensure that the cursor and the associated statement are in the same package.
- Fully qualify DB2 table names in positioned UPDATE and positioned DELETE statements.

Preparing SQLJ programs for execution under the JDBC/SQLJ Driver for OS/390 and z/OS

To prepare an SQLJ application to run in a JVM, and with the JDBC/SQLJ Driver for OS/390 and z/OS, follow these steps:

1. Translate the source code to produce generated Java source code and serialized profiles, and compile the generated source code to product Java bytecodes.
2. Customize the serialized profiles. This an optional, but **highly recommended** step. Some SQLJ programs do not operate correctly unless they are customized.
3. Bind plans or packages.

Figure 66 shows the steps of the program preparation process for a program that uses the JDBC/SQLJ Driver for OS/390 and z/OS.

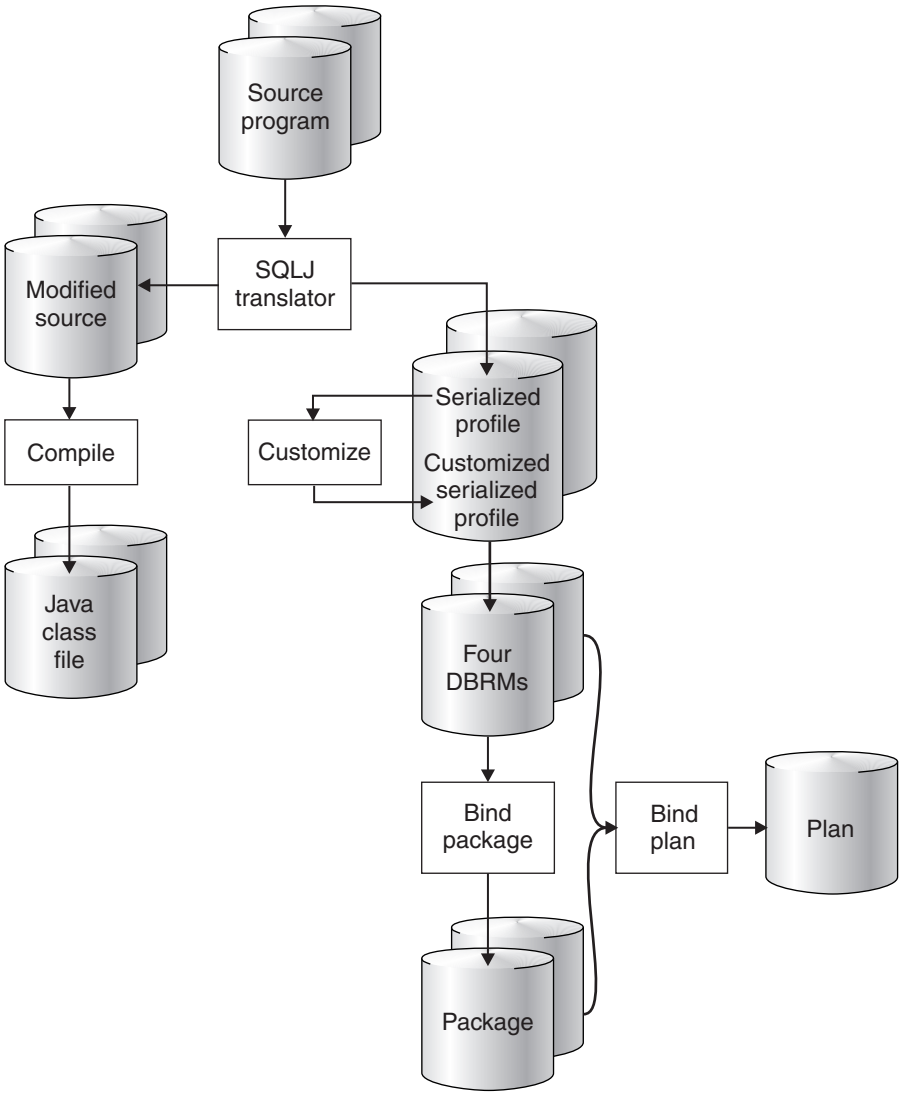


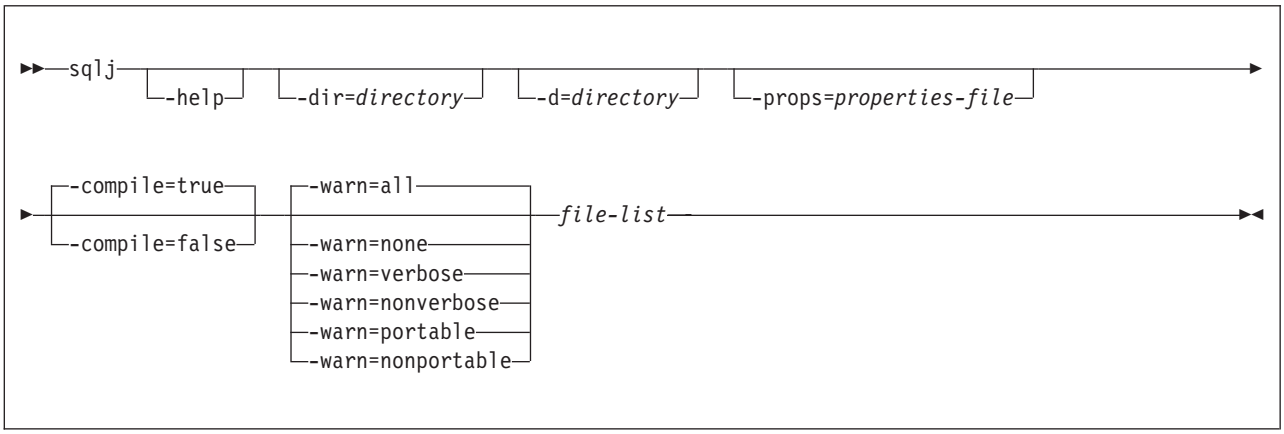
Figure 66. The SQLJ program preparation process for the JDBC/SQLJ Driver for OS/390 and z/OS

Translating and compiling SQLJ source code

The first steps in preparing an executable SQLJ program are to use the SQLJ translator to generate a Java source program, compile the Java source program,

and produce zero or more serialized profiles. You issue the sqlj command from the z/OS UNIX System Services command line to invoke the JDBC/SQLJ Driver for OS/390 and z/OS SQLJ translator. The SQLJ translator runs without connecting to DB2.

sqlj syntax



sqlj parameter descriptions

-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.

-warn=*warning-level*
Specifies the types of messages that the SQLJ translator is to return. The meanings of the warning levels are:

all The translator displays all warnings and informational messages. This is the default.

none
The translator displays no warnings or informational messages.

verbose
The translator displays informational messages about the semantic analysis process.

nonverbose
The translator displays no informational messages about the semantic analysis process.

portable
The translator displays warning messages about the portability of SQLJ clauses.

nonportable
The translator displays no warning messages about the portability of SQLJ clauses.

file-list
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.

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

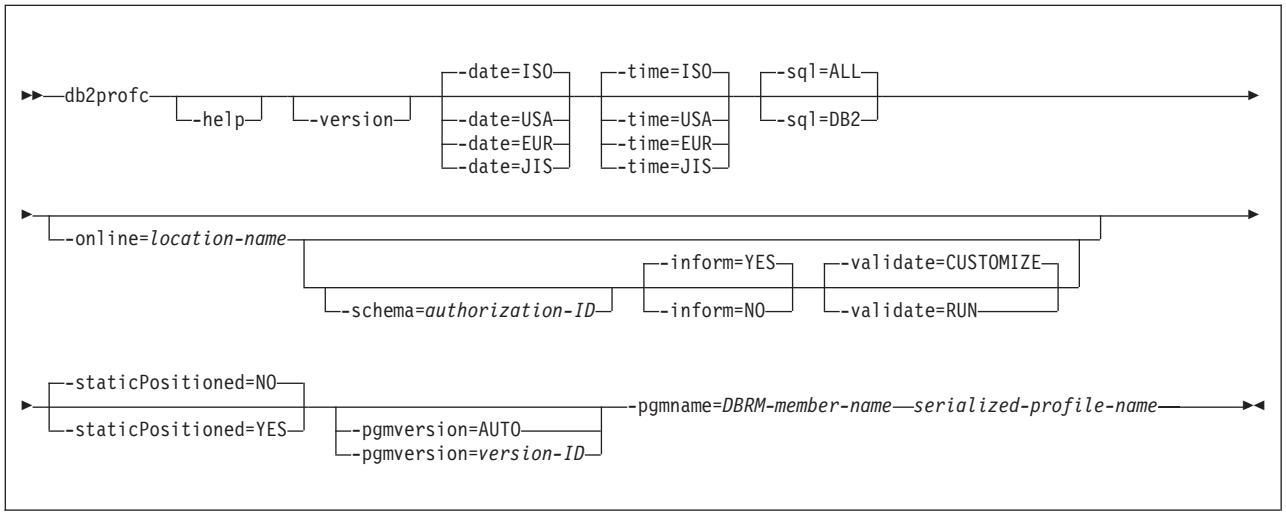
  A serialized profile name is of the following form:

  *program-name*_SJProfile*IDNumber*.ser
- If the SQLJ translator invokes the Java compiler, the class files that the compiler generates.

Customizing an SQLJ serialized profile under the JDBC/SQLJ Driver for OS/390 and z/OS

To produce standard DB2 UDB for z/OS DBRMs and a serialized profile that is customized for DB2 UDB for z/OS, execute the db2profc command on the z/OS UNIX System Services command line.

db2profc Syntax



db2profc parameter descriptions

-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.

-version
Specifies that the SQLJ customizer returns the version of the SQLJ customizer. If any other options are specified with -version, they are ignored.

-date=ISO | USA | EUR | JIS
Specifies that date values that you retrieve from an SQL table should always be in a particular format, regardless of the format that is specified as the location default. For a description of these formats, see Chapter 2 of *DB2 SQL Reference*. The default is ISO.

-time=ISO | USA | EUR | JIS
Specifies that time values that you retrieve from an SQL table should always be in a particular format, regardless of the format that is specified as the location default. For a description of these formats, see Chapter 2 of *DB2 SQL Reference*. The default is ISO.

-sql=ALL | DB2
Indicates whether the source program contains SQL statements other than those that DB2 UDB for z/OS recognizes.

ALL, which is the default, indicates that the SQL statements in the program are not necessarily for DB2 UDB for z/OS. Use ALL for application programs whose SQL statements must execute on a server other that DB2 UDB for z/OS.

DB2 indicates that the DB2 bind process should interpret SQL statements and check syntax for use by DB2 UDB for z/OS. Use DB2 when the database server is DB2 UDB for z/OS.

-online=*location-name*
Specifies that the SQLJ customizer connects to DB2 to do online checking of data types in the SQLJ program. *location-name* is the location name that corresponds to a DB2 subsystem to which the SQLJ customizer connects to do online checking. The name of the DB2 subsystem is specified in the DB2SQLJSSID keyword in the SQLJ run-time properties file.

Before you can do online checking, your SQLJ/JDBC environment must include a JDBC profile. See "Customizing the JDBC profile (optional)" on page 286 for information.

Online checking is optional. However, to get the best mapping of Java data types to DB2 data types, it is recommended that you request online checking.

-schema=*authorization-ID*
Specifies the authorization ID that the SQLJ customizer uses to qualify unqualified DB2 object names in the SQLJ program during online checking.

-inform=YES | NO
Indicates whether informational messages are generated when online checking is bypassed. The default is YES.

-validate=CUSTOMIZE | RUN
Indicates whether customization terminates when online checking detects errors in the application. CUSTOMIZE causes customization to terminate when online checking detects errors. RUN causes customization to continue when online checking detects errors. RUN should be used if tables that are used by the application do not exist at customization time. The default is CUSTOMIZE.

-staticPositioned=NO | YES
Indicates whether the DB2 processes positioned UPDATE or DELETE statements as static SQL statements.

`NO`, which is the default, DB2 processes positioned UPDATE or DELETE statements dynamically.

`YES` indicates that DB2 processes positioned UPDATE or DELETE statements as static SQL statements. Specifying `YES` can improve the performance of programs that contain positioned UPDATE or DELETE statements. However, if you pass iterators as variables between methods, you might need to modify applications that use those iterators. See "db2profc usage notes" on page 244 for details.

-pgmversion=*version-ID* **| AUTO**
Specifies a version identifier that the SQLJ customizer puts in the DBRMs and in the customized profile. The DB2 bind process puts this version identifier in the DB2 package. This parameter has the same function as the DB2 precompiler VERSION option. See Part 5 of *DB2 Application Programming and SQL Guide* for more information about the VERSION option.

The version identifier must be an alphanumeric string of 64 bytes or less.

If you specify AUTO, the SQLJ customizer generates a version identifier that is a string representation of the current time.

If you do not specify the pgmversion parameter, the version identifier value is an empty string.

-pgmname=*DBRM-name*
Specifies the common part of the names for the four DBRMs that the SQLJ customizer generates. *DBRM-name* must be seven or fewer characters in length and must conform to the rules for naming members of MVS partitioned data

sets. See "Binding packages and plans after running db2profc" for information on how to bind each of the DBRMs.

serialized-profile-name
Specifies the name of the serialized profile that is to be customized. A serialized profile name is of the following form:

*program-name*_SJProfile*IDNumber*.ser

db2profc output

When db2profc runs, it creates four DBRMs and customized serialized profiles. The customized serialized profiles overwrite the serialized profiles.

db2profc 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.
- CHAR columns in the result table of an SQLJ SELECT statement at a remote server (db2profc only):

  The JDBC driver cannot describe a SELECT statement that is run at a remote server. Therefore, without online checking, the driver cannot determine the exact data types and lengths of the result table columns. For character columns, the driver assigns a data type and length of VARCHAR(512). Therefore, if you do not perform online checking, and you select data from a CHAR column, the result is a character string of length 512, which is not the desired result.
- Column names in the result table of an SQLJ SELECT statement at a remote server (db2sqljcustomize only):

  Without online checking, the driver cannot determine the column names for the result table of a remote SELECT.

Online checking restriction: If a query produces an intermediate result table, the customizer cannot do online checking of that query and issues a warning message.

Binding packages and plans after running db2profc

Binding an SQLJ plan after running db2profc involves these steps:

1. Bind the DBRMs that are produced by the SQLJ customizer.

   You can bind the DBRMs directly into a plan or bind the DBRMs into packages and then bind the packages into a plan.The SQLJ customizer produces four DBRMs, one for each DB2 isolation level with which the application can run. Table 51 on page 245 shows the name of each DBRM and the isolation level

that you need to specify when you bind that DBRM.

Table 51. SQLJ DBRMs and their isolation levels

DBRM name	Bind with isolation level
*DBRM-name*1	Uncommitted read (UR)
*DBRM-name*2	Cursor stability (CS)
*DBRM-name*3	Read stability (RS)
*DBRM-name*4	Repeatable read (RR)

2. Bind the JDBC packages into your SQLJ plan. The default names of the JDBC packages are:
   - DSNJDBC.DSNJDBC1
   - DSNJDBC.DSNJDBC2
   - DSNJDBC.DSNJDBC3
   - DSNJDBC.DSNJDBC4
3. Ensure that the JDBC profile is in a directory that is specified in the CLASSPATH environment variable, or the path that contains the JDBC profile must be specified in the SQLJ/JDBC run-time properties file, with the db2.jdbc.profile.pathname property. "Customizing the JDBC profile (optional)" on page 286 explains how to create the JDBC profile.

For programs that include both statically executed and dynamically executed statements, such as programs that include JDBC methods as well as SQLJ statements, it is recommended that you bind your SQLJ plans with the DYNAMICRULES(BIND) option. This option causes DB2 to use uniform authorization and object qualification rules for dynamic and static SQL statements.

For more information on binding packages and plans, see Chapter 2 of *DB2 Command Reference*.

Preparing Java routines for execution

Java *routines* are user-defined functions or stored procedures that are written in Java. Java stored procedures or user-defined functions are referred to in this topic as interpreted Java routines. This topic explains how to prepare Java routines for execution.

See "Preparing JDBC programs for execution" on page 219, "Preparing SQLJ programs for execution under the DB2 Universal JDBC Driver" on page 219, or "Preparing SQLJ programs for execution under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 239 for detailed information on program preparation steps that are common to all JDBC or SQLJ programs. See "Defining a Java routine to DB2" on page 206 for information on defining Java routines and JAR files to DB2.

This topic outlines the program preparation steps for interpreted Java routines. Those steps vary, depending on whether your routine contains embedded SQL statements.

Preparing interpreted Java routines with no SQLJ statements

If the program contains only JDBC methods or no SQL statements, use one of the following procedures for program preparation.

Procedure 1: Use this procedure if you run your Java routine from a JAR file. This procedure is recommended over procedure 2.

1. Run the `javac` command to compile the Java program to produce Java bytecodes.
2. Run the `jar` command to collect the class files that contain the methods for your routine into a JAR file. See "Creating JAR files for Java routines" on page 247 for information on creating the JAR file.
3. Call the INSTALL_JAR stored procedure to define the JAR file to DB2.
4. If another user defines the routine to DB2, execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR file to that user.
5. Execute the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the EXTERNAL NAME parameter with the name of the JAR that you defined to DB2 in step 3.
6. Execute the SQL GRANT statement to grant the EXECUTE privilege on the routine to the appropriate users.

Procedure 2: Use this procedure if you do not run your Java routine from a JAR file.

1. Run the `javac` command to compile the Java program to produce Java bytecodes.
2. Ensure that the HFS directory that contains the class files for your routine is in the CLASSPATH for the WLM-established stored procedure address space.

   You specify this CLASSPATH in the JAVAENV data set. You specify the JAVAENV data set using a JAVAENV DD statement in the startup procedure for the WLM-established stored procedure address space.

   If you need to modify the CLASSPATH environment variable in the JAVAENV data set to include the directory for the Java routine's classes, you must restart the WLM address space to make it use the modified CLASSPATH.
3. Execute the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the EXTERNAL NAME parameter without a JAR name.
4. Execute the SQL GRANT statement to grant the EXECUTE privilege on the routine to the appropriate users.

Procedure 3:

Use DB2 Development Center to perform all of the program preparation steps.

Preparing interpreted Java routines with SQLJ statements

If the program contains embedded SQL statements, use one of the following procedures for program preparation.

Procedure 1: Use this procedure if you run your Java routine from a JAR file. This procedure is recommended over procedure 2.

1. Run the `sqlj` command to translate the source code to produce generated Java source code and serialized profiles, and to compile the Java program to produce Java bytecodes.
2. If you are using the JDBC/SQLJ Driver for OS/390 and z/OS, run the `db2profc` command to produce serialized profiles that are customized for DB2 UDB for z/OS and DBRMs.

   If you are using the DB2 Universal JDBC Driver, run the `db2sqljcustomize` command to produce serialized profiles that are customized for DB2 UDB for z/OS and DB2 packages.

3. Run the `jar` command to package the class files that contain the methods for your routine, and the profiles that you generated in step 2 on page 246 into a JAR file. See "Creating JAR files for Java routines" for information on creating the JAR file.
4. Call the INSTALL_JAR stored procedure to define the JAR file to DB2.
5. If another user defines the routine to DB2, execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR file to that user.
6. Execute the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the EXTERNAL NAME parameter with the name of the JAR that you defined to DB2 in step 4.
7. If you are using the JDBC/SQLJ Driver for OS/390 and z/OS, run the DB2 `BIND` command to bind the DBRMs that you created in step 2 on page 246 into packages.

Procedure 2: Use this procedure if you do not run your Java routine from a JAR file.

1. Run the `sqlj` command to translate the source code to produce generated Java source code and serialized profiles, and to compile the Java program to produce Java bytecodes.
2. If you are using the JDBC/SQLJ Driver for OS/390 and z/OS, run the `db2profc` command to produce serialized profiles that are customized for DB2 UDB for z/OS and DBRMs.

   If you are using the DB2 Universal JDBC Driver, run the `db2sqljcustomize` command to produce serialized profiles that are customized for DB2 UDB for z/OS and DB2 packages.
3. Ensure that the HFS directory that contains the class files for your routine is in the CLASSPATH for the WLM-established stored procedure address space.

   You specify this CLASSPATH in the JAVAENV data set. You specify the JAVAENV data set using a JAVAENV DD statement in the startup procedure for the WLM-established stored procedure address space.

   If you need to modify the CLASSPATH environment variable in the JAVAENV data set to include the directory for the Java routine's classes, you must restart the WLM address space to make it use the modified CLASSPATH.
4. Use the SQL CREATE PROCEDURE or CREATE FUNCTION statement to define the routine to DB2. Specify the EXTERNAL NAME parameter without a JAR name.
5. If you are using the JDBC/SQLJ Driver for OS/390 and z/OS, run the DB2 `BIND` command to bind the DBRMs that you created in step 2 into packages.

Procedure 3:

Use DB2 Development Center to perform all of the program preparation steps.

Creating JAR files for Java routines

A convenient way to ensure that all modules of a Java routine are accessible is to store those modules in a JAR file. You create the JAR file by running the `jar` command in z/OS UNIX System Services. To create the JAR file, follow these steps:

1. If the Java source file does not contain a `package` statement, change to the directory that contains the class file for the Java routine, which you created by running the javac command.

   For example, if JDBC routine Add_customer.java is in /u/db2res3/acmejos, change to directory /u/db2res3/acmejos.

If the Java source file contains a `package` statement, change to the directory that is one level above the directory that is named in the `package` statement.

For example, suppose the package statement is:

```
package lvlOne.lvlTwo.lvlThree;
```

Change to the directory that contains lvlOne as an immediate subdirectory.

2. Run the `jar` command. You might need to specify at least these options:

   c Creates a new or empty archive.

   v Generates verbose output on `stderr`.

   f Specifies that the argument immediately after the options list is the name of the JAR file to be created.

   For example, to create a JAR file named acmejos.jar from Add_customer.class, which is in package acmejos, execute this `jar` command:

   ```
   jar -cvf acmejos.jar acmejos/Add_customer.class
   ```

   To create a JAR file for an SQLJ routine, you also need to include all generated class files, such as classes that are generated for iterators, and all serialized profile files. For example, suppose that all classes are declared to be in package acmejos, and all class files, including generated class files, and all serialized profile files for SQLJ routine Add_customer.sqlj are in directory /u/db2res3/acmejos/. To create a JAR file named acmejos.jar, change the the /u/db2res3 directory, and then issue this `jar` command:

   ```
   jar -cvf acmejos.jar acmejos/*.class acmejos/*.ser
   ```

Example of preparing a Java routine for execution

The following example demonstrates how to prepare the SQLJ stored procedure that is shown in Figure 64 on page 217 for execution using the DB2 Universal JDBC Driver. This example uses Procedure 1 in "Preparing interpreted Java routines with SQLJ statements" on page 246.

1. On z/OS UNIX System Services, run the `sqlj` command to translate and compile the SQLJ source code.

   Assume that the path for the stored procedure source program is /u/db2res3/s1/s1sal.sqlj. Change to directory /u/db2res3/s1, and issue this command:

   ```
   sqlj s1sal.sqlj
   ```

   After this process completes, the /u/db2res3/s1 directory contains these files:

   ```
   s1sal.java
   s1sal.class
   s1sal_SJProfile0.ser
   ```

2. On z/OS UNIX System Services, run the `db2sqljcustomize` command to produce serialized profiles that are customized for DB2 UDB for z/OS and to bind the DB2 packages for the stored procedure.

   Change to the /u/db2res3 directory, and issue this command:

   ```
   db2sqljcustomize -url jdbc:db2://mvs1:446/SJCEC1
     -user db2adm -password db2adm \
     -bindoptions "EXPLAIN YES" \
     -collection ADMCOLL \
     -rootpkgname S1SAL \
     s1sal_SJProfile0.ser
   ```

   After this process completes, s1sal_SJProfile0.ser contains a customized serialized profile. The DB2 subsystem contains these packages:

```
S1SAL1
S1SAL2
S1SAL3
S1SAL4
```

3. On z/OS UNIX System Services, run the `jar` command to package the class files that you created in step 1 on page 248 and the customized serialized profile that you created in step 2 on page 248 into a JAR file.

   Change to the /u/db2res3 directory, and issue this command:

   ```
   jar -cvf s1sal.jar s1/*.class s1/*.ser
   ```

   After this process completes, the /u/db2res3 directory contains this file:

   ```
   s1sal.jar
   ```

4. Call the INSTALL_JAR stored procedure, which is on DB2 UDB for z/OS, to define the JAR file to DB2.

   You need to execute the CALL statement from a static SQL program or from an ODBC or JDBC program. The CALL statement looks similar to this:

   ```
   CALL SQLJ.INSTALL_JAR('file:/u/db2res3/s1sal.jar','MYSCHEMA.S1SAL',0);
   ```

   The exact form of the CALL statement depends on the language of the program that issues the CALL statement.

   After this process completes, the DB2 catalog contains JAR file MYSCHEMA.S1SAL.

5. If another user defines the routine to DB2, on DB2 UDB for z/OS, execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR file to that user.

   Suppose that you want any user to be able to define the stored procedure to DB2. This means that all users need the USAGE privilege on JAR MYSCHEMA.S1SAL. To grant this privilege, execute this SQL statement:

   ```
   GRANT USAGE ON JAR MYSCHEMA.S1SAL TO PUBLIC;
   ```

6. On DB2 UDB for z/OS, execute the SQL CREATE PROCEDURE statement to define the stored procedure to DB2:

   ```
   CREATE PROCEDURE SYSPROC.S1SAL
    (DECIMAL(10,2) INOUT)
     FENCED
     MODIFIES SQL DATA
     COLLID ADMCOLL
     LANGUAGE JAVA
     EXTERNAL NAME  'MYSCHEMA.S1SAL:s1.S1Sal.getSals'
     WLM ENVIRONMENT WLMIJAV
     DYNAMIC RESULT SETS 1
     PROGRAM TYPE SUB
     PARAMETER STYLE JAVA;
   ```

Running JDBC and SQLJ programs

After you have set the environment variables discussed in “Setting environment variables for the JDBC/SQLJ Driver for OS/390 and z/OS” on page 280 and prepared your program for execution, your program is ready to run.

To ensure that the program can find all the files that it needs:

- For an SQLJ program, put the serialized profiles for the program in the same directory as the class files for the program.
- Include class files that are used by the program in the CLASSPATH.

To run your JDBC or SQLJ program, execute the `java` command from the z/OS UNIX System Services command line:

```
java program-name
```

Chapter 7. Installing the DB2 Universal JDBC Driver

The procedures in this topic describe what you need to do to install the DB2 Universal JDBC Driver. The procedures are:

- "Installing the DB2 Universal JDBC Driver as part of a DB2 installation"

  Follow this procedure to provide DB2 Universal JDBC Driver type 2 connectivity and DB2 Universal JDBC Driver type 4 connectivity on a z/OS system that has a DB2 UDB for z/OS subsystem.
- "Installing the z/OS Application Connectivity to DB2 for z/OS feature" on page 274

  Follow this procedure to provide DB2 Universal JDBC Driver type 4 connectivity on a z/OS system that does not have a DB2 UDB for z/OS subsystem.

Installing the DB2 Universal JDBC Driver as part of a DB2 installation

To install the DB2 Universal JDBC Driver as part of a DB2 UDB for z/OS installation, follow these steps:

1. Install Java 2 Technology Edition, SDK 1.3.1 or higher. If you plan to implement Java stored procedures and user-defined functions on this DB2 subsystem, install Java 2 Technology Edition, SDK 1.3.1, SDK 1.4.1, or higher.
2. If you plan to use DB2 Universal JDBC Driver type 4 connectivity to connect to DB2 UDB for z/OS Version 7 servers, install OS/390 Support for Unicode or z/OS Support for Unicode on those servers. See Information APARs II13048 and II13049 for more information.
3. On z/OS, enable TCP/IP. See *IBM TCP/IP for MVS: Customization & Administration Guide*.
4. When you allocate and load the DB2 UDB for z/OS libraries, include the steps that allocate and load the DB2 Universal JDBC Driver libraries. See "Loading the DB2 Universal JDBC Driver libraries" on page 252 for details.
5. On DB2 UDB for z/OS, enable distributed data facility (DDF) and TCP/IP support. See Part 3 of *DB2 Installation Guide*.
6. On DB2 UDB for z/OS, set subsystem parameter DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPF. See Part 2 of *DB2 Installation Guide* for information on setting DESCSTAT. This step is necessary for SQLJ support.
7. In z/OS UNIX System Services, edit your .profile file to customize the environment variable settings. You use this step to set the libraries, paths, and files that the DB2 Universal JDBC Driver uses. See "Setting environment variables for the DB2 Universal JDBC Driver" on page 252 for details.
8. ***Optional:*** Customize the DB2 Universal JDBC Driver configuration properties. See "DB2 Universal JDBC Driver configuration properties customization" on page 253 for details.
9. On DB2 UDB for z/OS, enable the DB2-supplied stored procedures and define the tables that are used by the DB2 Universal JDBC Driver. See "Enabling the DB2-supplied stored procedures and defining the tables used by the DB2 Universal JDBC Driver" on page 261.
10. In z/OS UNIX System Services, run the DB2binder utility to bind the packages for the DB2 Universal JDBC Driver. See "Binding the packages for the DB2 Universal JDBC Driver" on page 264.

11. ***If you plan to use Universal Driver type 4 connectivity to implement distributed transactions against DB2 UDB for OS/390 and z/OS Version 7 servers:*** In z/OS UNIX System Services, run the DB2T4XAIndoubtUtil utility on the z/OS system on which you are installing z/OS Application Connectivity to DB2 for z/OS. Run the utility once for each of the DB2 UDB for OS/390 and z/OS Version 7 servers. See "DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers" on page 267 for details.
12. ***If you plan to use LOB locators to access DBCLOB columns in DB2 tables on DB2 UDB for z/OS servers:*** In z/OS UNIX System Services, run the DB2LobTableCreator utility on each of those servers to create tables that are needed for fetching LOB locators. See "Enabling retrieval of DBCLOB columns with LOB locators on DB2 UDB for OS/390 and z/OS servers" on page 271.
13. Verify the installation by running a simple JDBC application. See "Verifying the installation of the DB2 Universal JDBC Driver" on page 272 for suggestions on how to do that.

Loading the DB2 Universal JDBC Driver libraries

When you install DB2 UDB for z/OS, include the steps for allocating the HFS directory structure and using SMP/E to load the DB2 Universal JDBC Driver libraries. The jobs that perform these functions are:

DSNISMKD
Invokes the DSNMKDIR EXEC to allocate the HFS directory structure.

DSNDDEF2
Includes steps to define DDDEFs for the DB2 Universal JDBC Driver libraries.

DSNRECV3
Includes steps that perform the SMP/E RECEIVE function for the DB2 Universal JDBC Driver libraries.

DSNAPPL2
Includes the steps that perform the SMP/E APPLY CHECK and APPLY functions for the DB2 Universal JDBC Driver libraries.

DSNACEP2
Includes the steps that perform the SMP/E ACCEPT CHECK and ACCEPT functions for the DB2 Universal JDBC Driver libraries.

See *IBM DATABASE 2 Universal Database for z/OS Program Directory* for information on allocating and loading DB2 data sets.

Setting environment variables for the DB2 Universal JDBC Driver

The environment variables that you must set are:

PATH
Modify PATH to include the directory that contains the shell scripts that invoke DB2 Universal JDBC Driver program preparation and debugging functions. If the DB2 Universal JDBC Driver is installed in `/usr/lpp/db2810/jcc`, modify PATH as follows:

```
export PATH=/usr/lpp/db2810/jcc/bin:$PATH
```

LIBPATH
The DB2 Universal JDBC Driver contains the following dynamic load libraries (DLLs):

- libdb2jcct2.so
- libdb2jcct2zos.so

Those DLLs contain the native (C or C++) implementation of the DB2 Universal JDBC Driver. The driver uses this code when you use Universal Driver type 2 connectivity.

Modify LIBPATH to include the directory that contains these DLLs. If the DB2 Universal JDBC Driver is installed in `/usr/lpp/db2810/jcc`, modify LIBPATH as follows:

```
export LIBPATH=/usr/lpp/db2810/jcc/lib:$LIBPATH
```

CLASSPATH

The DB2 Universal JDBC Driver contains the following class files:

db2jcc.jar
Contains all JDBC classes and the SQLJ runtime classes for the DB2 Universal JDBC Driver.

db2jcc_javax.jar
Contains a subset of the J2EE classes that are needed by the DB2 Universal JDBC Driver.

sqlj.zip
Contains the classes that are needed to prepare SQLJ applications for execution under the DB2 Universal JDBC Driver.

db2jcc_license_cisuz.jar
A license file that permits access to the DB2 UDB server.

Modify your CLASSPATH to include these files. If the DB2 Universal JDBC Driver is installed in `/usr/lpp/db2810/jcc`, modify CLASSPATH as follows:

```
export CLASSPATH=/usr/lpp/db2810/jcc/classes/db2jcc.jar: \
/usr/lpp/db2810/jcc/classes/db2jcc_javax.jar: \
/usr/lpp/db2810/jcc/classes/sqlj.zip: \
/usr/lpp/db2810/jcc/classes/db2jcc_license_cisuz.jar: \
$CLASSPATH
```

Important: Do not include class files for both the DB2 Universal JDBC Driver and the JDBC/SQLJ Driver for OS/390 and z/OS in your CLASSPATH. The only exception to this rule is that you need to include classes for both drivers in your CLASSPATH while you convert serialized profiles that you customized under the JDBC/SQLJ Driver for OS/390 and z/OS to the format for the DB2 Universal JDBC Driver. See "Converting JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles for the DB2 Universal JDBC Driver" on page 269.

If you use Java stored procedures, you need to set additional environment variables in a JAVAENV data set. See "Setting the run-time environment for interpreted Java stored procedures" on page 202 for more information.

DB2 Universal JDBC Driver configuration properties customization

The DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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.

  For Java stored procedures or user-defined functions, you can set the configuration properties by specifying -D*property=value* for each configuration property in a file whose name you specify in the JVMPROPS option. You specify the JVMPROPS options in the ENVAR option of the Language Environment options string. The Language Environment options string is in a data set that is specified by the JAVAENV DD statement in the WLM address space startup procedure. See "Setting up the WLM application environment for interpreted Java routines" on page 200 for more information.
- 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.

  For Java stored procedures or user-defined functions, you can set the configuration properties by specifying the -Ddb2.jcc.propertiesFile=*path* option in a file whose name you specify in the JVMPROPS option. You specify the JVMPROPS options in the ENVAR option of the Language Environment options string. The Language Environment options string is in a data set that is specified by the JAVAENV DD statement in the WLM address space startup procedure. See "Setting up the WLM application environment for interpreted Java routines" on page 200 for more information.
- Set the configuration properties in a resource named DB2JccConfiguration.properties. A standard Java resource search is used to find DB2JccConfiguration.properties. The DB2 Universal JDBC Driver 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 DB2JccConfiguration.properties is a stand-alone file, the contents are automatically converted to Unicode. If you include DB2JccConfiguration.properties in a JAR file, you need to convert the contents to Unicode before you put them in the JAR file.

  If DB2JccConfiguration.properties is a stand-alone file, the path for DB2JccConfiguration.properties must be in the following places:

  – *For stand-alone Java applications:* Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation.

– *For Java stored procedures or user-defined functions:* Include the directory that contains DB2JccConfiguration.properties in the CLASSPATH concatenation in the ENVAR option of the Language Environment options string. The Language Environment options string is in a data set that is specified by the JAVAENV DD statement in the WLM address space startup procedure. See "Setting up the WLM application environment for interpreted Java routines" on page 200 for more information.

If DB2JccConfiguration.properties is in a JAR file, the JAR file must be in the CLASSPATH concatenation.

Recommendation: Because support for com/ibm/db2/jcc/DB2JccConfiguration.properties as the default resource name for configuration properties is deprecated, use DB2JccConfiguration.properties instead.

Example: Putting DB2JccConfiguration.properties in a JAR file: Suppose that your configuration properties are in a file that is in EBCDIC code page 1047. To put the properties file into a JAR file, follow these steps:

1. Rename DB2JccConfiguration.properties to another name, such as EBCDICVersion.properties.
2. Run the iconv shell utility on the z/OS UNIX System Services command line to convert the file contents to Unicode. For example, to convert EBCDICVersion.properties to a Unicode file named DB2JccConfiguration.properties, issue this command:

   ```
   iconv -f ibm-1047 -t utf-8 EBCDICVersion.properties \
     > DB2JccConfiguration.properties
   ```

3. Execute the jar command to add the Unicode file to the JAR file. In the JAR file, the configuration properties file must be named DB2JccConfiguration.properties. For example:

   ```
   jar -cvf jdbcProperties.jar DB2JccConfiguration.properties
   ```

You can set any of the following DB2 Universal JDBC Driver configuration properties. All properties are optional.

db2.jcc.accountingInterval

Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. If the value of db2.jcc.accountingInterval is COMMIT, DB2 accounting records are produced at commit points. For example:

```
db2.jcc.accountingInterval=COMMIT
```

Otherwise, accounting records are produced on termination of the physical connection to the data source.

db2.jcc.accountingInterval applies only to Universal Driver type 2 connectivity on DB2 UDB for z/OS. db2.jcc.accountingInterval is not applicable to connections under CICS or IMS, or for Java stored procedures.

You can override db2.jcc.accountingInterval by setting the `accountingInterval` property for a `Connection` or `DataSource` object.

db2.jcc.disableSQLJProfileCaching

Specifies whether serialized profiles are cached when the JVM under which their application is running is reset. db2.jcc.disableSQLJProfileCaching applies only to applications that run in a resettable JVM (applications that run in the

CICS, IMS, or Java stored procedure environment), and use Universal Driver type 2 connectivity on DB2 UDB for z/OS. Possible values are:

YES SQLJ serialized profiles are not cached every time the JVM is reset, so that new versions of the serialized profiles are loaded when the JVM is reset. Use this option when an application is under development, and new versions of the application and its serialized profiles are produced frequently.

NO SQLJ serialized profiles are cached when the JVM is reset. NO is the default.

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.lobOutputSize
Specifies the number of bytes of storage that the DB2 Universal JDBC Driver needs to allocate for output LOB values when the driver cannot determine the

size of those LOBs. This situation occurs for LOB stored procedure output parameters. db2.jcc.lobOutputSize applies only to Universal Driver type 2 connectivity on DB2 UDB for z/OS.

The default value for db2.jcc.lobOutputSize is 1048576. For systems with storage limitations and smaller LOBs, set the db2.jcc.lobOutputSize value to a lower number.

For example, if you know that the output LOB size is at most 64000, set db2.jcc.lobOutputSize to 64000.

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.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.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.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.

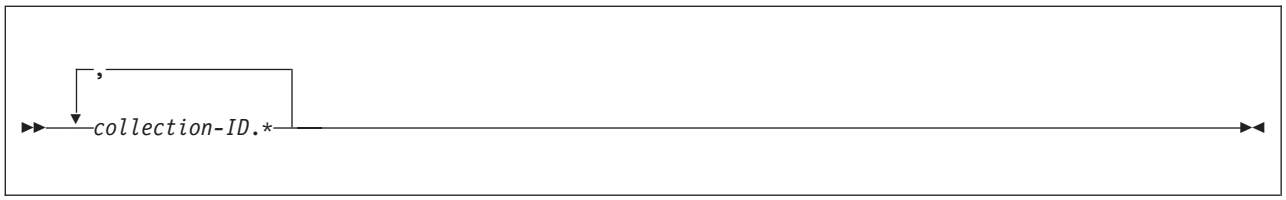
db2.jcc.pkList
Specifies a package list that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established.

Specify this property if you do not bind plans for your SQLJ programs or for the JDBC driver. If you specify this property, **do not specify db2.jcc.planName.**

db2.jcc.pkList does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the db2.jcc.pkList setting in those cases.

Recommendation: Use db2.jcc.pkList instead of db2.jcc.planName.

The format of the package list is:



The default value of db2.jcc.pkList is NULLID.*.

If you specify the -collection parameter when you run `com.ibm.db2.jcc.DB2Binder`, the collection ID that you specify for DB2 Universal JDBC Driver packages when you run `com.ibm.db2.jcc.DB2Binder` must also be in the package list for the db2.jcc.pkList property. See "Binding the packages for the DB2 Universal JDBC Driver" on page 264 for information about `com.ibm.db2.jcc.DB2Binder`.

You can override db2.jcc.pkList by setting the `pkList` property for a `Connection` or `DataSource` object.

The following example specifies a package list for a DB2 Universal JDBC Driver instance whose packages are in collection JDBCCID. SQLJ applications that are prepared under this driver instance are bound into collections SQLJCID1, SQLJCID2, or SQLJCID3.

```
db2.jcc.pkList=JDBCCID.*,SQLJCID1.*,SQLJCID2.*,SQLJCID3.*
```

db2.jcc.planName
Specifies a DB2 plan name that is used for the underlying RRSAF CREATE THREAD call when a JDBC or SQLJ connection to a data source is established. Specify this property if you bind plans for your SQLJ programs and for the JDBC driver packages. If you specify this property, **do not specify db2.jcc.pkList.**

db2.jcc.planName does not apply to applications that run under CICS or IMS, or to Java stored procedures. The JDBC driver ignores the db2.jcc.planName setting in those cases.

If you do not specify this property or the db2.jcc.pkList property, the DB2 Universal JDBC Driver uses the db2.jcc.pkList default value of NULLID.*.

If you specify db2.jcc.planName, you need to bind the packages that you produce when you run `com.ibm.db2.jcc.DB2Binder` into a plan whose name is the value of this property. You also need to bind all SQLJ packages into a plan whose name is the value of this property.

You can override db2.jcc.planName by setting the `planName` property for a `Connection` or `DataSource` object.

The following example specifies a plan name of MYPLAN for the DB2 Universal JDBC Driver JDBC packages and SQLJ packages.

```
db2.jcc.planName=MYPLAN
```

db2.jcc.traceDirectory or db2.jcc.override.traceDirectory
Enables the DB2 Universal JDBC Driver trace for Java driver code, and specifies a directory into which trace information is written. These properties do not apply to Universal Driver type 2 connectivity on DB2 UDB for z/OS. 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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver takes when an uncustomized SQLJ application runs.
db2.jcc.sqljUncustomizedWarningOrException can have the following values:

0 The DB2 Universal JDBC Driver does not throw a `Warning` or `Exception` when an uncustomized SQLJ application is run. This is the default.

1 The DB2 Universal JDBC Driver throws a `Warning` when an uncustomized SQLJ application is run.

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

db2.jcc.ssid
Specifies the name of the DB2 UDB subsystem that is used as the local subsystem when an application uses Universal Driver type 2 connectivity on DB2 UDB for z/OS. For example:

```
db2.jcc.ssid=DB2A
```

Group attachment is not supported.

If you do not specify the db2.jcc.ssid property, the DB2 Universal JDBC Driver uses the SSID value from the DSNHDECP data-only load module. When you install DB2 UDB for z/OS, a DSNHDECP module is created in the *prefix*.SDSNEXIT data set and the *prefix*.SDSNLOAD data set. Other DSNHDECP load modules might be created in other data sets for selected applications.

The DB2 Universal JDBC Driver must load a DSNHDECP module before it can read the SSID value. z/OS searches data sets in the following places, and in the following order, for the DSNHDECP module:

1. Job pack area (JPA)
2. TASKLIB
3. STEPLIB or JOBLIB
4. LPA
5. Libraries in the link list

You need to ensure that if your system has more than one copy of the DSNHDECP module, z/OS finds the data set that contains the correct copy for the DB2 Universal JDBC Driver first.

db2.jcc.traceFile or db2.jcc.override.traceFile
Enables the DB2 Universal JDBC Driver trace for Java driver code, and specifies the name on which the trace file names are based.The db2.jcc.traceFile property does not apply to Universal Driver type 2 connectivity on DB2 UDB for z/OS.

Specify a fully qualified z/OS UNIX System Services file name for the db2.jcc.override.traceFile property value.

The db2.jcc.override.traceFile property overrides the `traceFile` property for a `Connection` or `DataSource` object.

For example, specifying the following setting for db2.jcc.override.traceFile enables tracing of the DB2 Universal JDBC Driver 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. These properties do not apply to Universal Driver type 2 connectivity on DB2 UDB for z/OS. 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.

db2.jcc.t2zosTraceFile
Enables the DB2 Universal JDBC Driver trace for C/C++ native driver code for

Universal Driver type 2 connectivity, and specifies the name on which the trace file names are based. This property is required for collecting trace data for C/C++ native driver code.

Specify a fully qualified z/OS UNIX System Services file name for the db2.jcct.t2zosTraceFile property value.

For example, specifying the following setting for db2.jcct.t2zosTraceFile enables tracing of the DB2 Universal JDBC Driver C/C++ native code to a file named /SYSTEM/tmp/jdbctraceNative:

```
db2.jcc.t2zosTraceFile=/SYSTEM/tmp/jdbctraceNative
```

You should set the trace properties under the direction of IBM Software Support.

db2.jcc.t2zosTraceBufferSize
Specifies the size of a trace buffer in virtual storage that is used for tracing the processing that is done by the C/C++ native driver code. This value is also the maximum amount of C/C++ native driver trace information that can be collected.

Specify a value in kilobytes. The default is 256 KB.

This property is used only if the db2.jcc.t2zosTraceFile property is set.

Recommendation: To avoid a performance impact, specify a value of 1024 or less.

For example, to set a trace buffer size of 1024 KB, use this setting:

```
db2.jcc.t2zosTraceBufferSize=1024
```

You should set the trace properties under the direction of IBM Software Support.

db2.jcc.t2zosTraceWrap
Enables or disables wrapping of the SQLJ trace. db2.jcc.t2zosTraceWrap can have one of the following values:

1 Wrap the trace
0 Do not wrap the trace

The default is 1. This parameter is optional. For example:

```
DB2SQLJ_TRACE_WRAP=0
```

You should set db2.jcc.t2zosTraceWrap only under the direction of IBM Software Support.

Enabling the DB2-supplied stored procedures and defining the tables used by the DB2 Universal JDBC Driver

Before you can use certain functions of the DB2 Universal JDBC Driver on a DB2 UDB for z/OS subsystem, you need to do these things:

- Install the following DB2-supplied stored procedures:
  - SQLCOLPRIVILEGES
  - SQLCOLUMNS
  - SQLFOREIGNKEYS
  - SQLGETTYPEINFO
  - SQLPRIMARYKEYS
  - SQLPROCEDURECOLS
  - SQLPROCEDURES

- SQLSPECIALCOLUMNS
- SQLSTATISTICS
- SQLTABLEPRIVILEGES
- SQLTABLES
- SQLUDTS
- SQLCAMESSAGE

- Define the following tables:
  - SYSIBM.SYSDUMMYU
  - SYSIBM.SYSDUMMYA
  - SYSIBM.SYSDUMMYE

  Those tables ensure that character conversion does not occur when Unicode data is stored in DBCLOB or CLOB columns.

To install the stored procedures and define the tables, you need to perform these steps. It is assumed that you already have WLM installed.

1. Set up a WLM environment for running the stored procedures.

   To set up a WLM application environment for these stored procedures, you need to define a JCL startup procedure for the WLM environment, and define the application environment to WLM. See "Creating the WLM address space startup procedure for the DB2 Universal JDBC Driver stored procedures" and "Defining the WLM application environment for the the DB2 Universal JDBC Driver stored procedures."
2. Define the stored procedures to DB2, bind the stored procedure packages, and define the SYSIBM.SYSDUMMYU, SYSIBM.SYSDUMMYA, and SYSIBM.SYSDUMMYE tables. See "Defining the DB2 Universal JDBC Driver stored procedures to DB2 and creating the stored procedure packages" on page 264.

Creating the WLM address space startup procedure for the DB2 Universal JDBC Driver stored procedures

You can use the DSN8WLMP sample startup procedure as a model for your stored procedure address space startup procedure. Make the following changes to that procedure:

1. Change the APPLENV value to match the definition name that you specify in the WLM Definition Menu. See "Defining the WLM application environment for the the DB2 Universal JDBC Driver stored procedures."
2. Change the startup procedure name to match the procedure name that you specify in the WLM Create an Application Environment menu.
3. Change the DB2SSN value to the subsystem name of your DB2 UDB for z/OS subsystem.
4. Edit the data set names to match your data set names.

Defining the WLM application environment for the the DB2 Universal JDBC Driver stored procedures

To define the application environment to WLM, specify values similar to those that are shown on the following WLM panels.

```
  File  Utilities  Notes  Options  Help
-------------------------------------------------------------------------
                    Definition Menu     WLM Appl
Command ===> ____________________________________________________________

Definition data set . :  none
Definition name  . . . . WLMENV
Description . . . . . .  Environment for Development Center
Select one of the
following options. . .   9  1. Policies
                            2. Workloads
                            3. Resource Groups
                            4. Service Classes
                            5. Classification Groups
                            6. Classification Rules
                            7. Report Classes
                            8. Service Coefficients/Options
                            9. Application Environments
                           10. Scheduling Environments
```

Definition name
Specify the name of the WLM application environment that you are setting up for stored procedures. This value needs to match the APPLENV value in the WLM address space startup procedure.

Description
Specify any value.

Options
Specify 9 (Application Environments).

```
  Application-Environment  Notes  Options  Help
-------------------------------------------------------------------------
                    Create an Application Environment
Command ===> ____________________________________________________________

Application Environment Name . : WLMENV
Description  . . . . . . . . . . Environment for Development Center
Subsystem Type . . . . . . . . . DB2
Procedure Name . . . . . . . . . DSN8WLMP
Start Parameters . . . . . . . . DB2SSN=DB2T,NUMTCB=3,APPLENV=WLMENV
                                 ____________________________________
                                 ___________________________________

Limit on starting server address spaces for a subsystem instance:
1   1.  No limit.
    2.  Single address space per system.
    3.  Single address spaces per sysplex.
```

Subsystem Type
Specify DB2.

Procedure Name
This name must match the name of the JCL startup procedure for the stored procedure address spaces that are associated with this application environment.

Start Parameters
If the DB2 subsystem in which the stored procedure runs is not in a Sysplex, the DB2SSN value must match the name of that DB2 subsystem. If the same JCL is used for multiple DB2 subsystems, specify DB2SSN=&IWMSSNM.

The NUMTCB value depends on the type of stored procedure that you are running. The maximum value should be between 5 and 8.

The APPLENV value must match the value that you specify in the WLM address space startup procedure and on the CREATE PROCEDURE statements for the stored procedures. See "Defining the DB2 Universal JDBC Driver stored procedures to DB2 and creating the stored procedure packages."

Limit on starting server address spaces for a subsystem instance
Specify 1 (no limit).

Defining the DB2 Universal JDBC Driver stored procedures to DB2 and creating the stored procedure packages

DB2 provides statements that you can use to define the DB2–supplied stored procedures for JDBC, bind the stored procedure packages, and define the SYSIBM.SYSDUMMYU, SYSIBM.SYSDUMMYA, and SYSIBM.SYSDUMMYE tables. The statements are in these jobs:

DSNTIJSG
Use this job if you are defining the stored procedures and tables as part of installing or migrating a DB2 subsystem.

Before you run this job, you need to modify the WLM ENVIRONMENT parameter value for each stored procedure to match the Application Environment Name value that you specified in the WLM panels and the APPLENV name that you specified in the WLM address space startup procedure. Other customizations are made as part of the installation process.

DSNTIJMS
Use this job if you are defining the stored procedures and tables after you install or migrate a DB2 subsystem.

For DB2 Version 8, do not run this job until your DB2 subsystem is in new-function mode.

Before you run this job, you need to make the modifications that are described in the job prolog.

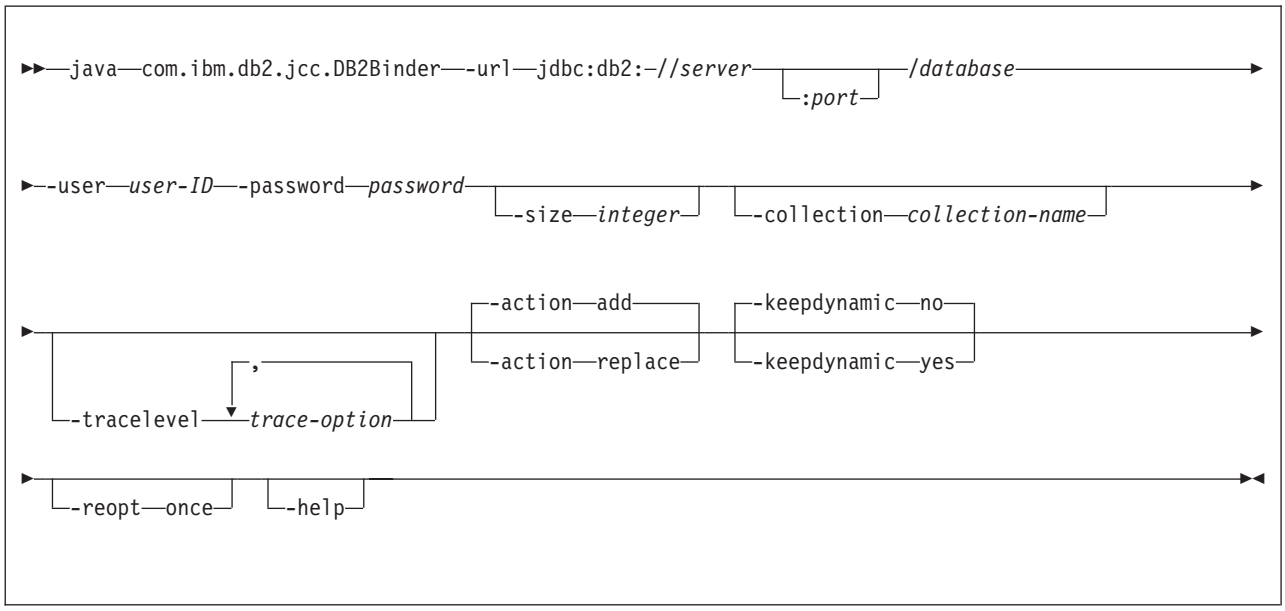
DSNTIJMC
Use this job if you are migrating to DB2 Version 8, and you defined the stored procedures and tables in a previous release of DB2. Do not run this job until your DB2 subsystem is in new-function mode.

Before you run this job, you need to make the modifications that are described in the job prolog.

Binding the packages for the DB2 Universal JDBC Driver

To bind the packages for the DB2 Universal JDBC Driver, run the DB2binder utility. This utility binds the packages and grants EXECUTE authority on the packages to PUBLIC.

DB2binder syntax



DB2Binder parameter descriptions

-url
Specifies the data source at which the DB2 Universal JDBC Driver packages are to be bound. The variable parts of the `-url` value are:

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

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;
```

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

-password
Specifes the password for the user ID.

-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 DB2 Universal JDBC Driver uses these packages to process dynamic SQL. In

addition, the DB2binder binds a single package that the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver. The default is NULLID. DB2binder translates this value to uppercase.

You can create multiple instances of the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver by setting the `currentPackageSet` property to a value that matches a -collection value. See "Properties for the DB2 Universal JDBC Driver" on page 185 for information on the `currentPackageSet` property.

-tracelevel
Specifies what to trace while DB2Binder runs. See the explanation of the `traceLevel` property in "Properties for the DB2 Universal JDBC Driver" on page 185 for the options that are available.

-action
Specifies whether the DB2 Universal JDBC Driver 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.

-keepdynamic
Specifies whether DB2 keeps already prepared dynamic SQL statements in the dynamic statement cache after commit points.

yes Indicates that DB2 keeps already prepared dynamic SQL statements in the dynamic statement cache after commit points.

no Indicates that DB2 does not keep already prepared dynamic SQL statements in the dynamic statement cache after commit points. This is the default.

If -keepdynamic is yes, dynamic statement caching can be done only if the EDM dynamic statement cache is enabled on the database server. The CACHEDYN subsystem parameter must be set to YES to enable the dynamic statement cache.

-keepdynamic is applicable only for DB2 UDB for z/OS database servers. See *DB2 Application Programming and SQL Guide* for more information on dynamic statement caching.

-reopt once
Specifies that DB2 determines and caches the access path for a dynamic statement only once at run time, or until the prepared statement is invalidated or removed from the dynamic statement cache and needs to be prepared again.

If -reopt once is not specified, the value is not set and is the default for the database server. -reopt once is applicable only for DB2 UDB for z/OS database servers.

-help
Specifies that the DB2binder utility describes each of the options that it supports. If any other options are specified with -help, they are ignored.

DB2Binder example

Bind the JDBC packages for a DB2 Universal JDBC Driver instance on the DB2 subsystem that has that has IP address mvs1, port number 446, and DB2 location name SJCEC1. Use the default collection name for the packages.

```
java com.ibm.db2.jcc.DB2Binder -url jdbc:db2://mvs1:446/SJCEC1 \
  -user SYSADM -password mypass
```

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

If you plan to implement distributed transactions using Universal Driver 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

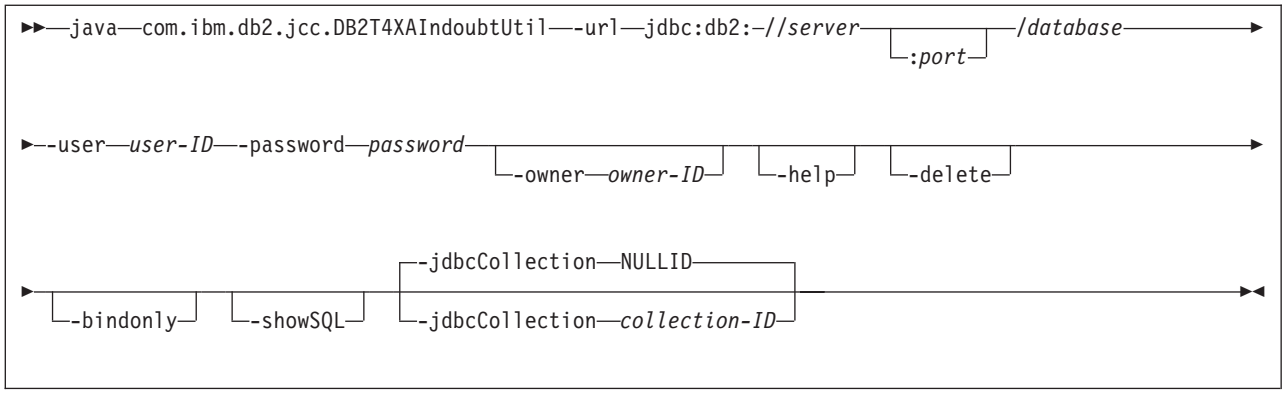
You should create and drop packages T4XAIN01, T4XAIN02, T4XAIN03, and T4XAIN04 only by running DB2T4XAIndoubtUtil. You can create and drop SYSTEM.INDOUBT and its index manually, but it is recommended that you use the utility. See "DB2T4XAIndoubtUtil usage notes" on page 269 for instructions on how to 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:



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 UDB 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*.

For information about the relationship between secondary authorization IDs and RACF groups, see *DB2 Administration Guide*.

-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 DB2 Universal JDBC Driver 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 UDB 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.

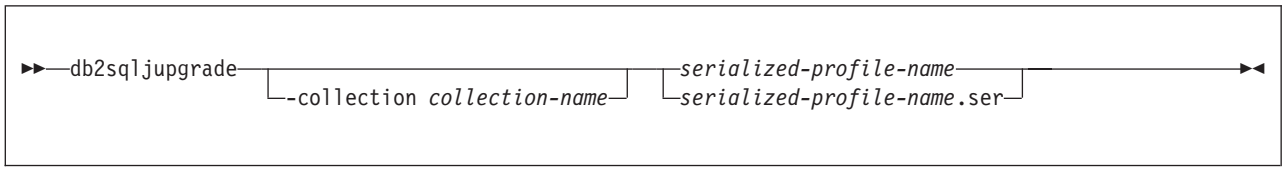
```
java com.ibm.db2.jcc.DB2T4XAIndoubtUtil -url jdbc:db2://mvs1:446/SJCEC1 \
  -user SYSADM -password mypass
```

Converting JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles for the DB2 Universal JDBC Driver

To convert serialized profiles that you customized under JDBC/SQLJ Driver for OS/390 and z/OS to a format that is compatible with the DB2 Universal JDBC Driver, run the db2sqljupgrade utility. After you run the db2sqljupgrade utility, you do not need to bind new packages for the associated SQLJ applications.

Before you can run the db2sqljupgrade utility, your CLASSPATH must contain the full path names for the db2j2classes.zip file for the JDBC/SQLJ Driver for OS/390 and z/OS, and the db2jcc.jar and sqlj.zip files for the DB2 Universal JDBC Driver.

db2sqljupgrade syntax



db2sqljupgrade parameter descriptions

-collection
Specifies the collection ID for the DB2 packages that were bound for the application that is associated with the JDBC/SQLJ Driver for OS/390 and z/OS serialized profile. This collection ID is stored in the converted serialized profile and is used as the qualifier for the DB2 packages for the application. The packages were created using the DB2 BIND command from DBRMs that were created when the db2profc command was run to create the serialized profile. The default is NULLID.

serialized-profile-name **or** *serialized-profile-name***.ser**
Specifies the name of the JDBC/SQLJ Driver for OS/390 and z/OS serialized profile that is to be converted to the DB2 Universal JDBC Driver format.

The db2sqljupgrade utility saves the original serialized profile as *serialized-profile-name*.ser_old.

db2sqljupgrade usage notes

You can use the following technique to find the correct -collection parameter value for a serialized profile:

1. Run the db2profp utility.
2. Locate the program name in the db2profp output. The program name is the stem for each of the four DBRMs that the SQLJ customizer produces for a serialized profile.

   For example, db2profp output from sample program Sample02.sqlj looks like this:

   ```
   ================================================
   printing contents of profile Sample02_SJProfile0
   created 1137709347170 (Thu Jan 19 14:22:27 PST 2006)
   DB2 consistency token is x'00000108E4C2F162'
   DB2 program version string is null
   DB2 program name is "SQLJ01"
   associated context is Sample02ctx
   profile loader is sqlj.runtime.profile.DefaultLoader@6a049a03
   contains 1 customizations
   COM.ibm.db2os390.sqlj.custom.DB2SQLJCustomizer@38f81a03
   original source file: null
   contains 2 entries
   ================================================
   ```

   The program name is SQLJ01, so the four DBRMs, and the packages into which they are bound, are SQLJ011, SQLJ012, SQLJ013, and SQLJ014.
3. Query catalog table SYSIBM.SYSPACKAGE to determine the collection ID that is associated with the four packages. For example:

```
SELECT NAME, COLLID
  FROM SYSIBM.SYSPACKAGE
WHERE NAME IN ('SQLJ011', 'SQLJ012', 'SQLJ013', 'SQLJ014')
```

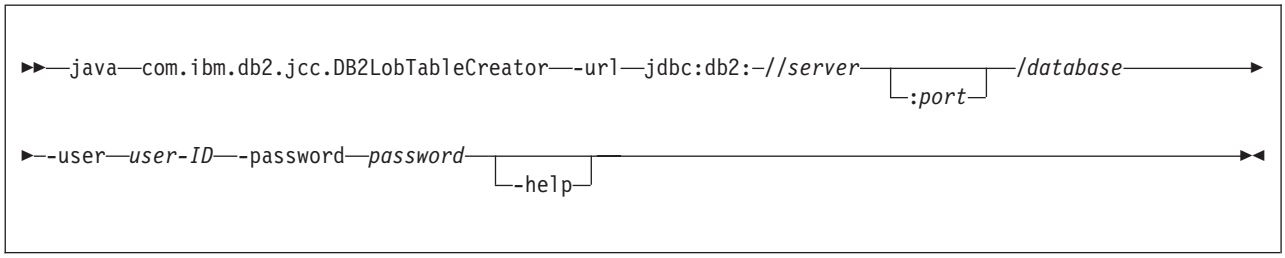
Enabling retrieval of DBCLOB columns with LOB locators on DB2 UDB for OS/390 and z/OS servers

If you plan to use LOB locators to retrieve data from DBCLOB columns on DB2 UDB for OS/390 and z/OS servers, you need to run the DB2LobTableCreator utility against those servers.

DB2LobTableCreator creates an EBCDIC table named SYSIBM.SYSDUMMYE, an ASCII table named SYSIBM.SYSDUMMYA, and a Unicode table named SYSIBM.SYSDUMMYU. You should create these objects only by running DB2LobTableCreator.

To run the DB2LobTableCreator utility, you need authority to create tables in the DSNATPDB database.

DB2LobTableCreator syntax



DB2LobTableCreator parameter 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 UDB 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.

DB2LobTableCreator example

Run the DB2LobTableCreator to allow LOB locators to retrieve data from DBCLOB columns in tables on a DB2 UDB for z/OS subsystem that has IP address mvs1, port number 446, and DB2 location name SJCEC1. User DBADM has authority to create tables in the DSNATPDB database.

```
java com.ibm.db2.jcc.DB2LobTableCreator -url jdbc:db2://mvs1:446/SJCEC1 \
  -user DBADM -password mypass
```

Verifying the installation of the DB2 Universal JDBC Driver

To verify the installation of the DB2 Universal JDBC Driver, compile and run a simple JDBC application.

If you installed the JDBC/SQLJ Driver for OS/390 and z/OS, you can modify the Sample01.java program to run with the DB2 Universal JDBC Driver. If the JDBC/SQLJ Driver for OS/390 and z/OS is installed in `/usr/lpp/db2810`, you can find Sample01.java in the following path:

```
/usr/lpp/db2810/samples
```

To modify Sample01.java for the DB2 Universal JDBC Driver:

1. Find this statement:

   ```
   Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
   ```

   Change it to this statement:

   ```
   Class.forName("com.ibm.db2.jcc.DB2Driver");
   ```

2. Find this statement:

   ```
   String URLprefix = "jdbc:db2os390sqlj:";
   ```

   Change it to this statement:

   ```
   String URLprefix = "jdbc:db2:";
   ```

If you did not install the JDBC/SQLJ Driver for OS/390 and z/OS, you can compile and run this program to verify your installation:

```
/**
 * File: TestJDBCSelect.java
 *
 * Purpose: Verify DB2 Universal JDBC Driver installation.
 *          This program uses Universal Driver type 2 connectivity
 *          on DB2 UDB for z/OS.
 *
 * Authorization: This program requires SELECT authority on
 *                DB2 catalog table SYSIBM.SYSTABLES.
 *
 * Flow:
 *   - Load the DB2 Universal JDBC Driver.
 *   - Get the driver version and display it.
 *   - Establish a connection to the local DB2 UDB for z/OS server.
 *   - Get the DB2 version and display it.
 *   - Execute a query against SYSIBM.SYSTABLES.
 *   - Clean up by closing all open objects.
 */

import java.sql.*;

public class TestJDBCSelect
{
  public static void main(String[] args)
  {
    try
    {
      // Load the driver and get the version
      System.out.println("\nLoading DB2 Universal JDBC Driver");
      Class.forName("com.ibm.db2.jcc.DB2Driver");
      System.out.println("  Successful load. Driver version: " +
        com.ibm.db2.jcc.DB2Version.getVersion());

      // Connect to the local DB2 UDB for z/OS server
      System.out.println("\nEstablishing connection to local server");
      Connection conn = DriverManager.getConnection("jdbc:db2:");
      System.out.println("  Successful connect");
      conn.setAutoCommit(false);

      // Use DatabaseMetaData to determine the DB2 version
      System.out.println("\nAcquiring DatabaseMetaData");
      DatabaseMetaData dbmd = conn.getMetaData();
      System.out.println("  DB2 version: " +
                         dbmd.getDatabaseProductVersion());

      // Create a Statement object for executing a query
      System.out.println("\nCreating Statement");
      Statement stmt = conn.createStatement();
      System.out.println("  successful creation of Statement");
```

Figure 67. Example of a JDBC program for verifying the DB2 Universal JDBC Driver installation (Part 1 of 2)

```
      // Execute the query and retrieve the ResultSet object
      String sqlText =
          "SELECT CREATOR, "       +
                 "NAME "           +
          "FROM SYSIBM.SYSTABLES " +
          "ORDER BY CREATOR, NAME";
      System.out.println("\nPreparing to execute SELECT");
      ResultSet results = stmt.executeQuery(sqlText);
      System.out.println("  Successful execution of SELECT");

      // Retrieve and display the rows from the ResultSet
      System.out.println("\nPreparing to fetch from ResultSet");
      int recCnt = 0;
      while(results.next())
      {
        String creator = results.getString("CREATOR");
        String name    = results.getString("NAME");
        System.out.println("CREATOR: <" + creator + "> NAME: <" + name + ">");

        recCnt++;
        if(recCnt == 10) break;
      }
      System.out.println("  Successful processing of ResultSet");

      // Close the ResultSet, Statement, and Connection objects
      System.out.println("\nPreparing to close ResultSet");
      results.close();
      System.out.println("  Successful close of ResultSet");

      System.out.println("\nPreparing to close Statement");
      stmt.close();
      System.out.println("  Successful close of Statement");

      System.out.println("\nPreparing to rollback Connection");
      conn.rollback();
      System.out.println("  Successful rollback");

      System.out.println("\nPreparing to close Connection");
      conn.close();
      System.out.println("  Successful close of Connection");
    }
    // Handle errors
    catch(ClassNotFoundException e)
    {
      System.err.println("Unable to load DB2 Universal Driver, " + e);
    }
    catch(SQLException e)
    {
      System.out.println("SQLException: " + e);
      e.printStackTrace();
    }
  }
}
```

Figure 67. Example of a JDBC program for verifying the DB2 Universal JDBC Driver installation (Part 2 of 2)

Installing the z/OS Application Connectivity to DB2 for z/OS feature

z/OS Application Connectivity to DB2 for z/OS is a DB2 UDB for OS/390 and z/OS or DB2 UDB for z/OS feature that allows Universal Driver type 4 connectivity from clients that do not have DB2 UDB for z/OS or DB2 UDB for OS/390 and z/OS installed to DB2 UDB for z/OS or DB2 UDB for Linux, UNIX and Windows servers. To install the z/OS Application Connectivity to DB2 for

z/OS, follow these steps. Unless otherwise noted, all steps apply to the z/OS system on which you are installing z/OS Application Connectivity to DB2 for z/OS.

1. Install Java 2 Technology Edition, SDK 1.3.1 or higher.
2. If you plan to connect to DB2 UDB for z/OS Version 7 servers, install OS/390 Support for Unicode or z/OS Support for Unicode on those servers. See Information APARs II13048 and II13049 for more information.
3. On the z/OS system on which you are installing z/OS Application Connectivity to DB2 for z/OS, and on any z/OS systems that contain DB2 servers to which you plan to connect, enable TCP/IP. See *IBM TCP/IP for MVS: Customization & Administration Guide*.
4. Allocate and load the z/OS Application Connectivity to DB2 for z/OS libraries. See "Loading the z/OS Application Connectivity to DB2 for z/OS libraries" on page 276 for details.
5. On all DB2 UDB for z/OS servers to which you plan to connect, enable distributed data facility (DDF) and TCP/IP support. See Part 3 of *DB2 Installation Guide*.
6. On all DB2 UDB for z/OS servers to which you plan to connect, set subsystem parameter DESCSTAT to YES. DESCSTAT corresponds to installation field DESCRIBE FOR STATIC on panel DSNTIPF. See Part 2 of *DB2 Installation Guide* for information on setting DESCSTAT. This step is necessary for SQLJ support.
7. In z/OS UNIX System Services, edit your .profile file to customize the environment variable settings. You use this step to set the libraries, paths, and files that the DB2 Universal JDBC Driver uses. See "Setting environment variables for the z/OS Application Connectivity to DB2 for z/OS feature" on page 276 for details.
8. On all DB2 UDB for z/OS servers to which you plan to connect, enable the DB2-supplied stored procedures that are used by the DB2 Universal JDBC Driver. See "Enabling the DB2-supplied stored procedures and defining the tables used by the DB2 Universal JDBC Driver" on page 261.
9. In z/OS UNIX System Services, run the DB2binder utility on the z/OS system on which you are installing z/OS Application Connectivity to DB2 for z/OS to bind the packages for the DB2 Universal JDBC Driver at all DB2 UDB for z/OS servers to which you plan to connect. You need to run DB2binder once for each server. See "Binding the packages for the DB2 Universal JDBC Driver" on page 264.
10. ***If you plan to use Universal Driver type 4 connectivity to implement distributed transactions against DB2 UDB for OS/390 and z/OS Version 7 servers:*** In z/OS UNIX System Services, run the DB2T4XAIndoubtUtil utility on the z/OS system on which you are installing z/OS Application Connectivity to DB2 for z/OS. Run the utility once for each of the DB2 UDB for OS/390 and z/OS Version 7 servers. See "DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers" on page 267 for details.
11. ***If you plan to use LOB locators to access DBCLOB columns in DB2 tables on DB2 UDB for z/OS servers:*** In z/OS UNIX System Services, run the DB2LobTableCreator utility on each of those servers to create tables that are needed for fetching LOB locators. See "Enabling retrieval of DBCLOB columns with LOB locators on DB2 UDB for OS/390 and z/OS servers" on page 271.

Loading the z/OS Application Connectivity to DB2 for z/OS libraries

To allocate the HFS directory structure and use SMP/E to load the z/OS Application Connectivity to DB2 for z/OS libraries, run the following jobs:

DDAALA
Creates the SMP/E consolidate software inventory (CSI) file. DDAALA is required only if the SMP/E target and distribution zones are not created and allocated to the SMP/E global zone.

DDAALB
Creates the z/OS Application Connectivity to DB2 for z/OS target and distribution zones. Also creates DDDEFs for SMP/E data sets. DDAALB is required only if the SMP/E target and distribution zones are not created and allocated to the SMP/E global zone.

DDAALLOC
Creates the z/OS Application Connectivity to DB2 for z/OS target and distribution libraries and defines them in the SMP/E target and distribution zones.

DDADDDEF
Creates DDDEFs for the z/OS Application Connectivity to DB2 for z/OS target and distribution libraries.

DDAISMKD
Invokes the DDAMKDIR EXEC to allocate the HFS directory structure for the z/OS Application Connectivity to DB2 for z/OS.

DDARECEV
Performs the SMP/E RECEIVE function for the z/OS Application Connectivity to DB2 for z/OS libraries.

DDAAPPLY
Performs the SMP/E APPLY CHECK and APPLY functions for the z/OS Application Connectivity to DB2 for z/OS libraries.

DDAACCEP
Performs the SMP/E ACCEPT CHECK and ACCEPT functions for the z/OS Application Connectivity to DB2 for z/OS libraries.

See *z/OS Application Connectivity to DB2 for z/OS Program Directory* for information on allocating and loading z/OS Application Connectivity to DB2 for z/OS data sets.

Setting environment variables for the z/OS Application Connectivity to DB2 for z/OS feature

The environment variables that you must set are:

PATH
Modify PATH to include the directory that contains the shell scripts that invoke DB2 Universal JDBC Driver program preparation and debugging functions. If z/OS Application Connectivity to DB2 for z/OS is installed in `/usr/lpp/jcct4`, modify PATH as follows:

```
export PATH=/usr/lpp/jcct4/bin:$PATH
```

CLASSPATH
z/OS Application Connectivity to DB2 for z/OS contains the following class files:

db2jcc.jar
Contains all JDBC classes and the SQLJ runtime classes for Universal Driver type 4 connectivity.

db2jcc_javax.jar
Contains a subset of the J2EE classes that are needed for Universal Driver type 4 connectivity.

sqlj.zip
Contains the classes that are needed to prepare SQLJ applications for execution under the DB2 Universal JDBC Driver.

db2jcc_license_cisuz.jar
A license file that permits access to DB2 UDB servers.

Modify your CLASSPATH to include these files. If z/OS Application Connectivity to DB2 for z/OS is installed in `/usr/lpp/jcct4`, modify CLASSPATH as follows:

```
export CLASSPATH=/usr/lpp/jcct4/classes/db2jcc.jar: \
/usr/lpp/jcct4/classes/db2jcc_javax.jar: \
/usr/lpp/jcct4/classes/sqlj.zip: \
/usr/lpp/jcct4/classes/db2jcc_license_cisuz.jar: \
$CLASSPATH
```

Important: Do not include class files for both the DB2 Universal JDBC Driver and the JDBC/SQLJ Driver for OS/390 and z/OS in your CLASSPATH. The only exception to this rule is that you need to include classes for both drivers in your CLASSPATH while you convert serialized profiles that you customized under the JDBC/SQLJ Driver for OS/390 and z/OS to the format for the DB2 Universal JDBC Driver. See "Converting JDBC/SQLJ Driver for OS/390 and z/OS serialized profiles for the DB2 Universal JDBC Driver" on page 269.

If you use Java stored procedures, you need to set additional environment variables in a JAVAENV data set. See "Setting the run-time environment for interpreted Java stored procedures" on page 202 for more information.

Chapter 8. Installing the JDBC/SQLJ Driver for OS/390 and z/OS

To install the JDBC/SQLJ Driver for OS/390 and z/OS, follow these steps:

1. Install Java 2 Technology Edition, SDK 1.3.1 or higher. If you plan to implement Java stored procedures and user-defined functions on this DB2 subsystem, install Java 2 Technology Edition, SDK 1.3.1, SDK 1.4.1, or higher.
2. When you allocate and load the DB2 libraries, include the steps that allocate and load the JDBC and SQLJ libraries. See "Loading the JDBC and SQLJ libraries" for details.
3. Set DB2 subsystem parameters for SQLJ support. See "Setting DB2 subsystem parameters for SQLJ support" on page 280 for details.
4. Log on to TSO.

   Specify a maximum region size of at least 200000.

   Ensure that you have superuser authority (UID 0).
5. In z/OS UNIX System Services, edit your .profile file to customize the environment variable settings. You use this step to set the libraries, paths, and files that JDBC and SQLJ use, and to indicate which JDBC driver you want to use. See "Setting environment variables for the JDBC/SQLJ Driver for OS/390 and z/OS" on page 280 for details.
6. ***Optional:*** In z/OS UNIX System Services, customize the SQLJ/JDBC run-time properties file. See "The SQLJ/JDBC run-time properties file" on page 281 for details.

   The default path name is /usr/lpp/db2810/classes/db2sqljjdbc.properties. If you use a new path name for your customized run-time properties file, you need to specify that file name in the DB2SQLJPROPERTIES environment variable.
7. ***Optional:*** Run the db2genJDBC utility in z/OS UNIX System Services to customize JDBC resources. You do not need to perform this step unless you need to alter the default JDBC resource limits. See "Customizing the JDBC profile (optional)" on page 286 for details.
8. Prepare the JDBC DBRMs for execution.

   If you did not run the db2genJDBC utility, these are the DBRMs in the DSN810.SDSNDBRM data set. If you ran the db2genJDBC utility, these are the DBRMs that the db2genJDBC utility produces.

   In TSO, customize and run job DSNTJJCL to bind the JDBC DBRMs into packages, and bind the packages into the JDBC plan. DSNTJJCL is in data set DSN810.SDSNSAMP. See "Binding the DBRMs" on page 287 for details.

   In TSO, grant EXECUTE authority on the packages and plan to PUBLIC.
9. Verify the installation by running a simple JDBC application. See "Verifying the installation of the JDBC/SQLJ Driver for OS/390 and z/OS" on page 288 for suggestions on how to do that.

Loading the JDBC and SQLJ libraries

When you install DB2, include the steps for allocating the HFS directory structure and using SMP/E to load the JDBC and SQLJ libraries. The jobs that perform these functions are:

DSNISMKD
Invokes the DSNMKDIR EXEC to allocate the HFS directory structure.

DSNDDEF2
Includes steps to define DDDEFs for the JDBC and SQLJ libraries.

DSNRECV3
Includes steps that perform the SMP/E RECEIVE function for the JDBC and SQLJ libraries.

DSNAPPL2
Includes the steps that perform the SMP/E APPLY CHECK and APPLY functions for the JDBC and SQLJ libraries.

DSNACEP2
Includes the steps that perform the SMP/E ACCEPT CHECK and ACCEPT functions for the JDBC and SQLJ libraries.

See *IBM DATABASE 2 Universal Database for z/OS Program Directory* for information on allocating and loading DB2 data sets.

Setting DB2 subsystem parameters for SQLJ support

The DESCRIBE FOR STATIC field on DB2 installation panel DSNTIPF sets subsystem parameter DESCSTAT, which controls whether DB2 executes DESCRIBEs on static SQL statements when it performs a bind operation. If you use named iterators in your SQLJ programs, and you do not use online checking, DESCRIBE FOR STATIC must be set to YES. See Part 2 of *DB2 Installation Guide* for information on setting the DESCRIBE FOR STATIC. See "Using a named iterator in an SQLJ application" on page 74 for information on named iterators. See "Customizing an SQLJ serialized profile under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 242 for information on online checking.

Setting environment variables for the JDBC/SQLJ Driver for OS/390 and z/OS

The environment variables that you must set are:

STEPLIB
Modify STEPLIB to include the `SDSNEXIT`, `SDSNLOAD`, and `SDSNLOD2` data sets. For example:

```
export STEPLIB=DSN810.SDSNEXIT:DSN810.SDSNLOAD:DSN810.SDSNLOD2:$STEPLIB
```

PATH
Modify PATH to include the directory that contains the shell scripts that invoke JDBC and SQLJ program preparation and debugging functions. If JDBC and SQLJ are installed in `/usr/lpp/db2810`, modify PATH as follows:

```
export PATH=/usr/lpp/db2810/bin:$PATH
```

The PATH environment variable is not used in the CICS environment.

LIBPATH
The DB2 UDB for z/OS JDBC/SQLJ Driver for OS/390 and z/OS contains several dynamic load libraries (DLLs).

Modify LIBPATH to include the directory that contains these DLLs. If SQLJ and JDBC are installed in `/usr/lpp/db2810`, modify LIBPATH as follows:

```
export LIBPATH=/usr:/usr/lib:/usr/lpp/db2810/lib:$LIBPATH
```

CLASSPATH
Modify the CLASSPATH to include the following file:

db2j2classes.zip
Contains all of the classes necessary to prepare and run JDBC and SQLJ programs with the JDBC 2.0 driver. Assuming that JDBC and SQLJ are installed in `/usr/lpp/db2810`, modify CLASSPATH as follows:

```
export CLASSPATH=/usr/lpp/db2810/classes/db2j2classes.zip:$CLASSPATH
```

DB2SQLJPROPERTIES
Specifies the fully-qualified name of the run-time properties file for the JDBC/SQLJ Driver for OS/390 and z/OS. The run-time properties file contains various entries of the form *parameter=value* that specify program preparation and run-time options that the DB2 UDB for z/OS JDBC/SQLJ Driver for OS/390 and z/OS uses. The run-time properties file is read when the driver is loaded. If you do not set the DB2SQLJPROPERTIES environment variable, the driver uses the default name `./db2sqljjdbc.properties`.

For example, to use a run-time properties file named `db2sqljjdbc.properties` that is in the `/usr/lpp/db2810/classes` directory, specify:

```
export DB2SQLJPROPERTIES=/usr/lpp/db2810/classes/db2sqljjdbc.properties
```

If you use Java stored procedures, you need to set additional environment variables in a JAVAENV data set. See "Setting the run-time environment for interpreted Java stored procedures" on page 202 for more information.

The SQLJ/JDBC run-time properties file

The SQLJ/JDBC run-time properties file contains settings for the JDBC/SQLJ Driver for OS/390 and z/OS. The SQLJ/JDBC run-time properties file is a text file in which each line is of this form:

```
property=value
```

See "Properties in the JDBC/SQLJ Driver for OS/390 and z/OS SQLJ/JDBC run-time properties file" for a list of properties that you can specify.

The JDBC/SQLJ Driver for OS/390 and z/OS determines the run-time properties file to use in the following way:

1. If the DB2SQLJPROPERTIES environment variable is set, the driver uses the path name that is in this environment variable.
2. If the DB2SQLJPROPERTIES environment variable is not set, the driver looks in the current working directory for a file that is named db2sqljjdbc.properties.
3. If there is no file in the current working directory named db2sqljjdbc.properties, the driver uses default values for all properties that can be set in the run-time properties file.

For the CICS environment, the settings for some of the run-time properties are different than for other environments. See "Special considerations for CICS applications," on page 329 for information that is specific to CICS.

Properties in the JDBC/SQLJ Driver for OS/390 and z/OS SQLJ/JDBC run-time properties file

You can set any of the following properties in the SQLJ/JDBC run-time properties file.

DB2ACCTINTERVAL
Specifies whether DB2 accounting records are produced at commit points or on termination of the physical connection to the data source. If the value of DB2ACCTINTERVAL is COMMIT, DB2 accounting records are produced at commit points. For example:

```
DB2ACCTINTERVAL=COMMIT
```

Otherwise, accounting records are produced on termination of the physical connection to the data source.

DB2ACCTINTERVAL is not applicable to connections under CICS or IMS, or for Java stored procedures.

DB2SQLJDBRMLIB
Specifies the fully-qualified name of the z/OS partitioned data set into which DBRMs are placed. DBRMs are generated by the creation of a JDBC profile and the customization step of the SQLJ program preparation process. For example:

```
DB2SQLJDBRMLIB=USER.DBRMLIB.DATA
```

The default DBRM data set name is *prefix*.DBRMLIB.DATA, where *prefix* is the high-level qualifier that was specified in the TSO profile for the user. *prefix* is usually the user's TSO user ID.

If the DBRM data set does not already exist, you need to create it. The DBRM data set requires space to hold all the SQL statements, with additional space for each host variable name and some header information. The header information requires approximately two records for each DBRM, 20 bytes for each SQL record, and 6 bytes for each host variable. For an exact format of the DBRM, see the DBRM mapping macro, DSNXDBRM in library DSN810.SDSNMACS. The DCB attributes of the DBRM data set are RECFM FB and LRECL 80.

See "Customizing the JDBC profile (optional)" on page 286 and "Customizing an SQLJ serialized profile under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 242 for more information on serialized profile customization.

DB2SQLJ_DISABLE_JTRACE
Specifies whether to disable Java-side tracing. Possible values are:

1 Disable Java-side tracing.

0 Do not disable Java-side tracing. 0 is the default.

DB2SQLJ_DISABLE_JTRACE_TIMESTAMP
Specifies whether to exclude timestamps from Java-side traces. Possible values are:

1 Exclude timestamps from Java-side traces. Use this option to decrease the size of the trace output.

0 Include timestamps in Java-side traces. 0 is the default.

DB2SQLJ_LOCAL_LOCATION_NAME
Specifies the local DB2 location name. The JDBC/SQLJ Driver for OS/390 and z/OS uses this value instead of the DB2 LOCATION NAME installation panel field to determine whether a connection is to the local DB2 subsystem. This property is optional. If it is specified, it must match the DB2 LOCATION NAME value.

DB2SQLJPLANNAME
Specifies the name of the plan that is associated with a JDBC or an SQLJ application. The plan is created by the DB2 UDB for z/OS bind process. For example:

```
DB2SQLJPLANNAME=SQLJPLAN
```

The default name is DSNJDBC.

DB2SQLJJDBCPROGRAM
Specifies the name of the JDBC profile that is used by the JDBC/SQLJ Driver for OS/390 and z/OS. For example:

```
DB2SQLJJDBCPROGRAM=CONNPROF
```

The default connected profile name is DSNJDBC.

See "Customizing the JDBC profile (optional)" on page 286 for information on creating a JDBC connected profile.

DB2SQLJSSID
Specifies the name of the DB2 subsystem to which a JDBC or an SQLJ application connects. For example:

```
DB2SQLJSSID=DSN
```

If you do not specify the DB2SQLJSSID property, the JDBC/SQLJ Driver for OS/390 and z/OS uses the SSID value from the DSNHDECP data-only load module. When you install DB2 UDB for z/OS, a DSNHDECP module is created in the *prefix*.SDSNEXIT data set and the *prefix*.SDSNLOAD data set. Other DSNHDECP load modules might be created in other data sets for selected applications.

The JDBC/SQLJ Driver for OS/390 and z/OS must load a DSNHDECP module before it can read the SSID value. z/OS searches data sets in the following places, and in the following order, for the DSNHDECP module:
1. Job pack area (JPA)
2. TASKLIB
3. STEPLIB or JOBLIB
4. LPA
5. Libraries in the link list

You need to ensure that if your system has more than one copy of the DSNHDECP module, z/OS finds the data set that contains the correct copy for the JDBC/SQLJ Driver for OS/390 and z/OS first.

DB2SQLJMULTICONTEXT
Specifies whether each connection in an application is independent of other connections in the application, and each connection is a separate unit of work, with its own commit scope. The value can be YES or NO. For example:

```
DB2SQLJMULTICONTEXT=YES
```

The default is YES.

See Chapter 13, "Multiple z/OS context support in JDBC/SQLJ Driver for OS/390 and z/OS," on page 309 for more information on multiple z/OS context support.

DB2CURSORHOLD
For JDBC, specifies the effect of a commit operation on open DB2 cursors (ResultSets). The value can be YES or NO. A value of YES means that cursors

are not destroyed when the transaction is committed. A value of NO means that cursors are destroyed when the transaction is committed. For example:

```
DB2CURSORHOLD=NO
```

The default is YES.

This parameter does not affect cursors in a transaction that is rolled back. All cursors are destroyed when a transaction is rolled back.

db2.connpool.max.size

Specifies the maximum number of concurrent physical connections (DB2 threads) that the driver maintains in the connection pool. For example:

```
db2.connpool.max.size=200
```

The default is 100.

When this limit is reached, no new connections are added to the pool. If a logical connection is closed, and the pool is at the maximum size, the driver closes the underlying physical connection.

db2.connpool.idle.timeout

Specifies the minimum number of seconds that an unused physical connection remains in the connection pool before the thread is closed. For example:

```
db2.connpool.idle.timeout=300
```

The default is 600.

Specifying a value of zero disables idle connection timeout.

db2.connpool.connect.create.timeout

Specifies maximum number of seconds that a `DataSource` object waits for a connection to a data source. This value is used when the loginTimeout property for the `DataSource` object has a value of 0. For example:

```
db2.connpool.connect.create.timeout=300
```

The default is 0.

A value of zero disables connection creation timeout.

db2.jdbc.profile.pathname

Specifies the path name that the JDBC driver uses to locate and load the JDBC # profile. For example:

```
# db2.jdbc.profile.pathname=/usr/lpp/db2710/classes/DSNJDBC_JDBCProfile.ser
```

If db2.jdbc.profile.pathname is not set, the JDBC driver attempts to load the # JDBC profile as a system resource. If that fails, the driver searches the # CLASSPATH for the JDBC profile.

You must specify db2.jdbc.profile.pathname if you are using WebSphere # Application Server Version 5.0 or later.

db2.sqlj.profile.caching

Specifies whether serialized profiles are cached when the JVM under which their application is running is reset. db2.sqlj.profile.caching applies only to applications that run in a resettable JVM (applications that run in the CICS, IMS, or Java stored procedure environment). Possible values are:

NO SQLJ serialized profiles are not cached every time the JVM is reset, so

that new versions of the serialized profiles are loaded when the JVM is reset. Use this option when an application is under development, and new versions of the application and its serialized profiles are produced frequently.

YES SQLJ serialized profiles are cached when the JVM is reset. YES is the default.

db2.sp.lob.output.parm.size
Specifies the number of bytes of storage that the JDBC driver needs to allocate for output LOB values when the driver cannot determine the size of those LOBs. This situation occurs for LOB stored procedure output parameters.

The default value for db2.sp.lob.output.parm.size is 1048576. For systems with storage limitations and smaller LOBs, set the db2.sp.lob.output.parm.size value to a lower number.

For example, if you know that the output LOB size is at most 64000, set db2.sp.lob.output.parm.size to 64000.

db2.sp.varchar.output.parm.override
Specifies whether the JDBC driver changes a JDBC VARCHAR argument in a `CallableStatement.registerOutParameter` call to a JDBC LONGVARCHAR data type. Possible values are YES and NO. The default is NO.

A value of YES is useful for applications that are ported from platforms in which the JDBC VARCHAR data type is mapped to an SQL VARCHAR data type that is greater than 256 characters.

DB2SQLJ_TRACE_DUMP_FREQ
Specifies the frequency with which the internal native trace buffer is flushed to disk. DB2SQLJ_TRACE_DUMP_FREQ applies only to native-side tracing, and is applicable only if tracing is enabled. The value is the number of trace entries that are written before the trace buffer is flushed to disk. A value of 1 means that the trace buffer is flushed after each entry is written.

You should set DB2SQLJ_TRACE_DUMP_FREQ only under the direction of IBM Software Support.

DB2SQLJ_TRACE_FILENAME
Enables the SQLJ/JDBC trace and specifies the names of the trace files to which the trace is written. This parameter is required for collecting trace data. For example, specifying the following setting for DB2SQLJ_TRACE_FILENAME enables the SQLJ/JDBC trace to two files named /SYSTEM/tmp/jdbctrace and /SYSTEM/tmp/jdbctrace.JTRACE:

```
DB2SQLJ_TRACE_FILENAME=/SYSTEM/tmp/jdbctrace
```

See “Formatting trace data with the JDBC/SQLJ Driver for OS/390 and z/OS” on page 326 for more information on the SQLJ/JDBC trace.

You should set DB2SQLJ_TRACE_FILENAME only under the direction of IBM Software Support. See “Formatting trace data with the JDBC/SQLJ Driver for OS/390 and z/OS” on page 326 for information on formatting trace data.

DB2SQLJ_TRACE_BUFFERSIZE
Specifies the size of the trace buffer in virtual storage in kilobytes. SQLJ rounds the number that you specify down to a multiple of 64 KB. The default is 256 KB. This is an optional parameter. For example:

```
DB2SQLJ_TRACE_BUFFERSIZE=1024
```

You should set DB2SQLJ_TRACE_BUFFERSIZE only under the direction of IBM Software Support. See "Formatting trace data with the JDBC/SQLJ Driver for OS/390 and z/OS" on page 326 for information on formatting trace data.

DB2SQLJ_TRACE_WRAP
Enables or disables wrapping of the SQLJ trace. DB2J_TRACE_WRAP can have one of the following values:

1 Wrap the trace
0 Do not wrap the trace

The default is 1. This parameter is optional. For example:

```
DB2SQLJ_TRACE_WRAP=0
```

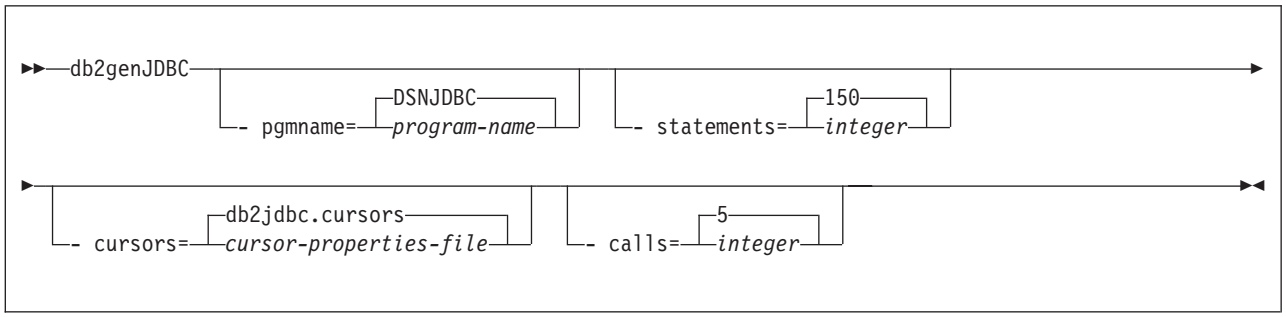
You should set DB2SQLJ_TRACE_WRAP only under the direction of IBM Software Support. See "Formatting trace data with the JDBC/SQLJ Driver for OS/390 and z/OS" on page 326 for information on formatting trace data.

DB2SQLJ_USE_CCSID420_SHAPED_CONVERTER
Specifies whether the JDBC driver uses the Java Cp420 shaped converter or the Cp420S shaped converter for conversion of CCSID 420 data. Possible values are 0 (for Cp420) or 1 (for Cp420S).) 0 is the default.

Customizing the JDBC profile (optional)

The JDBC profile that DB2 provides is sufficient for most installations. If you need additional resources for JDBC, you can run the db2genJDBC utility to customize JDBC resources.

Syntax



Parameter descriptions

-pgmname
Specifies the JDBC program name. This name must be seven or fewer characters in length. The default is DSNJDBC.

-statements
Specifes the number of sections to reserve in the DBRMs for JDBC statements and prepared statements for non-result set processing. The default is 150.

For CICS applications, you should not use the default value. See "Special considerations for CICS applications," on page 329 for more information.

-cursors
Specifies the name of the cursor properties file. The default is db2jdbc.cursors.

The file name must be either the fully-qualified file name, or the file name relative to the current working directory.

The cursor properties file must be located in a directory that is specified in the CLASSPATH environment variable, described in "Setting environment variables for the JDBC/SQLJ Driver for OS/390 and z/OS" on page 280.

If you do not use the default cursor properties file, you need to modify the contents of the file *before* you run db2genJDBC. The cursor properties file defines cursors that DB2 uses to retrieve rows from JDBC `ResultSets`. You can customize the cursor properties file to modify the number of DB2 cursors available for JDBC and to control cursor names. The default cursor properties file defines 100 cursors with the WITH HOLD attribute, and 100 cursors without the WITH HOLD attribute.

For CICS applications, you should not use the default value. See "Special considerations for CICS applications," on page 329 for more information.

-calls
Specifes the number of sections to reserve in the DBRMs for JDBC callable statements for non-result set processing. The default is 5.

Output

The db2genJDBC utility creates four DBRMs and a JDBC serialized profile. The JDBC profile must be located in a directory that is specified in the CLASSPATH environment variable, or the path for the JDBC profile must be specified in the SQLJ/JDBC run-time properties file, with the db2.jdbc.profile.pathname property.

The JDBC profile name is in the following format:

program-name`_JDBCProfile.ser`

Binding the DBRMs

Customize and run job DSNTJJCL to bind the JDBC DBRMs into packages and bind the packages into the DSNJDBC plan. DSNTJJCL is shipped in the DB2 DSN810.SDSNSAMP data set. If you did not run the db2genJDBC utility, the JDBC DBRMs are in the DB2 DSN810.SDSNDBRM data set. If you ran the db2genJDBC utility, these DBRMs are in the data set whose name you specified for the DB2SQLJDBRMLIB property.

The DBRM names and isolation levels are shown in Table 52. *program-name* is DSNJDBC, or the name that you specified for the -pgmname parameter when you ran db2genJDBC. The default transaction level for the DSNJDBC plan is CS.

Table 52. JDBC DBRM names and package isolation levels

DBRM name	Isolation level
*program-name*1	UR
*program-name*2	CS
*program-name*3	RS
*program-name*4	RR

The default transaction level for the DSNJDBC plan is CS. To change the transaction level of a connection in a JDBC program, use the `Connection.setTransactionIsolation` method.

For SQLJ applications, you need to include the JDBC packages in every SQLJ application plan.

Verifying the installation of the JDBC/SQLJ Driver for OS/390 and z/OS

To help you verify the installation of the JDBC/SQLJ Driver for OS/390 and z/OS and to get you started on writing your own JDBC and SQLJ applications, DB2 UDB for z/OS provides sample JDBC program Sample01.java and sample SQLJ program Sample02.sqlj.

The sample applications are designed to run under the JDBC/SQLJ Driver for OS/390 and z/OS.

Sample01.java demonstrates the following techniques:

- Connecting to a data source using the `DriverManager` interface
- Retrieving data using the `ResultSet` interface
- Processing errors using the DB2 UDB for z/OS-only `SQLException` interface

Sample02.sqlj demonstrates the following techniques:

- Connecting to a data source using the `DriverManager` interface
- Retrieving data using a named iterator
- Processing errors using the DB2 UDB for z/OS-only `SQLException` interface

If your SQLJ driver is installed in `/usr/lpp/db2810`, you can find Sample01.java and Sample02.sqlj in the following path:

```
/usr/lpp/db2810/samples
```

Chapter 9. JDBC and SQLJ security

The following topics provide information on security mechanisms that are available under the JDBC drivers:

Security under the DB2 Universal JDBC Driver

When you use the DB2 Universal JDBC Driver, 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 53 lists the security mechanisms that the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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 53. Security mechanisms supported by the DB2 Universal JDBC Driver

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`

Table 53. Security mechanisms supported by the DB2 Universal JDBC Driver (continued)

Security mechanism	securityMechanism property value
# Encrypted user ID, encrypted # password, and encrypted # security-sensitive data	DB2BaseDataSource.ENCRYPTED_USER_PASSWORD_AND_DATA_SECURITY
Kerberos[1]	DB2BaseDataSource.KERBEROS_SECURITY

Note:

1. Available for Universal Driver type 4 connectivity only.

User ID and password security under the DB2 Universal JDBC Driver

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 2.0
...
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 2.0
...
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
```

Universal Driver type 2 connectivity with no user ID or password: For Universal Driver type 2 connectivity, if you use user ID and password security, but you do not specify a user ID and password, DB2 uses the external security environment, such as the RACF security environment, that was previously established for the user. For a CICS® connection, you cannot specify a user ID or password.

User ID-only security under the DB2 Universal JDBC Driver

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 2.0
...
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 2.0
...
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 user ID security and encrypted password security under the DB2 Universal JDBC Driver

If you use encrypted user ID security or encrypted password security when you access a DB2® for z/OS® server, the Java™ Cryptography Extension, IBMJCE for z/OS needs to be enabled on the server. The Java Cryptography Extension is part of the IBM® Developer Kit for OS/390®, Java 2 Technology Edition, or the IBM Developer Kit for z/OS, Java 2 Technology Edition. For information on how to enable IBMJCE, go to this URL on the Web:

`http://www.ibm.com/servers/eserver/zseries/software/java/aboutj2.html`

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

Before you can use encrypted security-sensitive data, the z/OS Integrated Cryptographic Services Facility needs to be installed and enabled on the z/OS® operating system.

To specify encrypted user ID or encrypted password security for a JDBC connection, use one of the following techniques.

*For the **DriverManager** interface:* Set the user ID, password, and security mechanism by setting the user, password, and securityMechanism properties in a Properties object, and then invoking the form of the getConnection method that includes the Properties object as a parameter. For example, use code like this to set the user ID and encrypted password security mechanism:

```
import java.sql.*;                          // JDBC base
import com.ibm.db2.jcc.*;                   // DB2 implementation of JDBC 2.0
...
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 2.0
...
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
                                            // encrypted user ID and password
```

Kerberos security under the DB2 Universal JDBC Driver

Kerberos security is available for Universal Driver type 4 connectivity only.

If you use Kerberos security when you access a DB2® for z/OS® server, you need to install and configure the following products, or their equivalents:

- The SecureWay® Security Server for z/OS and OS/390®
- OS/390 SecureWay Security Server Network Authentication and Privacy Service, which is a component of the OS/390 SecureWay Security Server

  This is the IBM® OS/390 implementation of Kerberos Version 5.

  For more information, see *OS/390 SecureWay Server Network Authentication and Privacy Service Administration*.

You also need to enable the following components of the IBM Developer Kit for OS/390, Java™ 2 Technology Edition, or the IBM Developer Kit for z/OS, Java 2 Technology Edition:

- Java Cryptography Extension (IBMJCE) for OS/390
- IBM Java Generic Security Service (IBMJGSS)
- Java Authentication and Authorization Service (JAAS) for OS/390

For information on how to enable these components, go to this URL on the Web:
`http://www.ibm.com/servers/eserver/zseries/software/java/aboutj2.html`

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 address of the Kerberos server for the realm in which the client is registered.

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 2.0
...
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", "kdcsrv1.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 2.0
...
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("kdcsrv1.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 2.0
...
Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "kdcsrv1.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 2.0
...
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("kdcsrv1.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. Because the `gssCredential` property is not a string, you cannot use the `Properties.put` method to set it. Instead, use the `DB2BaseDataSource.setGSSCredential` method. 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 2.0
...
Properties properties = new Properties(); // Create a Properties object
properties.put("kerberosServerPrincipal", "kdcsrv1.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("kdcsrv1.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
```

Security for preparing SQLJ applications with the DB2 Universal JDBC Driver

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 DB2 subsystem. 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 DB2 subsystem 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 writing 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.

Security under the JDBC/SQLJ Driver for OS/390 and z/OS

This topic describes the security model for the JDBC/SQLJ Driver for OS/390 and z/OS. It explains how authorization IDs are determined and how the choice of DB2 attachment facility affects security.

Determining an authorization ID with the JDBC/SQLJ Driver for OS/390 and z/OS

With the JDBC/SQLJ Driver for OS/390 and z/OS, the method that DB2 uses to determine the SQL Authorization ID to use for a connection depends on whether you provide user ID and password values for the connection.

- If you do not provide a user ID and password, the JDBC driver uses the external security environment that is associated with the thread to establish the DB2 authorization ID.
- If you provide a user ID and password, the JDBC driver passes these values to DB2 for validation, and uses these values for the connection.

DB2 attachment types and security

The security environment (the RACF ACEE) that DB2 uses to establish the DB2 authorization IDs is dependent on which DB2 attachment type you use. JDBC and SQLJ use a DB2 attachment facility to communicate with DB2. They use the RRS attachment facility (RRSAF) or the CICS attachment facility.

All attachment types support multithreading, that is, multiple, concurrent threads (TCBs) that execute within a single process (address space). In a multithreading environment, each process and thread can have its own unique security environment. The DB2 attachment facility that you select determines which security environment DB2 uses to verify the DB2 authorization IDs.

See “Special considerations for CICS applications,” on page 329 for information on using the CICS attachment facility.

The DB2 RRS attachment facility (RRSAF) supports multithreading, and applications can run under multiple authorization IDs. If you use the RRSAF, DB2 uses a task-level security environment, if present, to establish the DB2 authorization IDs.

Chapter 10. JDBC and SQLJ connection pooling support

Connection pooling is part of JDBC 2.0 `DataSource` support, and is supported by the JDBC/SQLJ Driver for OS/390 and z/OS and the DB2 Universal JDBC Driver. For the DB2 Universal JDBC Driver in the z/OS environment, connection pooling is supported for Universal Driver type 2 connectivity and Universal Driver type 4 connectivity.

The JDBC/SQLJ Driver for OS/390 and z/OS and the DB2 Universal JDBC Driver provide 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.

Chapter 11. Universal Driver type 4 connectivity JDBC and SQLJ distributed transaction support

The DB2 Universal JDBC Driver in the z/OS environment supports distributed transaction management when you use Universal Driver type 4 connectivity. This support implements the Java 2 Platform, Enterprise Edition (J2EE) Java Transaction Service (JTS) and Java Transaction API (JTA) specifications, and conforms to the X/Open standard for global transactions (*Distributed Transaction Processing: The XA Specification*, available from http://www.opengroup.org). DB2 Universal JDBC Driver distributed transaction support lets Enterprise Java Beans (EJBs) and Java servlets that run under WebSphere Application Server Version 5.01 and above participate in a distributed transaction system.

A distributed transaction system consists of a resource manager, a transaction manager, and transactional applications. Table 54 lists the products and programs in the z/OS environment that provide those components.

Table 54. Components of a distributed transaction system on DB2 UDB for z/OS

Distributed transaction system component	Component function provided by
Resource manager	DB2 UDB for z/OS or DB2 UDB for Linux, UNIX and Windows
Transaction manager	WebSphere Application Server or another application server
Transactional applications	JDBC or SQLJ applications

Your client application programs that run under the DB2 Universal JDBC Driver can use distributed transaction support for connections to DB2 UDB for z/OS or DB2 UDB for Linux, UNIX and Windows servers. DB2 UDB for z/OS Version 8 servers include native XA mode support. However, DB2 UDB for OS/390 and z/OS Version 7 servers do not have native XA mode support, so the DB2 Universal JDBC Driver emulates the XA mode support using the existing DB2 DRDA two-phase commit protocol. This XA mode emulation uses a DB2 table named SYSIBM.INDOUBT to store information about indoubt transactions. DB2 uses a package named T4XAIndbtPkg to perform SQL operations on SYSIBM.INDOUBT.

If your JDBC or SQLJ applications use distributed transactions, and those applications connect to DB2 UDB for OS/390 and z/OS Version 7 servers, you need to run the DB2T4XAIndoubtUtil utility at those servers to create the SYSIBM.INDOUBT table and the T4XAIndbtPkg package. You should never modify the SYSIBM.INDOUBT table manually. See "DB2T4XAIndoubtUtil for distributed transactions with DB2 UDB for OS/390 and z/OS Version 7 servers" on page 267 for information on running the DB2T4XAIndoubtUtil utility.

In JDBC or SQLJ applications, distributed transactions are supported for connections that are established using the `DataSource` interface. A connection is normally established by the application server.

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 68 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 68. 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 69 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 69. Example of a distributed transaction under an application server

Figure 70 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 70. 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 70. 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 70. 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 70. 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.

Chapter 12. JDBC and SQLJ global transaction support

The JDBC/SQLJ 2.0 Driver for OS/390 and z/OS and Universal Driver type 2 connectivity on DB2 UDB for z/OS include global transaction support. JDBC and SQLJ global transaction support lets Enterprise Java Beans (EJB) and Java servlets that run under WebSphere Application Server Version 4.0 or later access DB2 UDB for z/OS relational data within global transactions. WebSphere Application Server provides the environment to deploy EJBs and servlets, and RRS provides the transaction management.

You can use global transactions in JDBC or SQLJ applications. Global transactions are supported for connections that are established using the `DriverManager` or the `DataSource` interface.

The best way to demonstrate global transactions is to contrast them with local transactions. As Figure 71 shows, with local transactions, you call the `commit` or `rollback` methods of the `Connection` class to make the changes to the database permanent and indicate the end of each unit or work. Alternatively, you can use the `setAutoCommit(true)` method to perform a commit operation after every SQL statement.

```
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
.
.
.
```

Figure 71. Example of a local transaction

In contrast, applications cannot call the `commit`, `rollback`, or `setAutoCommit(true)` methods on the `Connection` object when the applications are in a global transaction. With global transactions, the `commit` or `rollback` methods on the `Connection` object do not indicate transaction boundaries. Instead, your applications let WebSphere manage transaction boundaries. Alternatively, you can use DB2-customized Java Transaction API (JTA) interfaces to indicate the boundaries of transactions. Although DB2 UDB for z/OS does not implement the JTA specification, the methods for delimiting transaction boundaries are available with the JDBC 2.0 driver. Figure 72 on page 308 demonstrates the use of the JTA interfaces to indicate global transaction boundaries.

```
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a global 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 global transaction.
utx.commit();
:
```

Figure 72. Example of a global transaction

When you run a multi-threaded client under WebSphere, a transaction can span multiple threads. This situation might occur in a Java servlet. An application that runs in this environment needs to perform some SQL on each `Connection` object before the application passes the object to another thread. Figure 73 illustrates this point.

```
javax.transaction.UserTransaction utx;
// Use the begin method on a UserTransaction object to indicate
// the beginning of a global transaction.
utx.begin();
:
// Obtain two JDBC Connections from DataSource ds
c1 = ds.getConnection();
c2 = ds.getConnection();
:
// Create a thread for each Connection object
ThreadClass1 tc1 = new ThreadClass1(c1);
ThreadClass2 tc2 = new ThreadClass1(c2);
Thread t1 = new Thread(tc1);
Thread t2 = new Thread(tc2);
// Execute some SQL on each Connection object to associate
// the threads with the global transaction
:
// Start the two threads that will use the Connection objects to do SQL
t1.start();
t2.start();
:
// Use the commit method on the UserTransaction object to
// drive all transaction branches to commit and indicate
// the end of the global transaction.
utx.commit();
:
```

Figure 73. Example of a global transaction in a multi-threaded environment

Chapter 13. Multiple z/OS context support in JDBC/SQLJ Driver for OS/390 and z/OS

The JDBC/SQLJ Driver for OS/390 and z/OS has multiple z/OS context support. The z/OS context includes the application's logical connection to the data source and the associated internal DB2 connection information that lets the application direct its operations to a data source. For JDBC or SQLJ applications, a context is equivalent to a DB2 thread.

The following topics provide additional information:

Connecting when multiple z/OS context support is not enabled

A context is always established when a Java thread creates its first `java.sql.Connection` object. If support for multiple contexts is not enabled, then subsequent `java.sql.Connection` objects created by a Java thread share that single context. Although multiple connections can share a single context, only one connection can have an active transaction at any time. If there is an active transaction on a connection, a COMMIT or ROLLBACK must be issued before the Java thread can use or create another connection object.

Without multiple context support:

- There can be one or more Java threads, any of which can issue JDBC or SQLJ calls.
- All `java.sql.Connection` objects must be explicitly closed by the application Java thread that created the connection object.
- Multiple `java.sql.Connection` objects can be created by a single Java thread if the application uses the connections serially. The application must not create or use a different connection object on the Java thread if the current connection is not on a transaction boundary. Multiple connections cannot create concurrent units of work.
- When more than one connection is opened, those connections are associated with the same DB2 thread. Returning from the current connection to a previous connection might not return you to the DB2 location that the previous connection was originally associated with. Previous connections become associated with the location of the most recently created connection.
- A Java thread can use a `java.sql.Connection` object only when the Java thread creates the `java.sql.Connection` object.
- WebSphere™ Application Server connection pooling using the ″com.ibm.servlet.connmgr″ package is not possible.

Connecting when multiple z/OS context support is enabled

With multiple z/OS context support enabled, each `java.sql.Connection` object is related to a unique context (DB2 thread). Under this model, a single Java thread (TCB) can have multiple, concurrent connections, each with its own independent transaction. The DB2 JDBC and SQLJ multiple context support requires:

- Use of the DB2 RRSAF attachment facility
- z/OS Context Services

With multiple z/OS context support:

- There can be one or more Java threads, any of which can issue JDBC or SQLJ calls.
- The Java threads can create multiple `java.sql.Connection` objects (and derived objects), each of which:
  - Can exist concurrently with other `java.sql.Connection` objects.
  - Has its own transaction scope that is independent from all other `java.sql.Connection` objects.
  - Does not need to be on a transaction boundary for a Java thread to create or use different connections.
- The `java.sql.Connection` objects can be shared between Java threads. However, the actions of one Java thread on a given connection object are also visible to all of the Java threads using that connection. Also, the JDBC/SQLJ application is responsible for ensuring proper serialization when sharing connection objects between threads.
- Although it is recommended that all `java.sql.Statement` and `java.sql.Connection` objects be explicitly closed by the application, it is not required.
- WebSphere Application Server connection pooling using the `com.ibm.servlet.connmgr` package is supported for JDBC connections only.

Enabling multiple z/OS context support

The DB2SQLJMULTICONTEXT parameter in the run-time properties file enables multiple context support. See "The SQLJ/JDBC run-time properties file" on page 281 for information about setting the DB2SQLJMULTICONTEXT parameter.

Multiple context performance

Setting the DB2SQLJMULTICONTEXT parameter to YES enhances SQLJ and JDBC performance.

Connection sharing

Connection sharing occurs whenever a Java thread (TCB) attempts to use a `java.sql.Connection` object, or any object derived from a connection, that the Java thread did not create.

One application of connection sharing is for cleanup of connection objects. Under the Java Virtual Machine (JVM) on z/OS, cleanup of connection objects is usually performed by a JVM finalizer thread, rather than the Java thread that created the object.

Connection sharing is supported only in a multiple context environment.

Chapter 14. DB2 Universal JDBC Driver support for connection concentrator and Sysplex workload balancing

The following topics contain information about DB2 Universal JDBC Driver support for the connection concentrator and Sysplex workload balancing functions of DB2.

- "JDBC connection concentrator and Sysplex workload balancing"
- "Example of enabling the DB2 Universal JDBC Driver connection concentrator and Sysplex workload balancing" on page 312
- "Techniques for monitoring DB2 Universal JDBC Driver connection concentrator and Sysplex workload balancing" on page 313

JDBC connection concentrator and Sysplex workload balancing

Java applications that use DB2 Universal JDBC Driver type 4 connectivity to access DB2 UDB for z/OS servers can take advantage of the connection concentrator and Sysplex workload balancing functions.

The DB2 Universal JDBC Driver connection concentrator and Sysplex workload balancing functions are similar to the connection concentrator and Sysplex workload balancing functions of DB2 Connect.

The DB2 Universal JDBC Driver connection concentrator can reduce the resources that DB2 UDB for z/OS database servers require to support large numbers of client applications. The DB2 Universal JDBC Driver connection concentrator function lets many connection objects use the same physical connection, which reduces the total number of physical connections to the database server.

DB2 Universal JDBC Driver 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 UDB 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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver configuration properties.

At the connection level, you enable and disable the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver configuration properties
- Using an application programming interface

Example of enabling the DB2 Universal JDBC Driver connection concentrator and Sysplex workload balancing

The following procedure is an example of enabling the DB2 Universal JDBC Driver connection concentrator and Sysplex workload balancing functions with WebSphere Application Server.

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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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:

- DB2 Universal JDBC Driver at the FixPak 10 level
- WebSphere Application Server, Version 5.1 or later

Procedure:

1. Verify that the DB2 Universal JDBC Driver is at the correct level to support the connection concentrator and Sysplex workload balancing by issuing the following command in UNIX System Services:

   ```
   java com.ibm.db2.jcc.DB2Jcc -version
   ```

   Find a line in the output like this:

   ```
   [ibm][db2][jcc] Driver: IBM DB2 JDBC Universal Driver Architecture nnn xxx
   ```

   nnn should be 2.7 or later.
2. Set DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver classpath.

3. Set DB2 Universal JDBC Driver 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 55. Example of data source property settings for DB2 Universal JDBC Driver 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.

Techniques for monitoring DB2 Universal JDBC Driver connection concentrator and Sysplex workload balancing

To monitor the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver has made to the pool since the pool was created.

nsr The number of successful requests that the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver.

totalRequestsToPool

Format:

```
public int totalRequestsToPool()
```

Retrieves the total number of requests that the DB2 Universal JDBC Driver has made to the pool since the pool was created.

successfullRequestsFromPool

Format:

```
public int successfullRequestsFromPool()
```

Retrieves the number of successful requests that the DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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 DB2 Universal JDBC Driver 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()
```

The number of objects that are currently in the pool.

Chapter 15. Diagnosing JDBC and SQLJ problems

The following topic provide information on diagnosing JDBC and SQLJ problems.
- "JDBC and SQLJ problem diagnosis with the DB2 Universal JDBC Driver"
- "Example of a trace program under the DB2 Universal JDBC Driver" on page 320
- "Formatting trace data for C/C++ native driver code with the DB2 Universal JDBC Driver" on page 324
- "Diagnosing SQLJ problems with the JDBC/SQLJ Driver for OS/390 and z/OS" on page 325

JDBC and SQLJ problem diagnosis with the DB2 Universal JDBC Driver

To obtain data for diagnosing SQLJ or JDBC problems with the DB2 Universal JDBC Driver, collect trace data and run utilities that format the trace data. You should run the trace and diagnostic utilities only under the direction of IBM® software support.

Collecting JDBC trace data:

Use one of the following procedures to start the trace:

Procedure 1: For Universal Driver type 2 connectivity, the recommended method is to start the trace by setting the db2.jcc.override.traceFile property and the db2.jcc.t2zosTraceFile property in the DB2 Universal JDBC Driver configuration properties file. See "DB2 Universal JDBC Driver configuration properties customization" on page 253 for information on how to do this.

Procedure 2: For Universal Driver type 4 connectivity, 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 DB2 Universal JDBC Driver configuration properties file. See "DB2 Universal JDBC Driver configuration properties customization" on page 253 for information on how to do this.

Procedure 3:

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 DB2 Universal JDBC Driver" on page 185 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 4:

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 DB2 Universal JDBC Driver" on page 185 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 5: 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" on page 319.

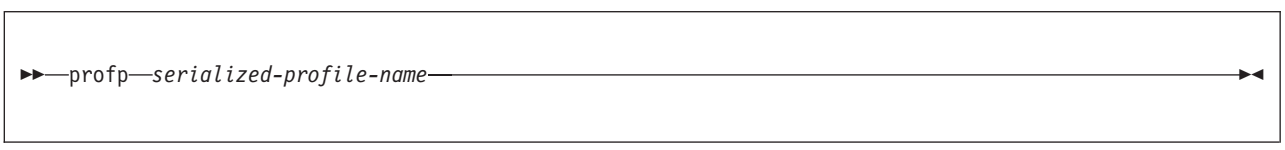
Trace example program: For a complete example of a program for tracing under the DB2 Universal JDBC Driver, see "Example of a trace program under the DB2 Universal JDBC Driver" on page 320.

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:

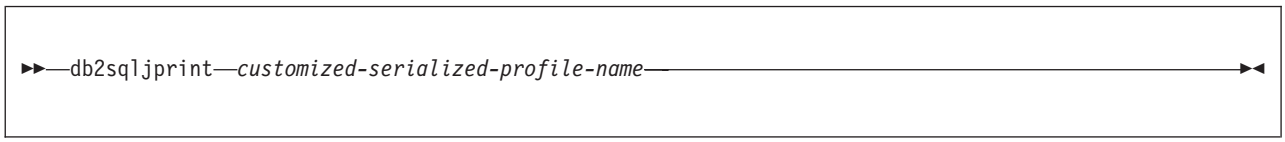
The profp utility formats information about each SQLJ clause in a serialized profile. The format of the profp utility is:



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 DB2 Universal JDBC Driver. The format of the db2sqljprint utility is:



Run the db2sqljprint utility on the customized serialized profile for the connection in which the error occurs.

Example of using configuration properties to start a JDBC trace

Suppose that you want to collect trace data for a program named Test.java, which uses Universal Driver 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

Example of a trace program under the DB2 Universal JDBC Driver

The following example shows a class for establishing a connection and gathering and displaying trace data under the DB2 Universal JDBC Driver. 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 74. Example of tracing under the DB2 Universal JDBC Driver (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 74. Example of tracing under the DB2 Universal JDBC Driver (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 74. Example of tracing under the DB2 Universal JDBC Driver (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] Universal Driver 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 74. Example of tracing under the DB2 Universal JDBC Driver (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 74. Example of tracing under the DB2 Universal JDBC Driver (Part 5 of 5)

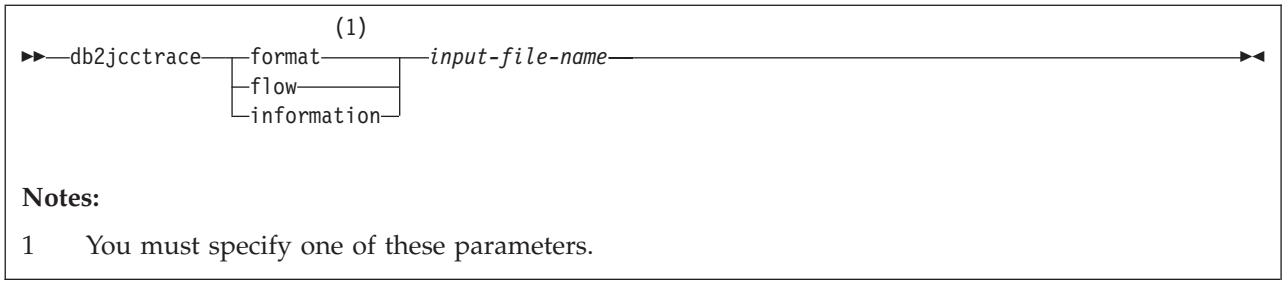
Formatting trace data for C/C++ native driver code with the DB2 Universal JDBC Driver

To format trace data for C/C++ native driver code under DB2 Universal JDBC Driver type 2 connectivity on DB2 UDB for z/OS, you execute the `db2jcctrace` command from the z/OS UNIX System Services command line.

You enable tracing of C/C++ native driver code by setting a value for the db2.jcc.t2zosTraceFile property. That value is the name of the file to which the DB2 Universal JDBC Driver writes the trace data.

The value of db2.jcc.t2zosTraceFile is the name of the input file for `db2jcctrace`. `db2jcctrace` writes formatted trace data to stdout. You can pipe the output to any file.

The format of `db2jccjtrace` is:



Notes:

1 You must specify one of these parameters.

The meanings of the parameters are:

format
Specifies that the output trace file contains formatted trace data.

Abbreviation: `fmt`

flow
Specifies that the output trace file contains control flow information.

Abbreviation: `flw`

information
Specifies that the output trace file contains information about the trace, such as the version of the driver, the time at which the trace was taken, and whether the trace file wrapped or was truncated. This information is also included in the output trace file when you specify format or flow.

Abbreviation: `inf` or `info`

input-file-name
Specifies the name of the file from which `db2jcctrace` is to read the unformatted trace data.

Diagnosing SQLJ problems with the JDBC/SQLJ Driver for OS/390 and z/OS

SQLJ programs can generate two types of errors:

- Recoverable errors

  SQLJ reports recoverable SQL errors through the JDBC `java.sql.SQLException` class. You can use methods `getErrorCode` and `getSQLState` to retrieve error codes and SQLSTATEs. See "Handling an SQLException under the JDBC/SQLJ Driver for OS/390 and z/OS" on page 55 for information on how to write your application program to retrieve error codes and SQLSTATEs.

  All SQLSTATEs except FFFFF are documented in Part 2 of *DB2 Messages*. FFFFF is a special SQLSTATE that indicates an internal error in the JDBC/SQLJ Driver for OS/390 and z/OS. error code values that are associated with SQLSTATE FFFFF are also not documented. If you receive SQLSTATE FFFFF, contact your IBM service representative.

- Non-recoverable errors

  These errors do not throw an SQLException, or the application cannot catch the exception.

To diagnose recoverable errors that generate SQLSTATE FFFFF or repeatable, non-recoverable errors, you can collect trace data and run utilities that generate additional diagnostic information. You should run the trace and diagnostic utilities only under the direction of your IBM service representative.

Formatting trace data with the JDBC/SQLJ Driver for OS/390 and z/OS

Before you can format SQLJ trace data, you must set several environment variables. You must also set several parameters in the run-time properties file that you name in environment variable DB2SQLJPROPERTIES. "The SQLJ/JDBC run-time properties file" on page 281 describes these variables and parameters.

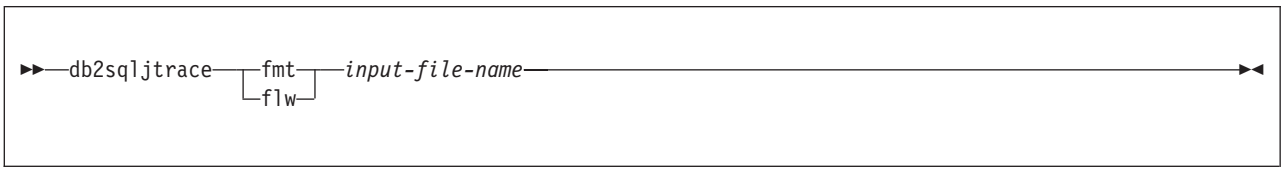
In the CICS environment, configuring for traces is somewhat different than in other environments. See "Special considerations for CICS applications," on page 329 for information on tracing in the CICS environment.

When you set the parameter DB2SQLJ_TRACE_FILENAME in the run-time properties file, you enable SQLJ/JDBC tracing. The JDBC/SQLJ Driver for OS/390 and z/OS generates two trace files:

- One trace file has a proprietary, binary format and must be formatted using the `db2sqljtrace` command. The name of that trace file is *trace-file*, where *trace-file* is the value to which you set DB2SQLJ_TRACE_FILENAME.
- The other trace file contains readable text, which requires no additional formatting. The name of that trace file is *trace-file*.JTRACE.

If your IBM service representative requests a DB2 SQLJ/JDBC trace, you need to format *trace-file* using `db2sqljtrace`. Send the `db2sqljtrace` output and *trace-file*.JTRACE to IBM.

The `db2sqljtrace` command writes the formatted trace data to stdout. The format of `db2sqljtrace` is:



The meanings of the parameters are:

fmt
Specifies that the output trace file is to contain a record of each time a function is entered or exited before the failure occurs.

flw
Specifies that the output trace file is to contain the function flow before the failure occurs.

input-file-name
Specifies the name of the file from which db2sqljtrace is to read the unformatted trace data. This name is the name you specified for environment variable DB2SQLJ_TRACE_FILENAME.

Running utilities to format diagnostic data

This topic describes utilities that you can run to retrieve and format diagnostic data when an internal error occurs.

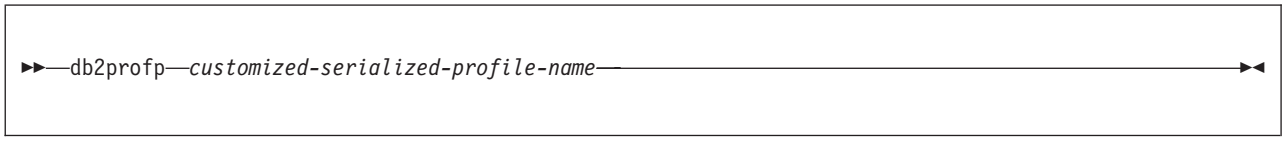
Using the profp utility to format information about a serialized profile

The `profp` utility formats information about each SQLJ clause in a serialized profile. The format of the `profp` utility is:

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.

Using the db2profp utility to format information about a JDBC/SQLJ Driver for OS/390 and z/OS customized profile

The `db2profp` utility formats information about each SQLJ clause in a serialized profile that is customized for the JDBC/SQLJ Driver for OS/390 and z/OS. The format of the `db2profp` utility is:



Run the `db2profp` utility on the customized serialized profile for the connection in which the error occurs.

Appendix. Special considerations for CICS applications

In general, writing and running JDBC and SQLJ applications for a CICS environment is similar to writing and running any other JDBC and SQLJ applications. However, there are some important differences.

The following topics explain the differences in running in the CICS environment and in other environments:

The *CICS Transaction Server for z/OS DB2 Guide* is the primary source for information on setting up the CICS environment for JDBC and SQLJ. Refer to that document before you read this material.

Choosing parameter values for the SQLJ/JDBC run-time properties file

Some parameters in the SQLJ/JDBC run-time properties file have different meanings in the CICS environment from other environments. Those parameters are:

DB2SQLJPLANNAME
This parameter is not used in a CICS environment. Specify the name of the plan that is associated with the SQLJ or JDBC application in one of the following places:
- The PLAN parameter of the DB2CONN definition
- The PLAN parameter of the DB2ENTRY definition
- The CPRMPLAN parameter of a dynamic plan exit

DB2SQLJ_TRACE_FILENAME
For the JVM environment, you can specify a fully-qualified path name or an unqualified file name. If you specify an unqualified file name, the file is allocated in the directory path that is specified by the CICS JVM environment variable CICS_HOME.

If you want to use the same properties file for both environments, specify a fully-qualified path name.

DB2SQLJSSID
This parameter is not used in a CICS environment.

DB2SQLJMULTICONTEXT
This parameter is not used in a CICS environment. You cannot enable z/OS multiple context support in the CICS environment. Each CICS Java application can have a maximum of one connection.

Choosing parameter values for the db2genJDBC utility

The db2genJDBC creates a JDBC profile. The default value for the `statements` parameters might not be appropriate for CICS applications. The default value generates a large JDBC profile.

Choose a value for the `statements` parameter that is lower than the default of 150. The default value produces more sections than are necessary for typical CICS applications. A larger number of sections results in a larger JDBC profile size. A value of 10 should be adequate for most CICS applications.

Choosing the number of cursors for JDBC result sets

The cursor properties file describes the DB2 cursors that the JDBC/SQLJ Driver for OS/390 and z/OS uses to process JDBC result sets. The default cursor properties file, db2jdbc.cursors, defines 100 cursors with the WITH HOLD attribute, and 100 cursors without the WITH HOLD attribute. This number of cursors is too large for CICS applications, and it results in a JDBC profile size that is large enough to degrade performance.

Specifying five cursors with hold and five cursors without hold should be should be adequate for most CICS applications.

Setting environment variables for the CICS environment

For CICS JDBC and SQLJ programs that run in the JVM environment, the way in which you specify environment variables depends on the release of CICS:

- For CICS Transaction Server V1R3, you specify the environment variables that are listed in "Setting environment variables for the JDBC/SQLJ Driver for OS/390 and z/OS" on page 280 in the DFHJVM member of the SDFHENV data set. The DB2SQLJPROPERTIES environment variable specifies the name of the run-time properties file.
- For CICS Transaction Server V2R2 or later, which uses the IBM Developer Kit for OS/390, Java 2 Technology Edition, SDK 1.3.1 or later, the DB2SQLJPROPERTIES environment variable is not used. You need to set all system properties that are required by the JDBC/SQLJ Driver for OS/390 and z/OS in the system properties file that is referenced by the JVMPROPS parameter in the relevant JVM profile. For more information, see *CICS Transaction Server for z/OS DB2 Guide*.

Connecting to DB2 in the CICS environment

For SQLJ or JDBC applications in a CICS environment, the connection to DB2 is always through the CICS attachment facility. Unlike SQLJ and JDBC applications that use other attachment facilities, SQLJ and JDBC applications that use the CICS attachment facility can create only one JDBC `java.sql.Connection` object within a unit of work. That `java.sql.Connection` object is associated with the CICS unit of work. CICS coordinates all DB2 updates within the unit of work.

A program that runs in the CICS environment cannot specify a user ID or password in a `getConnection` method call. Doing so causes an `SQLException`.

In CICS DB2 programs that are written in languages other than Java, calling applications and called applications can share a DB2 thread. JDBC does not allow several applications to share a `java.sql.Connection` object, which, in the CICS

environment, means that calling applications and called applications cannot share a DB2 thread. Therefore, if a CICS application is doing DB2 work, and that application calls an SQLJ or JDBC application, the calling application needs to commit all updates before calling the SQLJ or JDBC application.

The CICS attachment facility supports multithreading. Multiple Java threads are supported for a single CICS application. However, only the Java thread for the main application is associated with the DB2 attachment. JDBC and SQLJ processing is not supported for Java child threads.

In a CICS SQLJ or JDBC application, you need to explicitly close the `java.sql.Connection` before the program ends. This ensures that work done on the `Connection` object is committed and that the `java.sql.Connection` object is available for use by another application.

In the CICS environment, when an application creates a `Connection` object using the default URL (`"jdbc:default:connection"` or `"jdbc:db2os390sqlj:"`), CICS continues an existing connection for a DB2 thread. The new `Connection` object has the previous server location and transaction state. When you close this `Connection` object, CICS does not do an automatic commit, and the application does not throw an `SQLException` if the DB2 thread is not on a transaction boundary.

Commit and rollback processing in CICS SQLJ and JDBC applications

In a CICS environment, the default state of autoCommit for a JDBC connection is off. You can use JDBC and SQLJ commit and rollback processing in your CICS applications. The JDBC/SQLJ Driver for OS/390 and z/OS translates commit and rollback statements to CICS syncpoint calls. The scope of those calls is the entire CICS transaction.

Abnormal terminations in the CICS attachment facility

Abends in code that is called by the JDBC/SQLJ Driver for OS/390 and z/OS, such as abends in the CICS attachment facility, do not generate exceptions in SQLJ or JDBC programs.

A CICS attachment facility abend causes a rollback to the last syncpoint.

Running traces in a CICS environment

When you trace a JDBC or SQLJ CICS application that runs in a JVM, the trace output goes to *trace-file* (the binary trace) and *trace-file*.JTRACE (the readable trace), as described in "Formatting trace data with the JDBC/SQLJ Driver for OS/390 and z/OS" on page 326.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country 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.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Programming interface information

This book is intended to help the customer write applications that use Java to access IBM DB2 UDB for z/OS servers. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by DB2 Universal Database for z/OS (DB2 UDB for z/OS).

General-use programming interfaces allow the customer to write programs that obtain the services of DB2 UDB for z/OS.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both.

CICS	MVS
Cloudscape	Notes
DB2	OS/390
DB2 Universal Database	RACF
DRDA	RETAIN
IBM	SecureWay
ibm.com	WebSphere
IMS	z/OS
Language Environment	zSeries

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

Glossary

The following terms and abbreviations are defined as they are used in the DB2 library.

A

abend. Abnormal end of task.

abend reason code. A 4-byte hexadecimal code that uniquely identifies a problem with DB2.

abnormal end of task (abend). Termination of a task, job, or subsystem because of an error condition that recovery facilities cannot resolve during execution.

access method services. The facility that is used to define and reproduce VSAM key-sequenced data sets.

access path. The path that is used to locate data that is specified in SQL statements. An access path can be indexed or sequential.

active log. The portion of the DB2 log to which log records are written as they are generated. The active log always contains the most recent log records, whereas the archive log holds those records that are older and no longer fit on the active log.

active member state. A state of a member of a data sharing group. The cross-system coupling facility identifies each active member with a group and associates the member with a particular task, address space, and z/OS system. A member that is not active has either a failed member state or a quiesced member state.

address space. A range of virtual storage pages that is identified by a number (ASID) and a collection of segment and page tables that map the virtual pages to real pages of the computer's memory.

address space connection. The result of connecting an allied address space to DB2. Each address space that contains a task that is connected to DB2 has exactly one address space connection, even though more than one task control block (TCB) can be present. See also *allied address space* and *task control block*.

| **address space identifier (ASID).** A unique
| system-assigned identifier for and address space.

administrative authority. A set of related privileges that DB2 defines. When you grant one of the administrative authorities to a person's ID, the person has all of the privileges that are associated with that administrative authority.

after trigger. A trigger that is defined with the trigger activation time AFTER.

agent. As used in DB2, the structure that associates all processes that are involved in a DB2 unit of work. An *allied agent* is generally synonymous with an *allied thread*. *System agents* are units of work that process tasks that are independent of the allied agent, such as prefetch processing, deferred writes, and service tasks.

aggregate function. An operation that derives its
result by using values from one or more rows. Contrast
with *scalar function*.

alias. An alternative name that can be used in SQL statements to refer to a table or view in the same or a remote DB2 subsystem.

allied address space. An area of storage that is external to DB2 and that is connected to DB2. An allied address space is capable of requesting DB2 services.

allied thread. A thread that originates at the local DB2 subsystem and that can access data at a remote DB2 subsystem.

allocated cursor. A cursor that is defined for stored procedure result sets by using the SQL ALLOCATE CURSOR statement.

already verified. An LU 6.2 security option that allows DB2 to provide the user's verified authorization ID when allocating a conversation. With this option, the user is not validated by the partner DB2 subsystem.

| **ambiguous cursor.** A database cursor that is in a plan
| or package that contains either PREPARE or EXECUTE
| IMMEDIATE SQL statements, and for which the
| following statements are true: the cursor is not defined
| with the FOR READ ONLY clause or the FOR UPDATE
| OF clause; the cursor is not defined on a read-only
| result table; the cursor is not the target of a WHERE
| CURRENT clause on an SQL UPDATE or DELETE
| statement.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

APAR. Authorized program analysis report.

APAR fix corrective service. A temporary correction of an IBM software defect. The correction is temporary,

because it is usually replaced at a later date by a more permanent correction, such as a program temporary fix (PTF).

APF. Authorized program facility.

API. Application programming interface.

APPL. A VTAM® network definition statement that is used to define DB2 to VTAM as an application program that uses SNA LU 6.2 protocols.

application. A program or set of programs that performs a task; for example, a payroll application.

application-directed connection. A connection that an application manages using the SQL CONNECT statement.

application plan. The control structure that is produced during the bind process. DB2 uses the application plan to process SQL statements that it encounters during statement execution.

application process. The unit to which resources and locks are allocated. An application process involves the execution of one or more programs.

application programming interface (API). A functional interface that is supplied by the operating system or by a separately orderable licensed program that allows an application program that is written in a high-level language to use specific data or functions of the operating system or licensed program.

application requester. The component on a remote system that generates DRDA® requests for data on behalf of an application. An application requester accesses a DB2 database server using the DRDA application-directed protocol.

application server. The target of a request from a remote application. In the DB2 environment, the application server function is provided by the distributed data facility and is used to access DB2 data from remote applications.

archive log. The portion of the DB2 log that contains log records that have been copied from the active log.

ASCII. An encoding scheme that is used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC* and *Unicode*.

ASID. Address space identifier.

attachment facility. An interface between DB2 and TSO, IMS, CICS, or batch address spaces. An attachment facility allows application programs to access DB2.

attribute. A characteristic of an entity. For example, in database design, the phone number of an employee is one of that employee's attributes.

authorization ID. A string that can be verified for connection to DB2 and to which a set of privileges is allowed. It can represent an individual, an organizational group, or a function, but DB2 does not determine this representation.

authorized program analysis report (APAR). A report of a problem that is caused by a suspected defect in a current release of an IBM supplied program.

authorized program facility (APF). A facility that permits the identification of programs that are authorized to use restricted functions.

automatic query rewrite. A process that examines an SQL statement that refers to one or more base tables, and, if appropriate, rewrites the query so that it performs better. This process can also determine whether to rewrite a query so that it refers to one or more materialized query tables that are derived from the source tables.

auxiliary index. An index on an auxiliary table in which each index entry refers to a LOB.

auxiliary table. A table that stores columns outside the table in which they are defined. Contrast with *base table*.

B

backout. The process of undoing uncommitted changes that an application process made. This might be necessary in the event of a failure on the part of an application process, or as a result of a deadlock situation.

backward log recovery. The fourth and final phase of restart processing during which DB2 scans the log in a backward direction to apply UNDO log records for all aborted changes.

base table. (1) A table that is created by the SQL CREATE TABLE statement and that holds persistent data. Contrast with *result table* and *temporary table*.

(2) A table containing a LOB column definition. The actual LOB column data is not stored with the base table. The base table contains a row identifier for each row and an indicator column for each of its LOB columns. Contrast with *auxiliary table*.

base table space. A table space that contains base tables.

basic predicate. A predicate that compares two values.

basic sequential access method (BSAM). An access method for storing or retrieving data blocks in a continuous sequence, using either a sequential-access or a direct-access device.

batch message processing program. In IMS, an application program that can perform batch-type processing online and can access the IMS input and output message queues.

before trigger. A trigger that is defined with the trigger activation time BEFORE.

binary integer. A basic data type that can be further classified as small integer or large integer.

binary large object (BLOB). A sequence of bytes in which the size of the value ranges from 0 bytes to 2 GB–1. Such a string has a CCSID value of 65535.

binary string. A sequence of bytes that is not associated with a CCSID. For example, the BLOB data type is a binary string.

bind. The process by which the output from the SQL precompiler is converted to a usable control structure, often called an access plan, application plan, or package. During this process, access paths to the data are selected and some authorization checking is performed. The types of bind are:

automatic bind. (More correctly, *automatic rebind*) A process by which SQL statements are bound automatically (without a user issuing a BIND command) when an application process begins execution and the bound application plan or package it requires is not valid.

dynamic bind. A process by which SQL statements are bound as they are entered.

incremental bind. A process by which SQL statements are bound during the execution of an application process.

static bind. A process by which SQL statements are bound after they have been precompiled. All static SQL statements are prepared for execution at the same time.

bit data. Data that is character type CHAR or VARCHAR and has a CCSID value of 65535.

BLOB. Binary large object.

block fetch. A capability in which DB2 can retrieve, or fetch, a large set of rows together. Using block fetch can significantly reduce the number of messages that are being sent across the network. Block fetch applies only to cursors that do not update data.

BMP. Batch Message Processing (IMS). See *batch message processing program*.

bootstrap data set (BSDS). A VSAM data set that contains name and status information for DB2, as well as RBA range specifications, for all active and archive log data sets. It also contains passwords for the DB2 directory and catalog, and lists of conditional restart and checkpoint records.

BSAM. Basic sequential access method.

BSDS. Bootstrap data set.

buffer pool. Main storage that is reserved to satisfy the buffering requirements for one or more table spaces or indexes.

built-in data type. A data type that IBM supplies. Among the built-in data types for DB2 UDB for z/OS are string, numeric, ROWID, and datetime. Contrast with *distinct type*.

built-in function. A function that DB2 supplies. Contrast with *user-defined function*.

business dimension. A category of data, such as products or time periods, that an organization might want to analyze.

C

cache structure. A coupling facility structure that stores data that can be available to all members of a Sysplex. A DB2 data sharing group uses cache structures as group buffer pools.

CAF. Call attachment facility.

call attachment facility (CAF). A DB2 attachment facility for application programs that run in TSO or z/OS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment.

call-level interface (CLI). A callable application programming interface (API) for database access, which is an alternative to using embedded SQL. In contrast to embedded SQL, DB2 ODBC (which is based on the CLI architecture) does not require the user to precompile or bind applications, but instead provides a standard set of functions to process SQL statements and related services at run time.

cascade delete. The way in which DB2 enforces referential constraints when it deletes all descendent rows of a deleted parent row.

CASE expression. An expression that is selected based on the evaluation of one or more conditions.

cast function. A function that is used to convert instances of a (source) data type into instances of a different (target) data type. In general, a cast function has the name of the target data type. It has one single argument whose type is the source data type; its return type is the target data type.

castout. The DB2 process of writing changed pages from a group buffer pool to disk.

castout owner. The DB2 member that is responsible for casting out a particular page set or partition.

catalog. In DB2, a collection of tables that contains descriptions of objects such as tables, views, and indexes.

catalog table. Any table in the DB2 catalog.

CCSID. Coded character set identifier.

CDB. Communications database.

CDRA. Character Data Representation Architecture.

CEC. Central electronic complex. See *central processor complex*.

central electronic complex (CEC). See *central processor complex*.

central processor (CP). The part of the computer that contains the sequencing and processing facilities for instruction execution, initial program load, and other machine operations.

central processor complex (CPC). A physical collection of hardware (such as an ES/3090™) that consists of main storage, one or more central processors, timers, and channels.

CFRM. Coupling facility resource management.

CFRM policy. A declaration by a z/OS administrator regarding the allocation rules for a coupling facility structure.

character conversion. The process of changing characters from one encoding scheme to another.

Character Data Representation Architecture (CDRA). An architecture that is used to achieve consistent representation, processing, and interchange of string data.

character large object (CLOB). A sequence of bytes representing single-byte characters or a mixture of single- and double-byte characters where the size of the value can be up to 2 GB–1. In general, character large object values are used whenever a character string might exceed the limits of the VARCHAR type.

character set. A defined set of characters.

character string. A sequence of bytes that represent bit data, single-byte characters, or a mixture of single-byte and multibyte characters.

check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain.

check integrity. The condition that exists when each row in a table conforms to the check constraints that are defined on that table. Maintaining check integrity requires DB2 to enforce check constraints on operations that add or change data.

check pending. A state of a table space or partition that prevents its use by some utilities and by some SQL statements because of rows that violate referential constraints, check constraints, or both.

checkpoint. A point at which DB2 records internal status information on the DB2 log; the recovery process uses this information if DB2 abnormally terminates.

child lock. For explicit hierarchical locking, a lock that is held on either a table, page, row, or a large object (LOB). Each child lock has a parent lock. See also *parent lock*.

CI. Control interval.

CICS. Represents (in this publication): CICS Transaction Server for z/OS: Customer Information Control System Transaction Server for z/OS.

CICS attachment facility. A DB2 subcomponent that uses the z/OS subsystem interface (SSI) and cross-storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

CIDF. Control interval definition field.

claim. A notification to DB2 that an object is being accessed. Claims prevent drains from occurring until the claim is released, which usually occurs at a commit point. Contrast with *drain*.

claim class. A specific type of object access that can be one of the following isolation levels:
Cursor stability (CS)
Repeatable read (RR)
Write

claim count. A count of the number of agents that are accessing an object.

class of service. A VTAM term for a list of routes through a network, arranged in an order of preference for their use.

class word. A single word that indicates the nature of a data attribute. For example, the class word PROJ indicates that the attribute identifies a project.

clause. In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

CLI. Call- level interface.

client. See *requester*.

CLIST. Command list. A language for performing TSO tasks.

CLOB. Character large object.

closed application. An application that requires exclusive use of certain statements on certain DB2

objects, so that the objects are managed solely through the application's external interface.

CLPA. Create link pack area.

clustering index. An index that determines how rows are physically ordered (*clustered*) in a table space. If a clustering index on a partitioned table is not a partitioning index, the rows are ordered in cluster sequence within each data partition instead of spanning partitions. Prior to Version 8 of DB2 UDB for z/OS, the partitioning index was required to be the clustering index.

coded character set. A set of unambiguous rules that establish a character set and the one-to-one relationships between the characters of the set and their coded representations.

coded character set identifier (CCSID). A 16-bit number that uniquely identifies a coded representation of graphic characters. It designates an encoding scheme identifier and one or more pairs consisting of a character set identifier and an associated code page identifier.

code page. (1) A set of assignments of characters to code points. In EBCDIC, for example, the character 'A' is assigned code point X'C1' (2) , and character 'B' is assigned code point X'C2'. Within a code page, each code point has only one specific meaning.

code point. In CDRA, a unique bit pattern that represents a character in a code page.

code unit. The fundamental binary width in a computer architecture that is used for representing character data, such as 7 bits, 8 bits, 16 bits, or 32 bits. Depending on the character encoding form that is used, each code point in a coded character set can be represented internally by one or more code units.

coexistence. During migration, the period of time in which two releases exist in the same data sharing group.

cold start. A process by which DB2 restarts without processing any log records. Contrast with *warm start*.

collection. A group of packages that have the same qualifier.

column. The vertical component of a table. A column has a name and a particular data type (for example, character, decimal, or integer).

column function. See *aggregate function*.

"come from" checking. An LU 6.2 security option that defines a list of authorization IDs that are allowed to connect to DB2 from a partner LU.

command. A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.

command prefix. A one- to eight-character command identifier. The command prefix distinguishes the command as belonging to an application or subsystem rather than to MVS.

command recognition character (CRC). A character that permits a z/OS console operator or an IMS subsystem user to route DB2 commands to specific DB2 subsystems.

command scope. The scope of command operation in a data sharing group. If a command has *member scope*, the command displays information only from the one member or affects only non-shared resources that are owned locally by that member. If a command has *group scope*, the command displays information from all members, affects non-shared resources that are owned locally by all members, displays information on sharable resources, or affects sharable resources.

commit. The operation that ends a unit of work by releasing locks so that the database changes that are made by that unit of work can be perceived by other processes.

commit point. A point in time when data is considered consistent.

committed phase. The second phase of the multisite update process that requests all participants to commit the effects of the logical unit of work.

common service area (CSA). In z/OS, a part of the common area that contains data areas that are addressable by all address spaces.

communications database (CDB). A set of tables in the DB2 catalog that are used to establish conversations with remote database management systems.

comparison operator. A token (such as =, >, or <) that is used to specify a relationship between two values.

composite key. An ordered set of key columns of the same table.

compression dictionary. The dictionary that controls the process of compression and decompression. This dictionary is created from the data in the table space or table space partition.

concurrency. The shared use of resources by more than one application process at the same time.

conditional restart. A DB2 restart that is directed by a user-defined conditional restart control record (CRCR).

connection. In SNA, the existence of a communication path between two partner LUs that allows information

to be exchanged (for example, two DB2 subsystems that are connected and communicating by way of a conversation).

connection context. In SQLJ, a Java object that represents a connection to a data source.

connection declaration clause. In SQLJ, a statement that declares a connection to a data source.

connection handle. The data object containing information that is associated with a connection that DB2 ODBC manages. This includes general status information, transaction status, and diagnostic information.

connection ID. An identifier that is supplied by the attachment facility and that is associated with a specific address space connection.

consistency token. A timestamp that is used to generate the version identifier for an application. See also *version*.

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. Contrast with *variable*.

constraint. A rule that limits the values that can be inserted, deleted, or updated in a table. See *referential constraint*, *check constraint*, and *unique constraint*.

context. The application's logical connection to the data source and associated internal DB2 ODBC connection information that allows the application to direct its operations to a data source. A DB2 ODBC context represents a DB2 thread.

contracting conversion. A process that occurs when the length of a converted string is smaller than that of the source string. For example, this process occurs when an EBCDIC mixed-data string that contains DBCS characters is converted to ASCII mixed data; the converted string is shorter because of the removal of the shift codes.

control interval (CI). A fixed-length area or disk in which VSAM stores records and creates distributed free space. Also, in a key-sequenced data set or file, the set of records that an entry in the sequence-set index record points to. The control interval is the unit of information that VSAM transmits to or from disk. A control interval always includes an integral number of physical records.

control interval definition field (CIDF). In VSAM, a field that is located in the 4 bytes at the end of each control interval; it describes the free space, if any, in the control interval.

conversation. Communication, which is based on LU 6.2 or Advanced Program-to-Program Communication (APPC), between an application and a remote transaction program over an SNA logical unit-to-logical unit (LU-LU) session that allows communication while processing a transaction.

coordinator. The system component that coordinates the commit or rollback of a unit of work that includes work that is done on one or more other systems.

copy pool. A named set of SMS storage groups that contains data that is to be copied collectively. A copy pool is an SMS construct that lets you define which storage groups are to be copied by using FlashCopy® functions. HSM determines which volumes belong to a copy pool.

copy target. A named set of SMS storage groups that are to be used as containers for copy pool volume copies. A copy target is an SMS construct that lets you define which storage groups are to be used as containers for volumes that are copied by using FlashCopy functions.

copy version. A point-in-time FlashCopy copy that is managed by HSM. Each copy pool has a version parameter that specifies how many copy versions are maintained on disk.

correlated columns. A relationship between the value of one column and the value of another column.

correlated subquery. A subquery (part of a WHERE or HAVING clause) that is applied to a row or group of rows of a table or view that is named in an outer subselect statement.

correlation ID. An identifier that is associated with a specific thread. In TSO, it is either an authorization ID or the job name.

correlation name. An identifier that designates a table, a view, or individual rows of a table or view within a single SQL statement. It can be defined in any FROM clause or in the first clause of an UPDATE or DELETE statement.

cost category. A category into which DB2 places cost estimates for SQL statements at the time the statement is bound. A cost estimate can be placed in either of the following cost categories:

- A: Indicates that DB2 had enough information to make a cost estimate without using default values.
- B: Indicates that some condition exists for which DB2 was forced to use default values for its estimate.

The cost category is externalized in the COST_CATEGORY column of the DSN_STATEMNT_TABLE when a statement is explained.

coupling facility. A special PR/SM™ LPAR logical partition that runs the coupling facility control program and provides high-speed caching, list processing, and locking functions in a Parallel Sysplex®.

coupling facility resource management. A component of z/OS that provides the services to manage coupling facility resources in a Parallel Sysplex. This management includes the enforcement of CFRM policies to ensure that the coupling facility and structure requirements are satisfied.

CP. Central processor.

CPC. Central processor complex.

C++ member. A data object or function in a structure, union, or class.

C++ member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and to the member functions of objects in its class. Member functions are also called methods.

C++ object. (1) A region of storage. An object is created when a variable is defined or a new function is invoked. (2) An instance of a class.

CRC. Command recognition character.

CRCR. Conditional restart control record. See also *conditional restart*.

create link pack area (CLPA). An option that is used during IPL to initialize the link pack pageable area.

created temporary table. A table that holds temporary data and is defined with the SQL statement CREATE GLOBAL TEMPORARY TABLE. Information about created temporary tables is stored in the DB2 catalog, so this kind of table is persistent and can be shared across application processes. Contrast with *declared temporary table*. See also *temporary table*.

cross-memory linkage. A method for invoking a program in a different address space. The invocation is synchronous with respect to the caller.

cross-system coupling facility (XCF). A component of z/OS that provides functions to support cooperation between authorized programs that run within a Sysplex.

cross-system extended services (XES). A set of z/OS services that allow multiple instances of an application or subsystem, running on different systems in a Sysplex environment, to implement high-performance, high-availability data sharing by using a coupling facility.

CS. Cursor stability.

CSA. Common service area.

CT. Cursor table.

current data. Data within a host structure that is current with (identical to) the data within the base table.

current SQL ID. An ID that, at a single point in time, holds the privileges that are exercised when certain dynamic SQL statements run. The current SQL ID can be a primary authorization ID or a secondary authorization ID.

current status rebuild. The second phase of restart processing during which the status of the subsystem is reconstructed from information on the log.

cursor. A named control structure that an application program uses to point to a single row or multiple rows within some ordered set of rows of a result table. A cursor can be used to retrieve, update, or delete rows from a result table.

cursor sensitivity. The degree to which database updates are visible to the subsequent FETCH statements in a cursor. A cursor can be sensitive to changes that are made with positioned update and delete statements specifying the name of that cursor. A cursor can also be sensitive to changes that are made with searched update or delete statements, or with cursors other than this cursor. These changes can be made by this application process or by another application process.

cursor stability (CS). The isolation level that provides maximum concurrency without the ability to read uncommitted data. With cursor stability, a unit of work holds locks only on its uncommitted changes and on the current row of each of its cursors.

cursor table (CT). The copy of the skeleton cursor table that is used by an executing application process.

cycle. A set of tables that can be ordered so that each table is a descendent of the one before it, and the first table is a descendent of the last table. A self-referencing table is a cycle with a single member.

D

DAD. See *Document access definition.*

disk. A direct-access storage device that records data magnetically.

database. A collection of tables, or a collection of table spaces and index spaces.

database access thread. A thread that accesses data at the local subsystem on behalf of a remote subsystem.

database administrator (DBA). An individual who is responsible for designing, developing, operating, safeguarding, maintaining, and using a database.

database alias. The name of the target server if different from the location name. The database alias name is used to provide the name of the database server as it is known to the network. When a database alias name is defined, the location name is used by the application to reference the server, but the database alias name is used to identify the database server to be accessed. Any fully qualified object names within any SQL statements are not modified and are sent unchanged to the database server.

database descriptor (DBD). An internal representation of a DB2 database definition, which reflects the data definition that is in the DB2 catalog. The objects that are defined in a database descriptor are table spaces, tables, indexes, index spaces, relationships, check constraints, and triggers. A DBD also contains information about accessing tables in the database.

database exception status. An indication that something is wrong with a database. All members of a data sharing group must know and share the exception status of databases.

database identifier (DBID). An internal identifier of the database.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and the access to the data that is stored within it.

database request module (DBRM). A data set member that is created by the DB2 precompiler and that contains information about SQL statements. DBRMs are used in the bind process.

database server. The target of a request from a local application or an intermediate database server. In the DB2 environment, the database server function is provided by the distributed data facility to access DB2 data from local applications, or from a remote database server that acts as an intermediate database server.

data currency. The state in which data that is retrieved into a host variable in your program is a copy of data in the base table.

data definition name (ddname). The name of a data definition (DD) statement that corresponds to a data control block containing the same name.

data dictionary. A repository of information about an organization's application programs, databases, logical data models, users, and authorizations. A data dictionary can be manual or automated.

data-driven business rules. Constraints on particular data values that exist as a result of requirements of the business.

Data Language/I (DL/I). The IMS data manipulation language; a common high-level interface between a user application and IMS.

data mart. A small data warehouse that applies to a single department or team. See also *data warehouse*.

data mining. The process of collecting critical business information from a data warehouse, correlating it, and uncovering associations, patterns, and trends.

data partition. A VSAM data set that is contained within a partitioned table space.

data-partitioned secondary index (DPSI). A secondary index that is partitioned. The index is partitioned according to the underlying data.

data sharing. The ability of two or more DB2 subsystems to directly access and change a single set of data.

data sharing group. A collection of one or more DB2 subsystems that directly access and change the same data while maintaining data integrity.

data sharing member. A DB2 subsystem that is assigned by XCF services to a data sharing group.

data source. A local or remote relational or non-relational data manager that is capable of supporting data access via an ODBC driver that supports the ODBC APIs. In the case of DB2 UDB for z/OS, the data sources are always relational database managers.

data space. In releases prior to DB2 UDB for z/OS, Version 8, a range of up to 2 GB of contiguous virtual storage addresses that a program can directly manipulate. Unlike an address space, a data space can hold only data; it does not contain common areas, system data, or programs.

data type. An attribute of columns, literals, host variables, special registers, and the results of functions and expressions.

data warehouse. A system that provides critical business information to an organization. The data warehouse system cleanses the data for accuracy and currency, and then presents the data to decision makers so that they can interpret and use it effectively and efficiently.

date. A three-part value that designates a day, month, and year.

date duration. A decimal integer that represents a number of years, months, and days.

datetime value. A value of the data type DATE, TIME, or TIMESTAMP.

DBA. Database administrator.

DBCLOB. Double-byte character large object.

DBCS. Double-byte character set.

DBD. Database descriptor.

DBID. Database identifier.

DBMS. Database management system.

DBRM. Database request module.

DB2 catalog. Tables that are maintained by DB2 and contain descriptions of DB2 objects, such as tables, views, and indexes.

DB2 command. An instruction to the DB2 subsystem that a user enters to start or stop DB2, to display information on current users, to start or stop databases, to display information on the status of databases, and so on.

DB2 for VSE & VM. The IBM DB2 relational database management system for the VSE and VM operating systems.

DB2I. DB2 Interactive.

DB2 Interactive (DB2I). The DB2 facility that provides for the execution of SQL statements, DB2 (operator) commands, programmer commands, and utility invocation.

DB2I Kanji Feature. The tape that contains the panels and jobs that allow a site to display DB2I panels in Kanji.

DB2 PM. DB2 Performance Monitor.

DB2 thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services.

DCLGEN. Declarations generator.

DDF. Distributed data facility.

ddname. Data definition name.

deadlock. Unresolvable contention for the use of a resource, such as a table or an index.

declarations generator (DCLGEN). A subcomponent of DB2 that generates SQL table declarations and COBOL, C, or PL/I data structure declarations that conform to the table. The declarations are generated from DB2 system catalog information. DCLGEN is also a DSN subcommand.

declared temporary table. A table that holds temporary data and is defined with the SQL statement DECLARE GLOBAL TEMPORARY TABLE. Information about declared temporary tables is not stored in the DB2 catalog, so this kind of table is not persistent and can be used only by the application process that issued the DECLARE statement. Contrast with *created temporary table*. See also *temporary table*.

default value. A predetermined value, attribute, or option that is assumed when no other is explicitly specified.

deferred embedded SQL. SQL statements that are neither fully static nor fully dynamic. Like static statements, they are embedded within an application, but like dynamic statements, they are prepared during the execution of the application.

deferred write. The process of asynchronously writing changed data pages to disk.

degree of parallelism. The number of concurrently executed operations that are initiated to process a query.

delete-connected. A table that is a dependent of table P or a dependent of a table to which delete operations from table P cascade.

delete hole. The location on which a cursor is positioned when a row in a result table is refetched and the row no longer exists on the base table, because another cursor deleted the row between the time the cursor first included the row in the result table and the time the cursor tried to refetch it.

delete rule. The rule that tells DB2 what to do to a dependent row when a parent row is deleted. For each relationship, the rule might be CASCADE, RESTRICT, SET NULL, or NO ACTION.

delete trigger. A trigger that is defined with the triggering SQL operation DELETE.

delimited identifier. A sequence of characters that are enclosed within double quotation marks ("). The sequence must consist of a letter followed by zero or more characters, each of which is a letter, digit, or the underscore character (_).

delimiter token. A string constant, a delimited identifier, an operator symbol, or any of the special characters that are shown in DB2 syntax diagrams.

denormalization. A key step in the task of building a physical relational database design. Denormalization is the intentional duplication of columns in multiple tables, and the consequence is increased data redundancy. Denormalization is sometimes necessary to minimize performance problems. Contrast with *normalization*.

dependent. An object (row, table, or table space) that has at least one parent. The object is also said to be a dependent (row, table, or table space) of its parent. See also *parent row*, *parent table*, *parent table space*.

dependent row. A row that contains a foreign key that matches the value of a primary key in the parent row.

dependent table. A table that is a dependent in at least one referential constraint.

DES-based authenticator. An authenticator that is generated using the DES algorithm.

descendent. An object that is a dependent of an object or is the dependent of a descendent of an object.

descendent row. A row that is dependent on another row, or a row that is a descendent of a dependent row.

descendent table. A table that is a dependent of another table, or a table that is a descendent of a dependent table.

deterministic function. A user-defined function whose result is dependent on the values of the input arguments. That is, successive invocations with the same input values produce the same answer. Sometimes referred to as a *not-variant* function. Contrast this with an *nondeterministic function* (sometimes called a *variant function*), which might not always produce the same result for the same inputs.

DFP. Data Facility Product (in z/OS).

DFSMS. Data Facility Storage Management Subsystem (in z/OS). Also called *Storage Management Subsystem (SMS)*.

| **DFSMSdss™.** The data set services (dss) component of
| DFSMS (in z/OS).

| **DFSMShsm™.** The hierarchical storage manager (hsm)
| component of DFSMS (in z/OS).

dimension. A data category such as time, products, or markets. The elements of a dimension are referred to as members. Dimensions offer a very concise, intuitive way of organizing and selecting data for retrieval, exploration, and analysis. See also *dimension table*.

dimension table. The representation of a dimension in a star schema. Each row in a dimension table represents all of the attributes for a particular member of the dimension. See also *dimension*, *star schema*, and *star join*.

directory. The DB2 system database that contains internal objects such as database descriptors and skeleton cursor tables.

distinct predicate. In SQL, a predicate that ensures
that two row values are not equal, and that both row
values are not null.

distinct type. A user-defined data type that is internally represented as an existing type (its source type), but is considered to be a separate and incompatible type for semantic purposes.

distributed data. Data that resides on a DBMS other than the local system.

distributed data facility (DDF). A set of DB2 components through which DB2 communicates with another relational database management system.

Distributed Relational Database Architecture™ (DRDA). A connection protocol for distributed relational database processing that is used by IBM's relational database products. DRDA includes protocols for communication between an application and a remote relational database management system, and for communication between relational database management systems. See also *DRDA access*.

DL/I. Data Language/I.

DNS. Domain name server.

| **document access definition (DAD).** Used to define
| the indexing scheme for an XML column or the
| mapping scheme of an XML collection. It can be used
| to enable an XML Extender column of an XML
| collection, which is XML formatted.

domain. The set of valid values for an attribute.

domain name. The name by which TCP/IP applications refer to a TCP/IP host within a TCP/IP network.

domain name server (DNS). A special TCP/IP network server that manages a distributed directory that is used to map TCP/IP host names to IP addresses.

double-byte character large object (DBCLOB). A sequence of bytes representing double-byte characters where the size of the values can be up to 2 GB. In general, DBCLOB values are used whenever a double-byte character string might exceed the limits of the VARGRAPHIC type.

double-byte character set (DBCS). A set of characters, which are used by national languages such as Japanese and Chinese, that have more symbols than can be represented by a single byte. Each character is 2 bytes in length. Contrast with *single-byte character set* and *multibyte character set*.

double-precision floating point number. A 64-bit approximate representation of a real number.

downstream. The set of nodes in the syncpoint tree that is connected to the local DBMS as a participant in the execution of a two-phase commit.

| **DPSI.** Data-partitioned secondary index.

drain. The act of acquiring a locked resource by quiescing access to that object.

drain lock. A lock on a claim class that prevents a claim from occurring.

DRDA. Distributed Relational Database Architecture.

DRDA access. An open method of accessing distributed data that you can use to can connect to another database server to execute packages that were previously bound at the server location. You use the SQL CONNECT statement or an SQL statement with a three-part name to identify the server. Contrast with *private protocol access*.

DSN. (1) The default DB2 subsystem name. (2) The name of the TSO command processor of DB2. (3) The first three characters of DB2 module and macro names.

duration. A number that represents an interval of time. See also *date duration*, *labeled duration*, and *time duration*.

dynamic cursor. A named control structure that an application program uses to change the size of the result table and the order of its rows after the cursor is opened. Contrast with *static cursor.*

dynamic dump. A dump that is issued during the execution of a program, usually under the control of that program.

dynamic SQL. SQL statements that are prepared and executed within an application program while the program is executing. In dynamic SQL, the SQL source is contained in host language variables rather than being coded into the application program. The SQL statement can change several times during the application program's execution.

dynamic statement cache pool. A cache, located above the 2-GB storage line, that holds dynamic statements.

E

EA-enabled table space. A table space or index space that is enabled for extended addressability and that contains individual partitions (or pieces, for LOB table spaces) that are greater than 4 GB.

EB. See *exabyte*.

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the z/OS, VM, VSE, and iSeries™ environments. Contrast with *ASCII* and *Unicode*.

e-business. The transformation of key business processes through the use of Internet technologies.

EDM pool. A pool of main storage that is used for database descriptors, application plans, authorization cache, application packages.

EID. Event identifier.

embedded SQL. SQL statements that are coded within an application program. See *static SQL*.

enclave. In Language Environment , an independent collection of routines, one of which is designated as the main routine. An enclave is similar to a program or run unit.

encoding scheme. A set of rules to represent character data (ASCII, EBCDIC, or Unicode).

entity. A significant object of interest to an organization.

enumerated list. A set of DB2 objects that are defined with a LISTDEF utility control statement in which pattern-matching characters (*, %, _ or ?) are not used.

environment. A collection of names of logical and physical resources that are used to support the performance of a function.

environment handle. In DB2 ODBC, the data object that contains global information regarding the state of the application. An environment handle must be allocated before a connection handle can be allocated. Only one environment handle can be allocated per application.

EOM. End of memory.

EOT. End of task.

equijoin. A join operation in which the join-condition has the form *expression = expression*.

error page range. A range of pages that are considered to be physically damaged. DB2 does not allow users to access any pages that fall within this range.

escape character. The symbol that is used to enclose an SQL delimited identifier. The escape character is the double quotation mark ("), except in COBOL applications, where the user assigns the symbol, which is either a double quotation mark or an apostrophe (').

ESDS. Entry sequenced data set.

ESMT. External subsystem module table (in IMS).

EUR. IBM European Standards.

exabyte. For processor, real and virtual storage capacities and channel volume:
1 152 921 504 606 846 976 bytes or 2^{60}.

exception table. A table that holds rows that violate referential constraints or check constraints that the CHECK DATA utility finds.

exclusive lock. A lock that prevents concurrently executing application processes from reading or changing data. Contrast with *share lock*.

executable statement. An SQL statement that can be embedded in an application program, dynamically prepared and executed, or issued interactively.

execution context. In SQLJ, a Java object that can be used to control the execution of SQL statements.

exit routine. A user-written (or IBM-provided default) program that receives control from DB2 to perform specific functions. Exit routines run as extensions of DB2.

expanding conversion. A process that occurs when the length of a converted string is greater than that of the source string. For example, this process occurs when an ASCII mixed-data string that contains DBCS characters is converted to an EBCDIC mixed-data string; the converted string is longer because of the addition of shift codes.

explicit hierarchical locking. Locking that is used to make the parent-child relationship between resources known to IRLM. This kind of locking avoids global locking overhead when no inter-DB2 interest exists on a resource.

exposed name. A correlation name or a table or view name for which a correlation name is not specified. Names that are specified in a FROM clause are exposed or non-exposed.

expression. An operand or a collection of operators and operands that yields a single value.

extended recovery facility (XRF). A facility that minimizes the effect of failures in z/OS, VTAM , the host processor, or high-availability applications during sessions between high-availability applications and designated terminals. This facility provides an alternative subsystem to take over sessions from the failing subsystem.

Extensible Markup Language (XML). A standard metalanguage for defining markup languages that is a subset of Standardized General Markup Language (SGML). The less complex nature of XML makes it easier to write applications that handle document types, to author and manage structured information, and to transmit and share structured information across diverse computing environments.

external function. A function for which the body is written in a programming language that takes scalar argument values and produces a scalar result for each invocation. Contrast with *sourced function*, *built-in function*, and *SQL function*.

external procedure. A user-written application program that can be invoked with the SQL CALL statement, which is written in a programming language. Contrast with *SQL procedure*.

external routine. A user-defined function or stored procedure that is based on code that is written in an external programming language.

external subsystem module table (ESMT). In IMS, the table that specifies which attachment modules must be loaded.

F

failed member state. A state of a member of a data sharing group. When a member fails, the XCF permanently records the failed member state. This state usually means that the member's task, address space, or z/OS system terminated before the state changed from active to quiesced.

fallback. The process of returning to a previous release of DB2 after attempting or completing migration to a current release.

false global lock contention. A contention indication from the coupling facility when multiple lock names are hashed to the same indicator and when no real contention exists.

fan set. A direct physical access path to data, which is provided by an index, hash, or link; a fan set is the means by which the data manager supports the ordering of data.

federated database. The combination of a DB2 Universal Database server (in Linux, UNIX®, and Windows® environments) and multiple data sources to which the server sends queries. In a federated database system, a client application can use a single SQL statement to join data that is distributed across multiple database management systems and can view the data as if it were local.

fetch orientation. The specification of the desired placement of the cursor as part of a FETCH statement (for example, BEFORE, AFTER, NEXT, PRIOR, CURRENT, FIRST, LAST, ABSOLUTE, and RELATIVE).

field procedure. A user-written exit routine that is designed to receive a single value and transform (encode or decode) it in any way the user can specify.

filter factor. A number between zero and one that estimates the proportion of rows in a table for which a predicate is true.

fixed-length string. A character or graphic string whose length is specified and cannot be changed. Contrast with *varying-length string*.

FlashCopy. A function on the IBM Enterprise Storage Server® that can create a point-in-time copy of data while an application is running.

foreign key. A column or set of columns in a dependent table of a constraint relationship. The key must have the same number of columns, with the same descriptions, as the primary key of the parent table.

Each foreign key value must either match a parent key value in the related parent table or be null.

forest. An ordered set of subtrees of XML nodes.

forget. In a two-phase commit operation, (1) the vote that is sent to the prepare phase when the participant has not modified any data. The forget vote allows a participant to release locks and forget about the logical unit of work. This is also referred to as the read-only vote. (2) The response to the *committed* request in the second phase of the operation.

forward log recovery. The third phase of restart processing during which DB2 processes the log in a forward direction to apply all REDO log records.

free space. The total amount of unused space in a page; that is, the space that is not used to store records or control information is free space.

full outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of both tables. See also *join*.

fullselect. A subselect, a values-clause, or a number of both that are combined by set operators. *Fullselect* specifies a result table. If UNION is not used, the result of the fullselect is the result of the specified subselect.

fully escaped mapping. A mapping from an SQL identifier to an XML name when the SQL identifier is a column name.

function. A mapping, which is embodied as a program (the function body) that is invocable by means of zero or more input values (arguments) to a single value (the result). See also *aggregate function* and *scalar function*.

Functions can be user-defined, built-in, or generated by DB2. (See also *built-in function*, *cast function*, *external function*, *sourced function*, *SQL function*, and *user-defined function*.)

function definer. The authorization ID of the owner of the schema of the function that is specified in the CREATE FUNCTION statement.

function implementer. The authorization ID of the owner of the function program and function package.

function package. A package that results from binding the DBRM for a function program.

function package owner. The authorization ID of the user who binds the function program's DBRM into a function package.

function resolution. The process, internal to the DBMS, by which a function invocation is bound to a particular function instance. This process uses the function name, the data types of the arguments, and a list of the applicable schema names (called the *SQL path*) to make the selection. This process is sometimes called *function selection*.

function selection. See *function resolution*.

function signature. The logical concatenation of a fully qualified function name with the data types of all of its parameters.

G

GB. Gigabyte (1 073 741 824 bytes).

GBP. Group buffer pool.

GBP-dependent. The status of a page set or page set partition that is dependent on the group buffer pool. Either read/write interest is active among DB2 subsystems for this page set, or the page set has changed pages in the group buffer pool that have not yet been cast out to disk.

generalized trace facility (GTF). A z/OS service program that records significant system events such as I/O interrupts, SVC interrupts, program interrupts, or external interrupts.

generic resource name. A name that VTAM uses to represent several application programs that provide the same function in order to handle session distribution and balancing in a Sysplex environment.

getpage. An operation in which DB2 accesses a data page.

global lock. A lock that provides concurrency control within and among DB2 subsystems. The scope of the lock is across all DB2 subsystems of a data sharing group.

global lock contention. Conflicts on locking requests between different DB2 members of a data sharing group when those members are trying to serialize shared resources.

governor. See *resource limit facility*.

graphic string. A sequence of DBCS characters.

gross lock. The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group buffer pool (GBP). A coupling facility cache structure that is used by a data sharing group to cache data and to ensure that the data is consistent for all members.

group buffer pool duplexing. The ability to write data to two instances of a group buffer pool structure: a *primary group buffer pool* and a *secondary group buffer*

pool. z/OS publications refer to these instances as the "old" (for primary) and "new" (for secondary) structures.

group level. The release level of a data sharing group, which is established when the first member migrates to a new release.

group name. The z/OS XCF identifier for a data sharing group.

group restart. A restart of at least one member of a data sharing group after the loss of either locks or the shared communications area.

GTF. Generalized trace facility.

H

handle. In DB2 ODBC, a variable that refers to a data structure and associated resources. See also *statement handle*, *connection handle*, and *environment handle*.

help panel. A screen of information that presents tutorial text to assist a user at the workstation or terminal.

heuristic damage. The inconsistency in data between one or more participants that results when a heuristic decision to resolve an indoubt LUW at one or more participants differs from the decision that is recorded at the coordinator.

heuristic decision. A decision that forces indoubt resolution at a participant by means other than automatic resynchronization between coordinator and participant.

hole. A row of the result table that cannot be accessed because of a delete or an update that has been performed on the row. See also *delete hole* and *update hole*.

home address space. The area of storage that z/OS currently recognizes as *dispatched*.

host. The set of programs and resources that are available on a given TCP/IP instance.

host expression. A Java variable or expression that is referenced by SQL clauses in an SQLJ application program.

host identifier. A name that is declared in the host program.

host language. A programming language in which you can embed SQL statements.

host program. An application program that is written in a host language and that contains embedded SQL statements.

host structure. In an application program, a structure that is referenced by embedded SQL statements.

host variable. In an application program, an application variable that is referenced by embedded SQL statements.

host variable array. An array of elements, each of which corresponds to a value for a column. The dimension of the array determines the maximum number of rows for which the array can be used.

HSM. Hierarchical storage manager.

HTML. Hypertext Markup Language, a standard method for presenting Web data to users.

HTTP. Hypertext Transfer Protocol, a communication protocol that the Web uses.

I

ICF. Integrated catalog facility.

IDCAMS. An IBM program that is used to process access method services commands. It can be invoked as a job or jobstep, from a TSO terminal, or from within a user's application program.

IDCAMS LISTCAT. A facility for obtaining information that is contained in the access method services catalog.

identify. A request that an attachment service program in an address space that is separate from DB2 issues thorough the z/OS subsystem interface to inform DB2 of its existence and to initiate the process of becoming connected to DB2.

identity column. A column that provides a way for DB2 to automatically generate a numeric value for each row. The generated values are unique if cycling is not used. Identity columns are defined with the AS IDENTITY clause. Uniqueness of values can be ensured by defining a unique index that contains only the identity column. A table can have no more than one identity column.

IFCID. Instrumentation facility component identifier.

IFI. Instrumentation facility interface.

IFI call. An invocation of the instrumentation facility interface (IFI) by means of one of its defined functions.

IFP. IMS Fast Path.

image copy. An exact reproduction of all or part of a table space. DB2 provides utility programs to make full image copies (to copy the entire table space) or incremental image copies (to copy only those pages that have been modified since the last image copy).

implied forget. In the presumed-abort protocol, an implied response of *forget* to the second-phase *committed* request from the coordinator. The response is implied when the participant responds to any subsequent request from the coordinator.

IMS. Information Management System.

IMS attachment facility. A DB2 subcomponent that uses z/OS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

IMS DB. Information Management System Database.

IMS TM. Information Management System Transaction Manager.

in-abort. A status of a unit of recovery. If DB2 fails after a unit of recovery begins to be rolled back, but before the process is completed, DB2 continues to back out the changes during restart.

in-commit. A status of a unit of recovery. If DB2 fails after beginning its phase 2 commit processing, it "knows," when restarted, that changes made to data are consistent. Such units of recovery are termed *in-commit*.

independent. An object (row, table, or table space) that is neither a parent nor a dependent of another object.

index. A set of pointers that are logically ordered by the values of a key. Indexes can provide faster access to data and can enforce uniqueness on the rows in a table.

index-controlled partitioning. A type of partitioning in which partition boundaries for a partitioned table are controlled by values that are specified on the CREATE INDEX statement. Partition limits are saved in the LIMITKEY column of the SYSIBM.SYSINDEXPART catalog table.

index key. The set of columns in a table that is used to determine the order of index entries.

index partition. A VSAM data set that is contained within a partitioning index space.

index space. A page set that is used to store the entries of one index.

indicator column. A 4-byte value that is stored in a base table in place of a LOB column.

indicator variable. A variable that is used to represent the null value in an application program. If the value for the selected column is null, a negative value is placed in the indicator variable.

indoubt. A status of a unit of recovery. If DB2 fails after it has finished its phase 1 commit processing and before it has started phase 2, only the commit coordinator knows if an individual unit of recovery is to be committed or rolled back. At emergency restart, if DB2 lacks the information it needs to make this decision, the status of the unit of recovery is *indoubt* until DB2 obtains this information from the coordinator. More than one unit of recovery can be indoubt at restart.

indoubt resolution. The process of resolving the status of an indoubt logical unit of work to either the committed or the rollback state.

inflight. A status of a unit of recovery. If DB2 fails before its unit of recovery completes phase 1 of the commit process, it merely backs out the updates of its unit of recovery at restart. These units of recovery are termed *inflight*.

inheritance. The passing downstream of class resources or attributes from a parent class in the class hierarchy to a child class.

initialization file. For DB2 ODBC applications, a file containing values that can be set to adjust the performance of the database manager.

inline copy. A copy that is produced by the LOAD or REORG utility. The data set that the inline copy produces is logically equivalent to a full image copy that is produced by running the COPY utility with read-only access (SHRLEVEL REFERENCE).

inner join. The result of a join operation that includes only the matched rows of both tables that are being joined. See also *join*.

inoperative package. A package that cannot be used because one or more user-defined functions or procedures that the package depends on were dropped. Such a package must be explicitly rebound. Contrast with *invalid package.*

insensitive cursor. A cursor that is not sensitive to inserts, updates, or deletes that are made to the underlying rows of a result table after the result table has been materialized.

insert trigger. A trigger that is defined with the triggering SQL operation INSERT.

install. The process of preparing a DB2 subsystem to operate as a z/OS subsystem.

installation verification scenario. A sequence of operations that exercises the main DB2 functions and tests whether DB2 was correctly installed.

instrumentation facility component identifier (IFCID). A value that names and identifies a trace record of an event that can be traced. As a parameter on the START TRACE and MODIFY TRACE commands, it specifies that the corresponding event is to be traced.

instrumentation facility interface (IFI). A programming interface that enables programs to obtain online trace data about DB2, to submit DB2 commands, and to pass data to DB2.

Interactive System Productivity Facility (ISPF). An IBM licensed program that provides interactive dialog services in a z/OS environment.

inter-DB2 R/W interest. A property of data in a table space, index, or partition that has been opened by more than one member of a data sharing group and that has been opened for writing by at least one of those members.

intermediate database server. The target of a request from a local application or a remote application requester that is forwarded to another database server. In the DB2 environment, the remote request is forwarded transparently to another database server if the object that is referenced by a three-part name does not reference the local location.

internationalization. The support for an encoding scheme that is able to represent the code points of characters from many different geographies and languages. To support all geographies, the Unicode standard requires more than 1 byte to represent a single character. See also *Unicode*.

internal resource lock manager (IRLM). A z/OS subsystem that DB2 uses to control communication and database locking.

International Organization for Standardization. An international body charged with creating standards to facilitate the exchange of goods and services as well as cooperation in intellectual, scientific, technological, and economic activity.

invalid package. A package that depends on an object (other than a user-defined function) that is dropped. Such a package is implicitly rebound on invocation. Contrast with *inoperative package.*

invariant character set. (1) A character set, such as the syntactic character set, whose code point assignments do not change from code page to code page. (2) A minimum set of characters that is available as part of all character sets.

IP address. A 4-byte value that uniquely identifies a TCP/IP host.

IRLM. Internal resource lock manager.

ISO. International Organization for Standardization.

isolation level. The degree to which a unit of work is isolated from the updating operations of other units of work. See also *cursor stability*, *read stability*, *repeatable read*, and *uncommitted read*.

ISPF. Interactive System Productivity Facility.

ISPF/PDF. Interactive System Productivity Facility/Program Development Facility.

iterator. In SQLJ, an object that contains the result set of a query. An iterator is equivalent to a cursor in other host languages.

iterator declaration clause. In SQLJ, a statement that generates an iterator declaration class. An iterator is an object of an iterator declaration class.

J

Japanese Industrial Standard. An encoding scheme that is used to process Japanese characters.

JAR. Java Archive.

Java Archive (JAR). A file format that is used for aggregating many files into a single file.

JCL. Job control language.

JDBC. A Sun Microsystems database application programming interface (API) for Java that allows programs to access database management systems by using callable SQL. JDBC does not require the use of an SQL preprocessor. In addition, JDBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time.

JES. Job Entry Subsystem.

JIS. Japanese Industrial Standard.

job control language (JCL). A control language that is used to identify a job to an operating system and to describe the job's requirements.

Job Entry Subsystem (JES). An IBM licensed program that receives jobs into the system and processes all output data that is produced by the jobs.

join. A relational operation that allows retrieval of data from two or more tables based on matching column values. See also *equijoin*, *full outer join*, *inner join*, *left outer join*, *outer join*, *and right outer join*.

K

KB. Kilobyte (1024 bytes).

Kerberos. A network authentication protocol that is designed to provide strong authentication for client/server applications by using secret-key cryptography.

Kerberos ticket. A transparent application mechanism that transmits the identity of an initiating principal to its target. A simple ticket contains the principal's

identity, a session key, a timestamp, and other information, which is sealed using the target's secret key.

key. A column or an ordered collection of columns that is identified in the description of a table, index, or referential constraint. The same column can be part of more than one key.

key-sequenced data set (KSDS). A VSAM file or data set whose records are loaded in key sequence and controlled by an index.

keyword. In SQL, a name that identifies an option that is used in an SQL statement.

KSDS. Key-sequenced data set.

L

labeled duration. A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

large object (LOB). A sequence of bytes representing bit data, single-byte characters, double-byte characters, or a mixture of single- and double-byte characters. A LOB can be up to 2 GB–1 byte in length. See also *BLOB*, *CLOB*, and *DBCLOB*.

last agent optimization. An optimized commit flow for either presumed-nothing or presumed-abort protocols in which the last agent, or final participant, becomes the commit coordinator. This flow saves at least one message.

latch. A DB2 internal mechanism for controlling concurrent events or the use of system resources.

LCID. Log control interval definition.

LDS. Linear data set.

leaf page. A page that contains pairs of keys and RIDs and that points to actual data. Contrast with *nonleaf page*.

left outer join. The result of a join operation that includes the matched rows of both tables that are being joined, and that preserves the unmatched rows of the first table. See also *join*.

limit key. The highest value of the index key for a partition.

linear data set (LDS). A VSAM data set that contains data but no control information. A linear data set can be accessed as a byte-addressable string in virtual storage.

linkage editor. A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses.

link-edit. The action of creating a loadable computer program using a linkage editor.

list. A type of object, which DB2 utilities can process, that identifies multiple table spaces, multiple index spaces, or both. A list is defined with the LISTDEF utility control statement.

list structure. A coupling facility structure that lets data be shared and manipulated as elements of a queue.

LLE. Load list element.

L-lock. Logical lock.

load list element. A z/OS control block that controls the loading and deleting of a particular load module based on entry point names.

load module. A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

LOB. Large object.

LOB locator. A mechanism that allows an application program to manipulate a large object value in the database system. A LOB locator is a fullword integer value that represents a single LOB value. An application program retrieves a LOB locator into a host variable and can then apply SQL operations to the associated LOB value using the locator.

LOB lock. A lock on a LOB value.

LOB table space. A table space in an auxiliary table that contains all the data for a particular LOB column in the related base table.

local. A way of referring to any object that the local DB2 subsystem maintains. A *local table*, for example, is a table that is maintained by the local DB2 subsystem. Contrast with *remote*.

locale. The definition of a subset of a user's environment that combines a CCSID and characters that are defined for a specific language and country.

local lock. A lock that provides intra-DB2 concurrency control, but not inter-DB2 concurrency control; that is, its scope is a single DB2.

local subsystem. The unique relational DBMS to which the user or application program is directly connected (in the case of DB2, by one of the DB2 attachment facilities).

location. The unique name of a database server. An application uses the location name to access a DB2

database server. A database alias can be used to override the location name when accessing a remote server.

location alias. Another name by which a database server identifies itself in the network. Applications can use this name to access a DB2 database server.

lock. A means of controlling concurrent events or access to data. DB2 locking is performed by the IRLM.

lock duration. The interval over which a DB2 lock is held.

lock escalation. The promotion of a lock from a row, page, or LOB lock to a table space lock because the number of page locks that are concurrently held on a given resource exceeds a preset limit.

locking. The process by which the integrity of data is ensured. Locking prevents concurrent users from accessing inconsistent data.

lock mode. A representation for the type of access that concurrently running programs can have to a resource that a DB2 lock is holding.

lock object. The resource that is controlled by a DB2 lock.

lock promotion. The process of changing the size or mode of a DB2 lock to a higher, more restrictive level.

lock size. The amount of data that is controlled by a DB2 lock on table data; the value can be a row, a page, a LOB, a partition, a table, or a table space.

lock structure. A coupling facility data structure that is composed of a series of lock entries to support shared and exclusive locking for logical resources.

log. A collection of records that describe the events that occur during DB2 execution and that indicate their sequence. The information thus recorded is used for recovery in the event of a failure during DB2 execution.

log control interval definition. A suffix of the physical log record that tells how record segments are placed in the physical control interval.

logical claim. A claim on a logical partition of a nonpartitioning index.

logical data modeling. The process of documenting the comprehensive business information requirements in an accurate and consistent format. Data modeling is the first task of designing a database.

logical drain. A drain on a logical partition of a nonpartitioning index.

logical index partition. The set of all keys that reference the same data partition.

logical lock (L-lock). The lock type that transactions use to control intra- and inter-DB2 data concurrency between transactions. Contrast with *physical lock (P-lock)*.

logically complete. A state in which the concurrent copy process is finished with the initialization of the target objects that are being copied. The target objects are available for update.

logical page list (LPL). A list of pages that are in error and that cannot be referenced by applications until the pages are recovered. The page is in *logical error* because the actual media (coupling facility or disk) might not contain any errors. Usually a connection to the media has been lost.

logical partition. A set of key or RID pairs in a nonpartitioning index that are associated with a particular partition.

logical recovery pending (LRECP). The state in which the data and the index keys that reference the data are inconsistent.

logical unit (LU). An access point through which an application program accesses the SNA network in order to communicate with another application program.

logical unit of work (LUW). The processing that a program performs between synchronization points.

logical unit of work identifier (LUWID). A name that uniquely identifies a thread within a network. This name consists of a fully-qualified LU network name, an LUW instance number, and an LUW sequence number.

log initialization. The first phase of restart processing during which DB2 attempts to locate the current end of the log.

log record header (LRH). A prefix, in every logical record, that contains control information.

log record sequence number (LRSN). A unique identifier for a log record that is associated with a data sharing member. DB2 uses the LRSN for recovery in the data sharing environment.

log truncation. A process by which an explicit starting RBA is established. This RBA is the point at which the next byte of log data is to be written.

LPL. Logical page list.

LRECP. Logical recovery pending.

LRH. Log record header.

LRSN. Log record sequence number.

LU. Logical unit.

LU name. Logical unit name, which is the name by which VTAM refers to a node in a network. Contrast with *location name*.

LUW. Logical unit of work.

LUWID. Logical unit of work identifier.

M

mapping table. A table that the REORG utility uses to map the associations of the RIDs of data records in the original copy and in the shadow copy. This table is created by the user.

mass delete. The deletion of all rows of a table.

master terminal. The IMS logical terminal that has complete control of IMS resources during online operations.

master terminal operator (MTO). See *master terminal.*

materialize. (1) The process of putting rows from a view or nested table expression into a work file for additional processing by a query.

(2) The placement of a LOB value into contiguous storage. Because LOB values can be very large, DB2 avoids materializing LOB data until doing so becomes absolutely necessary.

| **materialized query table.** A table that is used to
| contain information that is derived and can be
| summarized from one or more source tables.

MB. Megabyte (1 048 576 bytes).

MBCS. Multibyte character set. UTF-8 is an example of an MBCS. Characters in UTF-8 can range from 1 to 4 bytes in DB2.

member name. The z/OS XCF identifier for a particular DB2 subsystem in a data sharing group.

menu. A displayed list of available functions for selection by the operator. A menu is sometimes called a *menu panel*.

| **metalanguage.** A language that is used to create other
| specialized languages.

migration. The process of converting a subsystem with a previous release of DB2 to an updated or current release. In this process, you can acquire the functions of the updated or current release without losing the data that you created on the previous release.

mixed data string. A character string that can contain both single-byte and double-byte characters.

MLPA. Modified link pack area.

MODEENT. A VTAM macro instruction that associates a logon mode name with a set of parameters representing session protocols. A set of MODEENT macro instructions defines a logon mode table.

modeling database. A DB2 database that you create on your workstation that you use to model a DB2 UDB for z/OS subsystem, which can then be evaluated by the Index Advisor.

mode name. A VTAM name for the collection of physical and logical characteristics and attributes of a session.

modify locks. An L-lock or P-lock with a MODIFY attribute. A list of these active locks is kept at all times in the coupling facility lock structure. If the requesting DB2 subsystem fails, that DB2 subsystem's modify locks are converted to retained locks.

MPP. Message processing program (in IMS).

MTO. Master terminal operator.

multibyte character set (MBCS). A character set that represents single characters with more than a single byte. Contrast with *single-byte character set* and *double-byte character set*. See also *Unicode*.

multidimensional analysis. The process of assessing and evaluating an enterprise on more than one level.

Multiple Virtual Storage. An element of the z/OS operating system. This element is also called the Base Control Program (BCP).

multisite update. Distributed relational database processing in which data is updated in more than one location within a single unit of work.

multithreading. Multiple TCBs that are executing one copy of DB2 ODBC code concurrently (sharing a processor) or in parallel (on separate central processors).

must-complete. A state during DB2 processing in which the entire operation must be completed to maintain data integrity.

mutex. Pthread mutual exclusion; a lock. A Pthread mutex variable is used as a locking mechanism to allow serialization of critical sections of code by temporarily blocking the execution of all but one thread.

| **MVS.** See *Multiple Virtual Storage*.

N

negotiable lock. A lock whose mode can be downgraded, by agreement among contending users, to be compatible to all. A physical lock is an example of a negotiable lock.

nested table expression. A fullselect in a FROM clause (surrounded by parentheses).

network identifier (NID). The network ID that is assigned by IMS or CICS, or if the connection type is RRSAF, the RRS unit of recovery ID (URID).

NID. Network identifier.

nonleaf page. A page that contains keys and page numbers of other pages in the index (either leaf or nonleaf pages). Nonleaf pages never point to actual data.

nonpartitioned index. An index that is not physically partitioned. Both partitioning indexes and secondary indexes can be nonpartitioned.

nonscrollable cursor. A cursor that can be moved only in a forward direction. Nonscrollable cursors are sometimes called forward-only cursors or serial cursors.

normalization. A key step in the task of building a logical relational database design. Normalization helps you avoid redundancies and inconsistencies in your data. An entity is normalized if it meets a set of constraints for a particular normal form (first normal form, second normal form, and so on). Contrast with *denormalization*.

nondeterministic function. A user-defined function whose result is not solely dependent on the values of the input arguments. That is, successive invocations with the same argument values can produce a different answer. this type of function is sometimes called a *variant* function. Contrast this with a *deterministic function* (sometimes called a *not-variant function*), which always produces the same result for the same inputs.

not-variant function. See *deterministic function*.

NPSI. See *nonpartitioned secondary index.*

NRE. Network recovery element.

NUL. The null character ('\0'), which is represented by the value X'00'. In C, this character denotes the end of a string.

null. A special value that indicates the absence of information.

NULLIF. A scalar function that evaluates two passed expressions, returning either NULL if the arguments are equal or the value of the first argument if they are not.

null-terminated host variable. A varying-length host variable in which the end of the data is indicated by a null terminator.

null terminator. In C, the value that indicates the end of a string. For EBCDIC, ASCII, and Unicode UTF-8 strings, the null terminator is a single-byte value (X'00'). For Unicode UCS-2 (wide) strings, the null terminator is a double-byte value (X'0000').

O

OASN (origin application schedule number). In IMS, a 4-byte number that is assigned sequentially to each IMS schedule since the last cold start of IMS. The OASN is used as an identifier for a unit of work. In an 8-byte format, the first 4 bytes contain the schedule number and the last 4 bytes contain the number of IMS sync points (*commit points*) during the current schedule. The OASN is part of the NID for an IMS connection.

ODBC. Open Database Connectivity.

ODBC driver. A dynamically-linked library (DLL) that implements ODBC function calls and interacts with a data source.

OBID. Data object identifier.

Open Database Connectivity (ODBC). A Microsoft® database application programming interface (API) for C that allows access to database management systems by using callable SQL. ODBC does not require the use of an SQL preprocessor. In addition, ODBC provides an architecture that lets users add modules called *database drivers*, which link the application to their choice of database management systems at run time. This means that applications no longer need to be directly linked to the modules of all the database management systems that are supported.

ordinary identifier. An uppercase letter followed by zero or more characters, each of which is an uppercase letter, a digit, or the underscore character. An ordinary identifier must not be a reserved word.

ordinary token. A numeric constant, an ordinary identifier, a host identifier, or a keyword.

originating task. In a parallel group, the primary agent that receives data from other execution units (referred to as *parallel tasks*) that are executing portions of the query in parallel.

OS/390. Operating System/390®.

outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves some or all of the unmatched rows of the tables that are being joined. See also *join*.

overloaded function. A function name for which multiple function instances exist.

P

package. An object containing a set of SQL statements that have been statically bound and that is available for processing. A package is sometimes also called an *application package*.

package list. An ordered list of package names that may be used to extend an application plan.

package name. The name of an object that is created by a BIND PACKAGE or REBIND PACKAGE command. The object is a bound version of a database request module (DBRM). The name consists of a location name, a collection ID, a package ID, and a version ID.

page. A unit of storage within a table space (4 KB, 8 KB, 16 KB, or 32 KB) or index space (4 KB). In a table space, a page contains one or more rows of a table. In a LOB table space, a LOB value can span more than one page, but no more than one LOB value is stored on a page.

page set. Another way to refer to a table space or index space. Each page set consists of a collection of VSAM data sets.

page set recovery pending (PSRCP). A restrictive state of an index space. In this case, the entire page set must be recovered. Recovery of a logical part is prohibited.

panel. A predefined display image that defines the locations and characteristics of display fields on a display surface (for example, a *menu panel*).

parallel complex. A cluster of machines that work together to handle multiple transactions and applications.

parallel group. A set of consecutive operations that execute in parallel and that have the same number of parallel tasks.

parallel I/O processing. A form of I/O processing in which DB2 initiates multiple concurrent requests for a single user query and performs I/O processing concurrently (in *parallel*) on multiple data partitions.

parallelism assistant. In Sysplex query parallelism, a DB2 subsystem that helps to process parts of a parallel query that originates on another DB2 subsystem in the data sharing group.

parallelism coordinator. In Sysplex query parallelism, the DB2 subsystem from which the parallel query originates.

Parallel Sysplex. A set of z/OS systems that communicate and cooperate with each other through certain multisystem hardware components and software services to process customer workloads.

parallel task. The execution unit that is dynamically created to process a query in parallel. A parallel task is implemented by a z/OS service request block.

parameter marker. A question mark (?) that appears in a statement string of a dynamic SQL statement. The question mark can appear where a host variable could appear if the statement string were a static SQL statement.

parameter-name. An SQL identifier that designates a parameter in an SQL procedure or an SQL function.

parent key. A primary key or unique key in the parent table of a referential constraint. The values of a parent key determine the valid values of the foreign key in the referential constraint.

parent lock. For explicit hierarchical locking, a lock that is held on a resource that might have child locks that are lower in the hierarchy. A parent lock is usually the table space lock or the partition intent lock. See also *child lock*.

parent row. A row whose primary key value is the foreign key value of a dependent row.

parent table. A table whose primary key is referenced by the foreign key of a dependent table.

parent table space. A table space that contains a parent table. A table space containing a dependent of that table is a dependent table space.

participant. An entity other than the commit coordinator that takes part in the commit process. The term participant is synonymous with *agent* in SNA.

partition. A portion of a page set. Each partition corresponds to a single, independently extendable data set. Partitions can be extended to a maximum size of 1, 2, or 4 GB, depending on the number of partitions in the partitioned page set. All partitions of a given page set have the same maximum size.

partitioned data set (PDS). A data set in disk storage that is divided into partitions, which are called members. Each partition can contain a program, part of a program, or data. The term partitioned data set is synonymous with program library.

partitioned index. An index that is physically partitioned. Both partitioning indexes and secondary indexes can be partitioned.

partitioned page set. A partitioned table space or an index space. Header pages, space map pages, data pages, and index pages reference data only within the scope of the partition.

partitioned table space. A table space that is subdivided into parts (based on index key range), each of which can be processed independently by utilities.

partitioning index. An index in which the leftmost columns are the partitioning columns of the table. The index can be partitioned or nonpartitioned.

partition pruning. The removal from consideration of inapplicable partitions through setting up predicates in a query on a partitioned table to access only certain partitions to satisfy the query.

partner logical unit. An access point in the SNA network that is connected to the local DB2 subsystem by way of a VTAM conversation.

path. See *SQL path*.

PCT. Program control table (in CICS).

PDS. Partitioned data set.

piece. A data set of a nonpartitioned page set.

physical claim. A claim on an entire nonpartitioning index.

physical consistency. The state of a page that is not in a partially changed state.

physical drain. A drain on an entire nonpartitioning index.

physical lock (P-lock). A type of lock that DB2 acquires to provide consistency of data that is cached in different DB2 subsystems. Physical locks are used only in data sharing environments. Contrast with *logical lock (L-lock)*.

physical lock contention. Conflicting states of the requesters for a physical lock. See also *negotiable lock*.

physically complete. The state in which the concurrent copy process is completed and the output data set has been created.

plan. See *application plan*.

plan allocation. The process of allocating DB2 resources to a plan in preparation for execution.

plan member. The bound copy of a DBRM that is identified in the member clause.

plan name. The name of an application plan.

plan segmentation. The dividing of each plan into sections. When a section is needed, it is independently brought into the EDM pool.

P-lock. Physical lock.

PLT. Program list table (in CICS).

point of consistency. A time when all recoverable data that an application accesses is consistent with other data. The term point of consistency is synonymous with *sync point* or *commit point*.

policy. See *CFRM policy*.

Portable Operating System Interface (POSIX). The IEEE operating system interface standard, which defines the Pthread standard of threading. See also *Pthread*.

POSIX. Portable Operating System Interface.

postponed abort UR. A unit of recovery that was inflight or in-abort, was interrupted by system failure or cancellation, and did not complete backout during restart.

PPT. (1) Processing program table (in CICS). (2) Program properties table (in z/OS).

precision. In SQL, the total number of digits in a decimal number (called the *size* in the C language). In the C language, the number of digits to the right of the decimal point (called the *scale* in SQL). The DB2 library uses the SQL terms.

precompilation. A processing of application programs containing SQL statements that takes place before compilation. SQL statements are replaced with statements that are recognized by the host language compiler. Output from this precompilation includes source code that can be submitted to the compiler and the database request module (DBRM) that is input to the bind process.

predicate. An element of a search condition that expresses or implies a comparison operation.

prefix. A code at the beginning of a message or record.

preformat. The process of preparing a VSAM ESDS for DB2 use, by writing specific data patterns.

prepare. The first phase of a two-phase commit process in which all participants are requested to prepare for commit.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

presumed-abort. An optimization of the presumed-nothing two-phase commit protocol that reduces the number of recovery log records, the duration of state maintenance, and the number of messages between coordinator and participant. The optimization also modifies the indoubt resolution responsibility.

presumed-nothing. The standard two-phase commit protocol that defines coordinator and participant responsibilities, relative to logical unit of work states, recovery logging, and indoubt resolution.

primary authorization ID. The authorization ID that is used to identify the application process to DB2.

primary group buffer pool. For a duplexed group buffer pool, the structure that is used to maintain the coherency of cached data. This structure is used for page registration and cross-invalidation. The z/OS equivalent is *old* structure. Compare with *secondary group buffer pool*.

primary index. An index that enforces the uniqueness of a primary key.

primary key. In a relational database, a unique, nonnull key that is part of the definition of a table. A table cannot be defined as a parent unless it has a unique key or primary key.

principal. An entity that can communicate securely with another entity. In Kerberos, principals are represented as entries in the Kerberos registry database and include users, servers, computers, and others.

principal name. The name by which a principal is known to the DCE security services.

private connection. A communications connection that is specific to DB2.

private protocol access. A method of accessing distributed data by which you can direct a query to another DB2 system. Contrast with *DRDA access*.

private protocol connection. A DB2 private connection of the application process. See also *private connection*.

privilege. The capability of performing a specific function, sometimes on a specific object. The types of privileges are:

explicit privileges, which have names and are held as the result of SQL GRANT and REVOKE statements. For example, the SELECT privilege.
implicit privileges, which accompany the ownership of an object, such as the privilege to drop a synonym that one owns, or the holding of an authority, such as the privilege of SYSADM authority to terminate any utility job.

privilege set. For the installation SYSADM ID, the set of all possible privileges. For any other authorization ID, the set of all privileges that are recorded for that ID in the DB2 catalog.

process. In DB2, the unit to which DB2 allocates resources and locks. Sometimes called an *application process*, a process involves the execution of one or more programs. The execution of an SQL statement is always associated with some process. The means of initiating and terminating a process are dependent on the environment.

program. A single, compilable collection of executable statements in a programming language.

program temporary fix (PTF). A solution or bypass of a problem that is diagnosed as a result of a defect in a current unaltered release of a licensed program. An authorized program analysis report (APAR) fix is corrective service for an existing problem. A PTF is preventive service for problems that might be encountered by other users of the product. A PTF is *temporary*, because a permanent fix is usually not incorporated into the product until its next release.

protected conversation. A VTAM conversation that supports two-phase commit flows.

PSRCP. Page set recovery pending.

PTF. Program temporary fix.

Pthread. The POSIX threading standard model for splitting an application into subtasks. The Pthread standard includes functions for creating threads, terminating threads, synchronizing threads through locking, and other thread control facilities.

Q

QMF™. Query Management Facility.

QSAM. Queued sequential access method.

query. A component of certain SQL statements that specifies a result table.

query block. The part of a query that is represented by one of the FROM clauses. Each FROM clause can have multiple query blocks, depending on DB2's internal processing of the query.

query CP parallelism. Parallel execution of a single query, which is accomplished by using multiple tasks. See also *Sysplex query parallelism*.

query I/O parallelism. Parallel access of data, which is accomplished by triggering multiple I/O requests within a single query.

queued sequential access method (QSAM). An extended version of the basic sequential access method (BSAM). When this method is used, a queue of data blocks is formed. Input data blocks await processing, and output data blocks await transfer to auxiliary storage or to an output device.

quiesce point. A point at which data is consistent as a result of running the DB2 QUIESCE utility.

quiesced member state. A state of a member of a data sharing group. An active member becomes quiesced when a STOP DB2 command takes effect without a failure. If the member's task, address space, or z/OS system fails before the command takes effect, the member state is failed.

R

RACF. Resource Access Control Facility, which is a component of the z/OS Security Server.

RAMAC®. IBM family of enterprise disk storage system products.

RBA. Relative byte address.

RCT. Resource control table (in CICS attachment facility).

RDB. Relational database.

RDBMS. Relational database management system.

RDBNAM. Relational database name.

RDF. Record definition field.

read stability (RS). An isolation level that is similar to repeatable read but does not completely isolate an application process from all other concurrently executing application processes. Under level RS, an application that issues the same query more than once might read additional rows that were inserted and committed by a concurrently executing application process.

rebind. The creation of a new application plan for an application program that has been bound previously. If, for example, you have added an index for a table that your application accesses, you must rebind the application in order to take advantage of that index.

rebuild. The process of reallocating a coupling facility structure. For the shared communications area (SCA) and lock structure, the structure is repopulated; for the group buffer pool, changed pages are usually cast out to disk, and the new structure is populated only with changed pages that were not successfully cast out.

RECFM. Record format.

record. The storage representation of a row or other data.

record identifier (RID). A unique identifier that DB2 uses internally to identify a row of data in a table. Compare with *row ID*.

record identifier (RID) pool. An area of main storage that is used for sorting record identifiers during list-prefetch processing.

record length. The sum of the length of all the columns in a table, which is the length of the data as it is physically stored in the database. Records can be fixed length or varying length, depending on how the columns are defined. If all columns are fixed-length columns, the record is a fixed-length record. If one or more columns are varying-length columns, the record is a varying-length column.

Recoverable Resource Manager Services attachment facility (RRSAF). A DB2 subcomponent that uses Resource Recovery Services to coordinate resource commitment between DB2 and all other resource managers that also use RRS in a z/OS system.

recovery. The process of rebuilding databases after a system failure.

recovery log. A collection of records that describes the events that occur during DB2 execution and indicates their sequence. The recorded information is used for recovery in the event of a failure during DB2 execution.

recovery manager. (1) A subcomponent that supplies coordination services that control the interaction of DB2 resource managers during commit, abort, checkpoint, and restart processes. The recovery manager also supports the recovery mechanisms of other subsystems (for example, IMS) by acting as a participant in the other subsystem's process for protecting data that has reached a point of consistency. (2) A coordinator or a participant (or both), in the execution of a two-phase commit, that can access a recovery log that maintains the state of the logical unit of work and names the immediate upstream coordinator and downstream participants.

recovery pending (RECP). A condition that prevents SQL access to a table space that needs to be recovered.

recovery token. An identifier for an element that is used in recovery (for example, NID or URID).

RECP. Recovery pending.

redo. A state of a unit of recovery that indicates that changes are to be reapplied to the disk media to ensure data integrity.

reentrant. Executable code that can reside in storage as one shared copy for all threads. Reentrant code is not self-modifying and provides separate storage areas for each thread. Reentrancy is a compiler and operating system concept, and reentrancy alone is not enough to guarantee logically consistent results when multithreading. See also *threadsafe*.

referential constraint. The requirement that nonnull values of a designated foreign key are valid only if they equal values of the primary key of a designated table.

referential integrity. The state of a database in which all values of all foreign keys are valid. Maintaining referential integrity requires the enforcement of referential constraints on all operations that change the data in a table on which the referential constraints are defined.

referential structure. A set of tables and relationships that includes at least one table and, for every table in the set, all the relationships in which that table participates and all the tables to which it is related.

refresh age. The time duration between the current time and the time during which a materialized query table was last refreshed.

registry. See *registry database*.

registry database. A database of security information about principals, groups, organizations, accounts, and security policies.

relational database (RDB). A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

relational database management system (RDBMS). A collection of hardware and software that organizes and provides access to a relational database.

relational database name (RDBNAM). A unique identifier for an RDBMS within a network. In DB2, this must be the value in the LOCATION column of table SYSIBM.LOCATIONS in the CDB. DB2 publications refer to the name of another RDBMS as a LOCATION value or a location name.

relationship. A defined connection between the rows of a table or the rows of two tables. A relationship is the internal representation of a referential constraint.

relative byte address (RBA). The offset of a data record or control interval from the beginning of the storage space that is allocated to the data set or file to which it belongs.

remigration. The process of returning to a current release of DB2 following a fallback to a previous release. This procedure constitutes another migration process.

remote. Any object that is maintained by a remote DB2 subsystem (that is, by a DB2 subsystem other than the local one). A *remote view*, for example, is a view that is maintained by a remote DB2 subsystem. Contrast with *local*.

remote attach request. A request by a remote location to attach to the local DB2 subsystem. Specifically, the request that is sent is an SNA Function Management Header 5.

remote subsystem. Any relational DBMS, except the *local subsystem*, with which the user or application can communicate. The subsystem need not be remote in any physical sense, and might even operate on the same processor under the same z/OS system.

reoptimization. The DB2 process of reconsidering the access path of an SQL statement at run time; during reoptimization, DB2 uses the values of host variables, parameter markers, or special registers.

REORG pending (REORP). A condition that restricts SQL access and most utility access to an object that must be reorganized.

REORP. REORG pending.

repeatable read (RR). The isolation level that provides maximum protection from other executing application programs. When an application program executes with repeatable read protection, rows that the program references cannot be changed by other programs until the program reaches a commit point.

repeating group. A situation in which an entity includes multiple attributes that are inherently the same. The presence of a repeating group violates the requirement of first normal form. In an entity that satisfies the requirement of first normal form, each attribute is independent and unique in its meaning and its name. See also *normalization*.

replay detection mechanism. A method that allows a principal to detect whether a request is a valid request from a source that can be trusted or whether an untrustworthy entity has captured information from a previous exchange and is replaying the information exchange to gain access to the principal.

request commit. The vote that is submitted to the prepare phase if the participant has modified data and is prepared to commit or roll back.

requester. The source of a request to access data at a remote server. In the DB2 environment, the requester function is provided by the distributed data facility.

resource. The object of a lock or claim, which could be a table space, an index space, a data partition, an index partition, or a logical partition.

resource allocation. The part of plan allocation that deals specifically with the database resources.

resource control table (RCT). A construct of the CICS attachment facility, created by site-provided macro parameters, that defines authorization and access attributes for transactions or transaction groups.

resource definition online. A CICS feature that you use to define CICS resources online without assembling tables.

resource limit facility (RLF). A portion of DB2 code that prevents dynamic manipulative SQL statements from exceeding specified time limits. The resource limit facility is sometimes called the governor.

resource limit specification table (RLST). A site-defined table that specifies the limits to be enforced by the resource limit facility.

resource manager. (1) A function that is responsible for managing a particular resource and that guarantees the consistency of all updates made to recoverable resources within a logical unit of work. The resource that is being managed can be physical (for example, disk or main storage) or logical (for example, a particular type of system service). (2) A participant, in the execution of a two-phase commit, that has recoverable resources that could have been modified. The resource manager has access to a recovery log so that it can commit or roll back the effects of the logical unit of work to the recoverable resources.

restart pending (RESTP). A restrictive state of a page set or partition that indicates that restart (backout) work needs to be performed on the object. All access to the page set or partition is denied except for access by the:
- RECOVER POSTPONED command
- Automatic online backout (which DB2 invokes after restart if the system parameter LBACKOUT=AUTO)

RESTP. Restart pending.

result set. The set of rows that a stored procedure returns to a client application.

result set locator. A 4-byte value that DB2 uses to uniquely identify a query result set that a stored procedure returns.

result table. The set of rows that are specified by a SELECT statement.

retained lock. A MODIFY lock that a DB2 subsystem was holding at the time of a subsystem failure. The lock is retained in the coupling facility lock structure across a DB2 failure.

RID. Record identifier.

RID pool. Record identifier pool.

right outer join. The result of a join operation that includes the matched rows of both tables that are being joined and preserves the unmatched rows of the second join operand. See also *join*.

RLF. Resource limit facility.

RLST. Resource limit specification table.

RMID. Resource manager identifier.

RO. Read-only access.

rollback. The process of restoring data that was changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

root page. The index page that is at the highest level (or the beginning point) in an index.

routine. A term that refers to either a user-defined function or a stored procedure.

row. The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

ROWID. Row identifier.

row identifier (ROWID). A value that uniquely identifies a row. This value is stored with the row and never changes.

row lock. A lock on a single row of data.

| **rowset.** A set of rows for which a cursor position is
| established.

| **rowset cursor.** A cursor that is defined so that one or
| more rows can be returned as a rowset for a single
| FETCH statement, and the cursor is positioned on the
| set of rows that is fetched.

| **rowset-positioned access.** The ability to retrieve
| multiple rows from a single FETCH statement.

| **row-positioned access.** The ability to retrieve a single
| row from a single FETCH statement.

row trigger. A trigger that is defined with the trigger granularity FOR EACH ROW.

RRE. Residual recovery entry (in IMS).

RRSAF. Recoverable Resource Manager Services attachment facility.

RS. Read stability.

RTT. Resource translation table.

RURE. Restart URE.

S

savepoint. A named entity that 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 restoration of data and schemas to a savepoint is usually referred to as *rolling back to a savepoint*.

SBCS. Single-byte character set.

SCA. Shared communications area.

scalar function. An SQL operation that produces a
single value from another value and is expressed as a
function name, followed by a list of arguments that are
enclosed in parentheses. Contrast with *aggregate*
function.

scale. In SQL, the number of digits to the right of the decimal point (called the *precision* in the C language). The DB2 library uses the SQL definition.

schema. (1) The organization or structure of a database. (2) A logical grouping for user-defined functions, distinct types, triggers, and stored procedures. When an object of one of these types is created, it is assigned to one schema, which is determined by the name of the object. For example, the following statement creates a distinct type T in schema C:

```
CREATE DISTINCT TYPE C.T ...
```

scrollability. The ability to use a cursor to fetch in either a forward or backward direction. The FETCH statement supports multiple fetch orientations to indicate the new position of the cursor. See also *fetch orientation*.

scrollable cursor. A cursor that can be moved in both a forward and a backward direction.

SDWA. System diagnostic work area.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

secondary authorization ID. An authorization ID that has been associated with a primary authorization ID by an authorization exit routine.

secondary group buffer pool. For a duplexed group buffer pool, the structure that is used to back up changed pages that are written to the primary group buffer pool. No page registration or cross-invalidation occurs using the secondary group buffer pool. The z/OS equivalent is *new* structure.

secondary index. A nonpartitioning index on a partitioned table.

section. The segment of a plan or package that contains the executable structures for a single SQL statement. For most SQL statements, one section in the plan exists for each SQL statement in the source program. However, for cursor-related statements, the DECLARE, OPEN, FETCH, and CLOSE statements reference the same section because they each refer to the SELECT statement that is named in the DECLARE CURSOR statement. SQL statements such as COMMIT, ROLLBACK, and some SET statements do not use a section.

segment. A group of pages that holds rows of a single table. See also *segmented table space*.

segmented table space. A table space that is divided into equal-sized groups of pages called segments. Segments are assigned to tables so that rows of different tables are never stored in the same segment.

self-referencing constraint. A referential constraint that defines a relationship in which a table is a dependent of itself.

self-referencing table. A table with a self-referencing constraint.

sensitive cursor. A cursor that is sensitive to changes that are made to the database after the result table has been materialized.

sequence. A user-defined object that generates a sequence of numeric values according to user specifications.

sequential data set. A non-DB2 data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. Several of the DB2 database utilities require sequential data sets.

sequential prefetch. A mechanism that triggers consecutive asynchronous I/O operations. Pages are fetched before they are required, and several pages are read with a single I/O operation.

serial cursor. A cursor that can be moved only in a forward direction.

serialized profile. A Java object that contains SQL statements and descriptions of host variables. The SQLJ translator produces a serialized profile for each connection context.

server. The target of a request from a remote requester. In the DB2 environment, the server function is provided by the distributed data facility, which is used to access DB2 data from remote applications.

server-side programming. A method for adding DB2 data into dynamic Web pages.

service class. An eight-character identifier that is used by the z/OS Workload Manager to associate user performance goals with a particular DDF thread or stored procedure. A service class is also used to classify work on parallelism assistants.

service request block. A unit of work that is scheduled to execute in another address space.

session. A link between two nodes in a VTAM network.

session protocols. The available set of SNA communication requests and responses.

shared communications area (SCA). A coupling facility list structure that a DB2 data sharing group uses for inter-DB2 communication.

share lock. A lock that prevents concurrently executing application processes from changing data, but not from reading data. Contrast with *exclusive lock*.

shift-in character. A special control character (X'0F') that is used in EBCDIC systems to denote that the subsequent bytes represent SBCS characters. See also *shift-out character*.

shift-out character. A special control character (X'0E') that is used in EBCDIC systems to denote that the subsequent bytes, up to the next shift-in control character, represent DBCS characters. See also *shift-in character*.

sign-on. A request that is made on behalf of an individual CICS or IMS application process by an attachment facility to enable DB2 to verify that it is authorized to use DB2 resources.

simple page set. A nonpartitioned page set. A simple page set initially consists of a single data set (page set piece). If and when that data set is extended to 2 GB, another data set is created, and so on, up to a total of 32 data sets. DB2 considers the data sets to be a single contiguous linear address space containing a maximum of 64 GB. Data is stored in the next available location within this address space without regard to any partitioning scheme.

simple table space. A table space that is neither partitioned nor segmented.

single-byte character set (SBCS). A set of characters in which each character is represented by a single byte. Contrast with *double-byte character set* or *multibyte character set*.

single-precision floating point number. A 32-bit approximate representation of a real number.

size. In the C language, the total number of digits in a decimal number (called the *precision* in SQL). The DB2 library uses the SQL term.

SMF. System Management Facilities.

SMP/E. System Modification Program/Extended.

SMS. Storage Management Subsystem.

SNA. Systems Network Architecture.

SNA network. The part of a network that conforms to the formats and protocols of Systems Network Architecture (SNA).

socket. A callable TCP/IP programming interface that TCP/IP network applications use to communicate with remote TCP/IP partners.

sourced function. A function that is implemented by another built-in or user-defined function that is already known to the database manager. This function can be a scalar function or a column (aggregating) function; it returns a single value from a set of values (for example, MAX or AVG). Contrast with *built-in function, external function,* and *SQL function*.

source program. A set of host language statements and SQL statements that is processed by an SQL precompiler.

source table. A table that can be a base table, a view, a table expression, or a user-defined table function.

source type. An existing type that DB2 uses to internally represent a distinct type.

space. A sequence of one or more blank characters.

special register. A storage area that DB2 defines for an application process to use for storing information that can be referenced in SQL statements. Examples of special registers are USER and CURRENT DATE.

specific function name. A particular user-defined function that is known to the database manager by its specific name. Many specific user-defined functions can have the same function name. When a user-defined function is defined to the database, every function is assigned a specific name that is unique within its schema. Either the user can provide this name, or a default name is used.

SPUFI. SQL Processor Using File Input.

SQL. Structured Query Language.

SQL authorization ID (SQL ID). The authorization ID that is used for checking dynamic SQL statements in some situations.

SQLCA. SQL communication area.

SQL communication area (SQLCA). A structure that is used to provide an application program with information about the execution of its SQL statements.

SQL connection. An association between an application process and a local or remote application server or database server.

SQLDA. SQL descriptor area.

SQL descriptor area (SQLDA). A structure that describes input variables, output variables, or the columns of a result table.

SQL escape character. The symbol that is used to enclose an SQL delimited identifier. This symbol is the double quotation mark ("). See also *escape character*.

SQL function. A user-defined function in which the CREATE FUNCTION statement contains the source code. The source code is a single SQL expression that evaluates to a single value. The SQL user-defined function can return only one parameter.

SQL ID. SQL authorization ID.

SQLJ. Structured Query Language (SQL) that is embedded in the Java programming language.

SQL path. An ordered list of schema names that are used in the resolution of unqualified references to user-defined functions, distinct types, and stored procedures. In dynamic SQL, the current path is found in the CURRENT PATH special register. In static SQL, it is defined in the PATH bind option.

SQL procedure. A user-written program that can be invoked with the SQL CALL statement. Contrast with *external procedure*.

SQL processing conversation. Any conversation that requires access of DB2 data, either through an application or by dynamic query requests.

SQL Processor Using File Input (SPUFI). A facility of the TSO attachment subcomponent that enables the DB2I user to execute SQL statements without embedding them in an application program.

SQL return code. Either SQLCODE or SQLSTATE.

SQL routine. A user-defined function or stored procedure that is based on code that is written in SQL.

SQL statement coprocessor. An alternative to the DB2 precompiler that lets the user process SQL statements at compile time. The user invokes an SQL statement coprocessor by specifying a compiler option.

SQL string delimiter. A symbol that is used to enclose an SQL string constant. The SQL string delimiter is the apostrophe ('), except in COBOL applications, where the user assigns the symbol, which is either an apostrophe or a double quotation mark (").

SRB. Service request block.

SSI. Subsystem interface (in z/OS).

SSM. Subsystem member (in IMS).

stand-alone. An attribute of a program that means that it is capable of executing separately from DB2, without using DB2 services.

star join. A method of joining a dimension column of a fact table to the key column of the corresponding dimension table. See also *join*, *dimension*, and *star schema*.

star schema. The combination of a fact table (which contains most of the data) and a number of dimension tables. See also *star join*, *dimension*, and *dimension table*.

statement handle. In DB2 ODBC, the data object that contains information about an SQL statement that is managed by DB2 ODBC. This includes information such as dynamic arguments, bindings for dynamic arguments and columns, cursor information, result values, and status information. Each statement handle is associated with the connection handle.

statement string. For a dynamic SQL statement, the character string form of the statement.

statement trigger. A trigger that is defined with the trigger granularity FOR EACH STATEMENT.

static cursor. A named control structure that does not change the size of the result table or the order of its rows after an application opens the cursor. Contrast with *dynamic cursor.*

static SQL. SQL statements, embedded within a program, that are prepared during the program preparation process (before the program is executed). After being prepared, the SQL statement does not change (although values of host variables that are specified by the statement might change).

storage group. A named set of disks on which DB2 data can be stored.

stored procedure. A user-written application program that can be invoked through the use of the SQL CALL statement.

string. See *character string* or *graphic string*.

strong typing. A process that guarantees that only user-defined functions and operations that are defined on a distinct type can be applied to that type. For example, you cannot directly compare two currency types, such as Canadian dollars and U.S. dollars. But you can provide a user-defined function to convert one currency to the other and then do the comparison.

structure. (1) A name that refers collectively to different types of DB2 objects, such as tables, databases, views, indexes, and table spaces. (2) A construct that uses z/OS to map and manage storage on a coupling facility. See also *cache structure*, *list structure*, or *lock structure*.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

structure owner. In relation to group buffer pools, the DB2 member that is responsible for the following activities:

- Coordinating rebuild, checkpoint, and damage assessment processing
- Monitoring the group buffer pool threshold and notifying castout owners when the threshold has been reached

subcomponent. A group of closely related DB2 modules that work together to provide a general function.

subject table. The table for which a trigger is created. When the defined triggering event occurs on this table, the trigger is activated.

subpage. The unit into which a physical index page can be divided.

subquery. A SELECT statement within the WHERE or HAVING clause of another SQL statement; a nested SQL statement.

subselect. That form of a query that does not include an ORDER BY clause, an UPDATE clause, or UNION operators.

substitution character. A unique character that is substituted during character conversion for any characters in the source program that do not have a match in the target coding representation.

subsystem. A distinct instance of a relational database management system (RDBMS).

surrogate pair. A coded representation for a single character that consists of a sequence of two 16-bit code units, in which the first value of the pair is a high-surrogate code unit in the range U+D800 through U+DBFF, and the second value is a low-surrogate code unit in the range U+DC00 through U+DFFF. Surrogate pairs provide an extension mechanism for encoding 917 476 characters without requiring the use of 32-bit characters.

SVC dump. A dump that is issued when a z/OS or a DB2 functional recovery routine detects an error.

sync point. See *commit point*.

syncpoint tree. The tree of recovery managers and resource managers that are involved in a logical unit of work, starting with the recovery manager, that make the final commit decision.

synonym. In SQL, an alternative name for a table or view. Synonyms can be used to refer only to objects at the subsystem in which the synonym is defined.

syntactic character set. A set of 81 graphic characters that are registered in the IBM registry as character set 00640. This set was originally recommended to the programming language community to be used for syntactic purposes toward maximizing portability and interchangeability across systems and country boundaries. It is contained in most of the primary registered character sets, with a few exceptions. See also *invariant character set*.

Sysplex. See *Parallel Sysplex*.

Sysplex query parallelism. Parallel execution of a single query that is accomplished by using multiple tasks on more than one DB2 subsystem. See also *query CP parallelism*.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

system agent. A work request that DB2 creates internally such as prefetch processing, deferred writes, and service tasks.

system conversation. The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

system diagnostic work area (SDWA). The data that is recorded in a SYS1.LOGREC entry that describes a program or hardware error.

system-directed connection. A connection that a relational DBMS manages by processing SQL statements with three-part names.

System Modification Program/Extended (SMP/E). A z/OS tool for making software changes in programming systems (such as DB2) and for controlling those changes.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information through and controlling the configuration and operation of networks.

SYS1.DUMPxx data set. A data set that contains a system dump (in z/OS).

SYS1.LOGREC. A service aid that contains important information about program and hardware errors (in z/OS).

T

table. A named data object consisting of a specific number of columns and some number of unordered rows. See also *base table* or *temporary table*.

table-controlled partitioning. A type of partitioning in which partition boundaries for a partitioned table are controlled by values that are defined in the CREATE TABLE statement. Partition limits are saved in the LIMITKEY_INTERNAL column of the SYSIBM.SYSTABLEPART catalog table.

table function. A function that receives a set of arguments and returns a table to the SQL statement that references the function. A table function can be referenced only in the FROM clause of a subselect.

table locator. A mechanism that allows access to trigger transition tables in the FROM clause of SELECT statements, in the subselect of INSERT statements, or from within user-defined functions. A table locator is a fullword integer value that represents a transition table.

table space. A page set that is used to store the records in one or more tables.

table space set. A set of table spaces and partitions that should be recovered together for one of these reasons:
- Each of them contains a table that is a parent or descendent of a table in one of the others.
- The set contains a base table and associated auxiliary tables.

A table space set can contain both types of relationships.

task control block (TCB). A z/OS control block that is used to communicate information about tasks within an address space that are connected to DB2. See also *address space connection*.

TB. Terabyte (1 099 511 627 776 bytes).

TCB. Task control block (in z/OS).

TCP/IP. A network communication protocol that computer systems use to exchange information across telecommunication links.

TCP/IP port. A 2-byte value that identifies an end user or a TCP/IP network application within a TCP/IP host.

template. A DB2 utilities output data set descriptor that is used for dynamic allocation. A template is defined by the TEMPLATE utility control statement.

temporary table. A table that holds temporary data. Temporary tables are useful for holding or sorting intermediate results from queries that contain a large number of rows. The two types of temporary table, which are created by different SQL statements, are the created temporary table and the declared temporary table. Contrast with *result table*. See also *created temporary table* and *declared temporary table*.

Terminal Monitor Program (TMP). A program that provides an interface between terminal users and command processors and has access to many system services (in z/OS).

thread. The DB2 structure that describes an application's connection, traces its progress, processes resource functions, and delimits its accessibility to DB2 resources and services. Most DB2 functions execute under a thread structure. See also *allied thread* and *database access thread*.

threadsafe. A characteristic of code that allows multithreading both by providing private storage areas for each thread, and by properly serializing shared (global) storage areas.

three-part name. The full name of a table, view, or alias. It consists of a location name, authorization ID, and an object name, separated by a period.

time. A three-part value that designates a time of day in hours, minutes, and seconds.

time duration. A decimal integer that represents a number of hours, minutes, and seconds.

timeout. Abnormal termination of either the DB2 subsystem or of an application because of the unavailability of resources. Installation specifications are set to determine both the amount of time DB2 is to wait for IRLM services after starting, and the amount of time IRLM is to wait if a resource that an application requests is unavailable. If either of these time specifications is exceeded, a timeout is declared.

Time-Sharing Option (TSO). An option in MVS that provides interactive time sharing from remote terminals.

timestamp. A seven-part value that consists of a date and time. The timestamp is expressed in years, months, days, hours, minutes, seconds, and microseconds.

TMP. Terminal Monitor Program.

to-do. A state of a unit of recovery that indicates that the unit of recovery's changes to recoverable DB2 resources are indoubt and must either be applied to the disk media or backed out, as determined by the commit coordinator.

trace. A DB2 facility that provides the ability to monitor and collect DB2 monitoring, auditing, performance, accounting, statistics, and serviceability (global) data.

transaction lock. A lock that is used to control concurrent execution of SQL statements.

transaction program name. In SNA LU 6.2 conversations, the name of the program at the remote logical unit that is to be the other half of the conversation.

transient XML data type. A data type for XML values that exists only during query processing.

transition table. A temporary table that contains all the affected rows of the subject table in their state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the table of changed rows in the old state or the new state.

transition variable. A variable that contains a column value of the affected row of the subject table in its state before or after the triggering event occurs. Triggered SQL statements in the trigger definition can reference the set of old values or the set of new values.

tree structure. A data structure that represents entities in nodes, with a most one parent node for each node, and with only one root node.

trigger. A set of SQL statements that are stored in a DB2 database and executed when a certain event occurs in a DB2 table.

trigger activation. The process that occurs when the trigger event that is defined in a trigger definition is executed. Trigger activation consists of the evaluation of the triggered action condition and conditional execution of the triggered SQL statements.

trigger activation time. An indication in the trigger definition of whether the trigger should be activated before or after the triggered event.

trigger body. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. A trigger body is also called *triggered SQL statements*.

trigger cascading. The process that occurs when the triggered action of a trigger causes the activation of another trigger.

triggered action. The SQL logic that is performed when a trigger is activated. The triggered action consists of an optional triggered action condition and a set of triggered SQL statements that are executed only if the condition evaluates to true.

triggered action condition. An optional part of the triggered action. This Boolean condition appears as a WHEN clause and specifies a condition that DB2 evaluates to determine if the triggered SQL statements should be executed.

triggered SQL statements. The set of SQL statements that is executed when a trigger is activated and its triggered action condition evaluates to true. Triggered SQL statements are also called the *trigger body*.

trigger granularity. A characteristic of a trigger, which determines whether the trigger is activated:
- Only once for the triggering SQL statement
- Once for each row that the SQL statement modifies

triggering event. The specified operation in a trigger definition that causes the activation of that trigger. The triggering event is comprised of a triggering operation (INSERT, UPDATE, or DELETE) and a subject table on which the operation is performed.

triggering SQL operation. The SQL operation that causes a trigger to be activated when performed on the subject table.

trigger package. A package that is created when a CREATE TRIGGER statement is executed. The package is executed when the trigger is activated.

TSO. Time-Sharing Option.

TSO attachment facility. A DB2 facility consisting of the DSN command processor and DB2I. Applications that are not written for the CICS or IMS environments can run under the TSO attachment facility.

typed parameter marker. A parameter marker that is specified along with its target data type. It has the general form:

```
CAST(? AS data-type)
```

type 1 indexes. Indexes that were created by a release of DB2 before DB2 Version 4 or that are specified as type 1 indexes in Version 4. Contrast with *type 2 indexes*. As of Version 8, type 1 indexes are no longer supported.

type 2 indexes. Indexes that are created on a release of DB2 after Version 7 or that are specified as type 2 indexes in Version 4 or later.

U

UCS-2. Universal Character Set, coded in 2 octets, which means that characters are represented in 16-bits per character.

UDF. User-defined function.

UDT. User-defined data type. In DB2 UDB for z/OS, the term *distinct type* is used instead of user-defined data type. See *distinct type*.

uncommitted read (UR). The isolation level that allows an application to read uncommitted data.

underlying view. The view on which another view is directly or indirectly defined.

undo. A state of a unit of recovery that indicates that the changes that the unit of recovery made to recoverable DB2 resources must be backed out.

Unicode. A standard that parallels the ISO-10646 standard. Several implementations of the Unicode standard exist, all of which have the ability to represent a large percentage of the characters that are contained in the many scripts that are used throughout the world.

uniform resource locator (URL). A Web address, which offers a way of naming and locating specific items on the Web.

union. An SQL operation that combines the results of two SELECT statements. Unions are often used to merge lists of values that are obtained from several tables.

unique constraint. An SQL rule that no two values in a primary key, or in the key of a unique index, can be the same.

unique index. An index that ensures that no identical key values are stored in a column or a set of columns in a table.

unit of recovery. A recoverable sequence of operations within a single resource manager, such as an instance of DB2. Contrast with *unit of work*.

unit of recovery identifier (URID). The LOGRBA of the first log record for a unit of recovery. The URID also appears in all subsequent log records for that unit of recovery.

unit of work. A recoverable sequence of operations within an application process. At any time, an application process is a single unit of work, but the life of an application process can involve many units of work as a result of commit or rollback operations. In a *multisite update* operation, a single unit of work can include several *units of recovery*. Contrast with *unit of recovery*.

Universal Unique Identifier (UUID). An identifier that is immutable and unique across time and space (in z/OS).

unlock. The act of releasing an object or system resource that was previously locked and returning it to general availability within DB2.

untyped parameter marker. A parameter marker that is specified without its target data type. It has the form of a single question mark (?).

updatability. The ability of a cursor to perform positioned updates and deletes. The updatability of a cursor can be influenced by the SELECT statement and the cursor sensitivity option that is specified on the DECLARE CURSOR statement.

update hole. The location on which a cursor is positioned when a row in a result table is fetched again and the new values no longer satisfy the search condition. DB2 marks a row in the result table as an update hole when an update to the corresponding row in the database causes that row to no longer qualify for the result table.

update trigger. A trigger that is defined with the triggering SQL operation UPDATE.

upstream. The node in the syncpoint tree that is responsible, in addition to other recovery or resource managers, for coordinating the execution of a two-phase commit.

UR. Uncommitted read.

URE. Unit of recovery element.

URID . Unit of recovery identifier.

URL. Uniform resource locator.

user-defined data type (UDT). See *distinct type*.

user-defined function (UDF). A function that is defined to DB2 by using the CREATE FUNCTION statement and that can be referenced thereafter in SQL statements. A user-defined function can be an *external function,* a *sourced function,* or an *SQL function.* Contrast with *built-in function*.

user view. In logical data modeling, a model or representation of critical information that the business requires.

UTF-8. Unicode Transformation Format, 8-bit encoding form, which is designed for ease of use with existing ASCII-based systems. The CCSID value for data in UTF-8 format is 1208. DB2 UDB for z/OS supports UTF-8 in mixed data fields.

UTF-16. Unicode Transformation Format, 16-bit encoding form, which is designed to provide code values for over a million characters and a superset of UCS-2. The CCSID value for data in UTF-16 format is 1200. DB2 UDB for z/OS supports UTF-16 in graphic data fields.

UUID. Universal Unique Identifier.

V

value. The smallest unit of data that is manipulated in SQL.

variable. A data element that specifies a value that can be changed. A COBOL elementary data item is an example of a variable. Contrast with *constant*.

variant function. See *nondeterministic function*.

varying-length string. A character or graphic string whose length varies within set limits. Contrast with *fixed-length string*.

version. A member of a set of similar programs, DBRMs, packages, or LOBs.

A version of a program is the source code that is produced by precompiling the program. The program version is identified by the program name and a timestamp (consistency token).

A version of a DBRM is the DBRM that is produced by precompiling a program. The DBRM version is identified by the same program name and timestamp as a corresponding program version.

A version of a package is the result of binding a DBRM within a particular database system. The package version is identified by the same program name and consistency token as the DBRM.

A version of a LOB is a copy of a LOB value at a point in time. The version number for a LOB is stored in the auxiliary index entry for the LOB.

view. An alternative representation of data from one or more tables. A view can include all or some of the columns that are contained in tables on which it is defined.

view check option. An option that specifies whether every row that is inserted or updated through a view must conform to the definition of that view. A view check option can be specified with the WITH CASCADED CHECK OPTION, WITH CHECK OPTION, or WITH LOCAL CHECK OPTION clauses of the CREATE VIEW statement.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed- and varying-length records on disk devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number (in z/OS).

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network (in z/OS).

| **volatile table.** A table for which SQL operations
| choose index access whenever possible.

VSAM. Virtual Storage Access Method.

VTAM. Virtual Telecommunication Access Method (in z/OS).

W

warm start. The normal DB2 restart process, which involves reading and processing log records so that data that is under the control of DB2 is consistent. Contrast with *cold start*.

WLM application environment. A z/OS Workload Manager attribute that is associated with one or more stored procedures. The WLM application environment determines the address space in which a given DB2 stored procedure runs.

write to operator (WTO). An optional user-coded service that allows a message to be written to the system console operator informing the operator of errors and unusual system conditions that might need to be corrected (in z/OS).

WTO. Write to operator.

WTOR. Write to operator (WTO) with reply.

X

XCF. See *cross-system coupling facility*.

XES. See *cross-system extended services*.

| **XML.** See *Extensible Markup Language*.

| **XML attribute.** A name-value pair within a tagged
| XML element that modifies certain features of the
| element.

XML element. A logical structure in an XML
document that is delimited by a start and an end tag.
Anything between the start tag and the end tag is the
content of the element.

| **XML node.** The smallest unit of valid, complete
| structure in a document. For example, a node can
| represent an element, an attribute, or a text string.

| **XML publishing functions.** Functions that return
| XML values from SQL values.

X/Open. An independent, worldwide open systems organization that is supported by most of the world's largest information systems suppliers, user organizations, and software companies. X/Open's goal is to increase the portability of applications by combining existing and emerging standards.

XRF. Extended recovery facility.

Z

| **z/OS.** An operating system for the eServer™ product
| line that supports 64-bit real and virtual storage.

z/OS Distributed Computing Environment (z/OS DCE). A set of technologies that are provided by the Open Software Foundation to implement distributed computing.

Bibliography

DB2 Universal Database for z/OS Version 8 product information:

- *DB2 Administration Guide*, SC18-7413
- *DB2 Application Programming and SQL Guide*, SC18-7415
- *DB2 Application Programming Guide and Reference for Java*, SC18-7414
- *DB2 Codes*, GC18-9603
- *DB2 Command Reference*, SC18-7416
- *DB2 Common Criteria Guide*, SC18-9672
- *DB2 Data Sharing: Planning and Administration*, SC18-7417
- *DB2 Diagnosis Guide and Reference*, LY37-3201
- *DB2 Diagnostic Quick Reference Card*, LY37-3202
- *DB2 Image, Audio, and Video Extenders Administration and Programming*, SC18-7429
- *DB2 Installation Guide*, GC18-7418
- *DB2 Licensed Program Specifications*, GC18-7420
- *DB2 Management Clients Package Program Directory*, GI10-8567
- *DB2 Messages*, GC18-9602
- *DB2 ODBC Guide and Reference*, SC18-7423
- *The Official Introduction to DB2 UDB for z/OS*
- *DB2 Program Directory*, GI10-8566
- *DB2 RACF Access Control Module Guide*, SC18-7433
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC18-7424
- *DB2 Reference Summary*, SX26-3853
- *DB2 Release Planning Guide*, SC18-7425
- *DB2 SQL Reference*, SC18-7426
- *DB2 Text Extender Administration and Programming*, SC18-7430
- *DB2 Utility Guide and Reference*, SC18-7427
- *DB2 What's New?*, GC18-7428
- *DB2 XML Extender for z/OS Administration and Programming*, SC18-7431

Books and resources about related products:

APL2®

- *APL2 Programming Guide*, SH21-1072
- *APL2 Programming: Language Reference*, SH21-1061
- *APL2 Programming: Using Structured Query Language (SQL)*, SH21-1057

BookManager® READ/MVS

- *BookManager READ/MVS V1R3: Installation Planning & Customization*, SC38-2035

C language: IBM C/C++ for z/OS

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821

Character Data Representation Architecture

- *Character Data Representation Architecture Overview*, GC09-2207
- *Character Data Representation Architecture Reference and Registry*, SC09-2190

CICS Transaction Server for z/OS

The publication order numbers below are for Version 2 Release 2 and Version 2 Release 3 (with the release 2 number listed first).

- *CICS Transaction Server for z/OS Information Center*, SK3T-6903 or SK3T-6957.
- *CICS Transaction Server for z/OS Application Programming Guide*, SC34-5993 or SC34-6231
- *CICS Transaction Server for z/OS Application Programming Reference*, SC34-5994 or SC34-6232
- *CICS Transaction Server for z/OS CICS-RACF Security Guide*, SC34-6011 or SC34-6249
- *CICS Transaction Server for z/OS CICS Supplied Transactions*, SC34-5992 or SC34-6230
- *CICS Transaction Server for z/OS Customization Guide*, SC34-5989 or SC34-6227
- *CICS Transaction Server for z/OS Data Areas*, LY33-6100 or LY33-6103
- *CICS Transaction Server for z/OS DB2 Guide*, SC34-6014 or SC34-6252
- *CICS Transaction Server for z/OS External Interfaces Guide*, SC34-6006 or SC34-6244
- *CICS Transaction Server for z/OS Installation Guide*, GC34-5985 or GC34-6224
- *CICS Transaction Server for z/OS Intercommunication Guide*, SC34-6005 or SC34-6243
- *CICS Transaction Server for z/OS Messages and Codes*, GC34-6003 or GC34-6241
- *CICS Transaction Server for z/OS Operations and Utilities Guide*, SC34-5991 or SC34-6229

- *CICS Transaction Server for z/OS Performance Guide*, SC34-6009 or SC34-6247
- *CICS Transaction Server for z/OS Problem Determination Guide*, SC34-6002 or SC34-6239
- *CICS Transaction Server for z/OS Release Guide*, GC34-5983 or GC34-6218
- *CICS Transaction Server for z/OS Resource Definition Guide*, SC34-5990 or SC34-6228
- *CICS Transaction Server for z/OS System Definition Guide*, SC34-5988 or SC34–6226
- *CICS Transaction Server for z/OS System Programming Reference*, SC34-5595 or SC34–6233

CICS Transaction Server for OS/390
- *CICS Transaction Server for OS/390 Application Programming Guide*, SC33-1687
- *CICS Transaction Server for OS/390 DB2 Guide*, SC33-1939
- *CICS Transaction Server for OS/390 External Interfaces Guide*, SC33-1944
- *CICS Transaction Server for OS/390 Resource Definition Guide*, SC33-1684

COBOL:
- *IBM COBOL Language Reference*, SC27-1408
- *Enterprise COBOL for z/OS Programming Guide*, SC27-1412

Database Design
- *DB2 for z/OS and OS/390 Development for Performance Volume I* by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-605-2
- *DB2 for z/OS and OS/390 Development for Performance Volume II* by Gabrielle Wiorkowski, Gabrielle & Associates, ISBN 0-96684-606-0
- *Handbook of Relational Database Design* by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8

DB2 Administration Tool
- *DB2 Administration Tool for z/OS User's Guide and Reference*, available on the Web at www.ibm.com/software/data/db2imstools/library.html

DB2 Buffer Pool Analyzer for z/OS
- *DB2 Buffer Pool Tool for z/OS User's Guide and Reference*, available on the Web at www.ibm.com/software/data/db2imstools/library.html

DB2 Connect™
- *IBM DB2 Connect Quick Beginnings for DB2 Connect Enterprise Edition*, GC09-4833
- *IBM DB2 Connect Quick Beginnings for DB2 Connect Personal Edition*, GC09-4834
- *IBM DB2 Connect User's Guide*, SC09-4835

DB2 DataPropagator™
- *DB2 Universal Database Replication Guide and Reference*, SC27-1121

DB2 Performance Expert for z/OS, Version 1

The following books are part of the DB2 Performance Expert library. Some of these books include information about the following tools: IBM DB2 Performance Expert for z/OS; IBM DB2 Performance Monitor for z/OS; and DB2 Buffer Pool Analyzer for z/OS.
- *OMEGAMON Buffer Pool Analyzer User's Guide*, SC18-7972
- *OMEGAMON Configuration and Customization*, SC18-7973
- *OMEGAMON Messages*, SC18-7974
- *OMEGAMON Monitoring Performance from ISPF*, SC18-7975
- *OMEGAMON Monitoring Performance from Performance Expert Client*, SC18-7976
- *OMEGAMON Program Directory*, GI10-8549
- *OMEGAMON Report Command Reference*, SC18-7977
- *OMEGAMON Report Reference*, SC18-7978
- *Using IBM Tivoli OMEGAMON XE on z/OS*, SC18-7979

DB2 Query Management Facility (QMF) Version 8.1
- *DB2 Query Management Facility: DB2 QMF High Performance Option User's Guide for TSO/CICS*, SC18-7450
- *DB2 Query Management Facility: DB2 QMF Messages and Codes*, GC18-7447
- *DB2 Query Management Facility: DB2 QMF Reference*, SC18-7446
- *DB2 Query Management Facility: Developing DB2 QMF Applications*, SC18-7651
- *DB2 Query Management Facility: Getting Started with DB2 QMF for Windows and DB2 QMF for WebSphere*, SC18-7449
- *DB2 Query Management Facility: Getting Started with DB2 QMF Query Miner*, GC18-7451
- *DB2 Query Management Facility: Installing and Managing DB2 QMF for TSO/CICS*, GC18-7444
- *DB2 Query Management Facility: Installing and Managing DB2 QMF for Windows and DB2 QMF for WebSphere*, GC18-7448

- *DB2 Query Management Facility: Introducing DB2 QMF*, GC18-7443
- *DB2 Query Management Facility: Using DB2 QMF*, SC18-7445
- *DB2 Query Management Facility: DB2 QMF Visionary Developer's Guide*, SC18-9093
- *DB2 Query Management Facility: DB2 QMF Visionary Getting Started Guide*, GC18-9092

DB2 Redbooks™

For access to all IBM Redbooks about DB2, see the IBM Redbooks Web page at www.ibm.com/redbooks

DB2 Server for VSE & VM
- *DB2 Server for VM: DBS Utility*, SC09-2983

DB2 Universal Database Cross-Platform information
- *IBM DB2 Universal Database SQL Reference for Cross-Platform Development*, available at www.ibm.com/software/data/developer/cpsqlref/

DB2 Universal Database for iSeries

The following books are available at www.ibm.com/iseries/infocenter
- *DB2 Universal Database for iSeries Performance and Query Optimization*
- *DB2 Universal Database for iSeries Database Programming*
- *DB2 Universal Database for iSeries SQL Programming Concepts*
- *DB2 Universal Database for iSeries SQL Programming with Host Languages*
- *DB2 Universal Database for iSeries SQL Reference*
- *DB2 Universal Database for iSeries Distributed Data Management*
- *DB2 Universal Database for iSeries Distributed Database Programming*

DB2 Universal Database for Linux, UNIX, and Windows:
- *DB2 Universal Database Administration Guide: Planning*, SC09-4822
- *DB2 Universal Database Administration Guide: Implementation*, SC09-4820
- *DB2 Universal Database Administration Guide: Performance*, SC09-4821
- *DB2 Universal Database Administrative API Reference*, SC09-4824
- *DB2 Universal Database Application Development Guide: Building and Running Applications*, SC09-4825
- *DB2 Universal Database Call Level Interface Guide and Reference, Volumes 1 and 2*, SC09-4849 and SC09-4850
- *DB2 Universal Database Command Reference*, SC09-4828
- *DB2 Universal Database SQL Reference Volume 1*, SC09-4844
- *DB2 Universal Database SQL Reference Volume 2*, SC09-4845

Device Support Facilities
- *Device Support Facilities User's Guide and Reference*, GC35-0033

DFSMS

These books provide information about a variety of components of DFSMS, including z/OS DFSMS, z/OS DFSMSdfp™, z/OS DFSMSdss, z/OS DFSMShsm, and z/OS DFP.
- *z/OS DFSMS Access Method Services for Catalogs*, SC26-7394
- *z/OS DFSMSdss Storage Administration Guide*, SC35-0423
- *z/OS DFSMSdss Storage Administration Reference*, SC35-0424
- *z/OS DFSMShsm Managing Your Own Data*, SC35-0420
- *z/OS DFSMSdfp: Using DFSMSdfp in the z/OS Environment*, SC26-7473
- *z/OS DFSMSdfp Diagnosis Reference*, GY27-7618
- *z/OS DFSMS: Implementing System-Managed Storage*, SC27-7407
- *z/OS DFSMS: Macro Instructions for Data Sets*, SC26-7408
- *z/OS DFSMS: Managing Catalogs*, SC26-7409
- *z/OS MVS: Program Management User's Guide and Reference*, SA22-7643
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
- *z/OS DFSMSdfp Storage Administration Reference*, SC26-7402
- *z/OS DFSMS: Using Data Sets*, SC26-7410
- *DFSMS/MVS: Using Advanced Services* , SC26-7400
- *DFSMS/MVS: Utilities*, SC26-7414

DFSORT™
- *DFSORT Application Programming: Guide*, SC33-4035
- *DFSORT Installation and Customization*, SC33-4034

Distributed Relational Database Architecture

- *Open Group Technical Standard*; the Open Group presently makes the following DRDA books available through its Web site at www.opengroup.org
  - *Open Group Technical Standard, DRDA Version 3 Vol. 1: Distributed Relational Database Architecture*
  - *Open Group Technical Standard, DRDA Version 3 Vol. 2: Formatted Data Object Content Architecture*
  - *Open Group Technical Standard, DRDA Version 3 Vol. 3: Distributed Data Management Architecture*

Domain Name System
- *DNS and BIND, Third Edition, Paul Albitz and Cricket Liu, O'Reilly*, ISBN 0-59600-158-4

Education
- Information about IBM educational offerings is available on the Web at http://www.ibm.com/software/sw-training/
- A collection of glossaries of IBM terms is available on the IBM Terminology Web site at www.ibm.com/ibm/terminology/index.html

eServer zSeries®
- *IBM eServer zSeries Processor Resource/System Manager Planning Guide*, SB10-7033

Fortran: VS Fortran
- *VS Fortran Version 2: Language and Library Reference*, SC26-4221
- *VS Fortran Version 2: Programming Guide for CMS and MVS*, SC26-4222

High Level Assembler
- *High Level Assembler for MVS and VM and VSE Language Reference*, SC26-4940
- *High Level Assembler for MVS and VM and VSE Programmer's Guide*, SC26-4941

ICSF
- *z/OS ICSF Overview*, SA22-7519
- *Integrated Cryptographic Service Facility Administrator's Guide*, SA22-7521

IMS Version 8

IMS product information is available on the IMS Library Web page, which you can find at www.ibm.com/ims
- *IMS Administration Guide: System*, SC27-1284
- *IMS Administration Guide: Transaction Manager*, SC27-1285
- *IMS Application Programming: Database Manager*, SC27-1286
- *IMS Application Programming: Design Guide*, SC27-1287
- *IMS Application Programming: Transaction Manager*, SC27-1289
- *IMS Command Reference*, SC27-1291
- *IMS Customization Guide*, SC27-1294
- *IMS Install Volume 1: Installation Verification*, GC27-1297
- *IMS Install Volume 2: System Definition and Tailoring*, GC27-1298
- *IMS Messages and Codes Volumes 1 and 2*, GC27-1301 and GC27-1302
- *IMS Open Transaction Manager Access Guide and Reference*, SC18-7829
- *IMS Utilities Reference: System*, SC27-1309

General information about IMS Batch Terminal Simulator for z/OS is available on the Web at www.ibm.com/software/data/db2imstools/library.html

IMS DataPropagator
- *IMS DataPropagator for z/OS Administrator's Guide for Log*, SC27-1216
- *IMS DataPropagator: An Introduction*, GC27-1211
- *IMS DataPropagator for z/OS Reference*, SC27-1210

ISPF
- *z/OS ISPF Dialog Developer's Guide*, SC23-4821
- *z/OS ISPF Messages and Codes*, SC34-4815
- *z/OS ISPF Planning and Customizing*, GC34-4814
- *z/OS ISPF User's Guide Volumes 1 and 2*, SC34-4822 and SC34-4823

Language Environment
- *Debug Tool User's Guide and Reference*, SC18-7171
- *Debug Tool for z/OS and OS/390 Reference and Messages*, SC18-7172
- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562

MQSeries®
- *MQSeries Application Messaging Interface*, SC34-5604

- *MQSeries for OS/390 Concepts and Planning Guide*, GC34-5650
- *MQSeries for OS/390 System Setup Guide*, SC34-5651

National Language Support
- *National Language Design Guide Volume 1*, SE09-8001
- *IBM National Language Support Reference Manual Volume 2*, SE09-8002

NetView®
- *Tivoli NetView for z/OS Installation: Getting Started*, SC31-8872
- *Tivoli NetView for z/OS User's Guide*, GC31-8849

Microsoft ODBC

Information about Microsoft ODBC is available at http://msdn.microsoft.com/library/

Parallel Sysplex Library
- *System/390 9672 Parallel Transaction Server, 9672 Parallel Enterprise Server, 9674 Coupling Facility System Overview For R1/R2/R3 Based Models*, SB10-7033
- *z/OS Parallel Sysplex Application Migration*, SA22-7662
- *z/OS Parallel Sysplex Overview: An Introduction to Data Sharing and Parallelism*, SA22-7661
- *z/OS Parallel Sysplex Test Report*, SA22-7663

The *Parallel Sysplex Configuration Assistant* is available at www.ibm.com/s390/pso/psotool

PL/I: Enterprise PL/I for z/OS
- *IBM Enterprise PL/I for z/OS Language Reference*, SC27-1460
- *IBM Enterprise PL/I for z/OS Programming Guide*, SC27-1457

PL/I: PL/I for MVS & VM
- *PL/I for MVS & VM Programming Guide*, SC26-3113

SMP/E
- *SMP/E for z/OS and OS/390 Reference*, SA22-7772
- *SMP/E for z/OS and OS/390 User's Guide*, SA22-7773

Storage Management
- *z/OS DFSMS: Implementing System-Managed Storage*, SC26-7407
- *MVS/ESA Storage Management Library: Managing Data*, SC26-7397
- *MVS/ESA Storage Management Library: Managing Storage Groups*, SC35-0421
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide*, GC26-7398

System Network Architecture (SNA)
- *SNA Formats*, GA27-3136
- *SNA LU 6.2 Peer Protocols Reference*, SC31-6808
- *SNA Transaction Programmer's Reference Manual for LU Type 6.2*, GC30-3084
- *SNA/Management Services Alert Implementation Guide*, GC31-6809

TCP/IP
- *IBM TCP/IP for MVS: Customization & Administration Guide*, SC31-7134
- *IBM TCP/IP for MVS: Diagnosis Guide*, LY43-0105
- *IBM TCP/IP for MVS: Messages and Codes*, SC31-7132
- *IBM TCP/IP for MVS: Planning and Migration Guide*, SC31-7189

TotalStorage™ Enterprise Storage Server
- *RAMAC Virtual Array: Implementing Peer-to-Peer Remote Copy*, SG24-5680
- *Enterprise Storage Server Introduction and Planning*, GC26-7444
- *IBM RAMAC Virtual Array*, SG24-6424

Unicode
- *z/OS Support for Unicode: Using Conversion Services*, SA22-7649

Information about Unicode, the Unicode consortium, the Unicode standard, and standards conformance requirements is available at www.unicode.org

VTAM
- *Planning for NetView, NCP, and VTAM*, SC31-8063
- *VTAM for MVS/ESA Diagnosis*, LY43-0078
- *VTAM for MVS/ESA Messages and Codes*, GC31-8369
- *VTAM for MVS/ESA Network Implementation Guide*, SC31-8370
- *VTAM for MVS/ESA Operation*, SC31-8372
- *z/OS Communications Server SNA Programming*, SC31-8829
- *z/OS Communicatons Server SNA Programmer's LU 6.2 Reference*, SC31-8810
- *VTAM for MVS/ESA Resource Definition Reference*, SC31-8377

WebSphere family

- *WebSphere MQ Integrator Broker: Administration Guide*, SC34-6171
- *WebSphere MQ Integrator Broker for z/OS: Customization and Administration Guide*, SC34-6175
- *WebSphere MQ Integrator Broker: Introduction and Planning*, GC34-5599
- *WebSphere MQ Integrator Broker: Using the Control Center*, SC34-6168

z/Architecture™

- *z/Architecture Principles of Operation*, SA22-7832

z/OS

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS Communications Server: IP Configuration Guide*, SC31-8875
- *z/OS DCE Administration Guide*, SC24-5904
- *z/OS DCE Introduction*, GC24-5911
- *z/OS DCE Messages and Codes*, SC24-5912
- *z/OS Information Roadmap*, SA22-7500
- *z/OS Introduction and Release Guide*, GA22-7502
- *z/OS JES2 Initialization and Tuning Guide*, SA22-7532
- *z/OS JES3 Initialization and Tuning Guide*, SA22-7549
- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Managed System Infrastructure for Setup User's Guide*, SC33-7985
- *z/OS MVS Diagnosis: Procedures*, GA22-7587
- *z/OS MVS Diagnosis: Reference*, GA22-7588
- *z/OS MVS Diagnosis: Tools and Service Aids*, GA22-7589
- *z/OS MVS Initialization and Tuning Guide*, SA22-7591
- *z/OS MVS Initialization and Tuning Reference*, SA22-7592
- *z/OS MVS Installation Exits*, SA22-7593
- *z/OS MVS JCL Reference*, SA22-7597
- *z/OS MVS JCL User's Guide*, SA22-7598
- *z/OS MVS Planning: Global Resource Serialization*, SA22-7600
- *z/OS MVS Planning: Operations*, SA22-7601
- *z/OS MVS Planning: Workload Management*, SA22-7602
- *z/OS MVS Programming: Assembler Services Guide*, SA22-7605
- *z/OS MVS Programming: Assembler Services Reference, Volumes 1 and 2*, SA22-7606 and SA22-7607
- *z/OS MVS Programming: Authorized Assembler Services Guide*, SA22-7608
- *z/OS MVS Programming: Authorized Assembler Services Reference Volumes 1-4*, SA22-7609, SA22-7610, SA22-7611, and SA22-7612
- *z/OS MVS Programming: Callable Services for High-Level Languages*, SA22-7613
- *z/OS MVS Programming: Extended Addressability Guide*, SA22-7614
- *z/OS MVS Programming: Sysplex Services Guide*, SA22-7617
- *z/OS MVS Programming: Sysplex Services Reference*, SA22-7618
- *z/OS MVS Programming: Workload Management Services*, SA22-7619
- *z/OS MVS Recovery and Reconfiguration Guide*, SA22-7623
- *z/OS MVS Routing and Descriptor Codes*, SA22-7624
- *z/OS MVS Setting Up a Sysplex*, SA22-7625
- *z/OS MVS System Codes* SA22-7626
- *z/OS MVS System Commands*, SA22-7627
- *z/OS MVS System Messages Volumes 1-10*, SA22-7631, SA22-7632, SA22-7633, SA22-7634, SA22-7635, SA22-7636, SA22-7637, SA22-7638, SA22-7639, and SA22-7640
- *z/OS MVS Using the Subsystem Interface*, SA22-7642
- *z/OS Planning for Multilevel Security and the Common Criteria*, SA22-7509
- *z/OS RMF User's Guide*, SC33-7990
- *z/OS Security Server Network Authentication Server Administration*, SC24-5926
- *z/OS Security Server RACF Auditor's Guide*, SA22-7684
- *z/OS Security Server RACF Command Language Reference*, SA22-7687
- *z/OS Security Server RACF Macros and Interfaces*, SA22-7682
- *z/OS Security Server RACF Security Administrator's Guide*, SA22-7683
- *z/OS Security Server RACF System Programmer's Guide*, SA22-7681
- *z/OS Security Server RACROUTE Macro Reference*, SA22-7692
- *z/OS Support for Unicode: Using Conversion Services*, SA22-7649
- *z/OS TSO/E CLISTs*, SA22-7781
- *z/OS TSO/E Command Reference*, SA22-7782

- *z/OS TSO/E Customization*, SA22-7783
- *z/OS TSO/E Messages*, SA22-7786
- *z/OS TSO/E Programming Guide*, SA22-7788
- *z/OS TSO/E Programming Services*, SA22-7789
- *z/OS TSO/E REXX Reference*, SA22-7790
- *z/OS TSO/E User's Guide*, SA22-7794
- *z/OS UNIX System Services Command Reference*, SA22-7802
- *z/OS UNIX System Services Messages and Codes*, SA22-7807
- *z/OS UNIX System Services Planning*, GA22-7800
- *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, SA22-7803
- *z/OS UNIX System Services User's Guide*, SA22-7801

Index

E

F

G

H

T

U

W

Z

Readers' Comments — We'd Like to Hear from You

DB2 Universal Database for z/OS
Application Programming
Guide and Reference
FOR JAVA™
Version 8

Publication No. SC18-7414-03

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name _____ Address _____

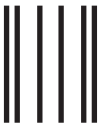
Company or Organization _____

Phone No. _____

Readers' Comments — We'd Like to Hear from You
SC18-7414-03

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines
Corporation
Reader Comments
DTX/E269
555 Bailey Avenue
San Jose, CA 95141-9989
U. S. A.

Fold and Tape **Please do not staple** Fold and Tape

Cut or Fold
Along Line

Program Number: 5625-DB2

Printed in USA

SC18-7414-03