DB2 Universal Database for OS/390 and z/OS

Application Programming Guide and Reference FOR JAVA™

Version 7

SC26-9932-01

DB2 Universal Database for OS/390 and z/OS

Application Programming Guide and Reference FOR JAVA™

Version 7

SC26-9932-01

Note

Before using this information and the product it supports, be sure to read the general information under “Notices” on page 117.

Second Edition, Softcopy Only (August 2001)

This edition applies to Version 7 of IBM DATABASE 2 Universal Database Server for OS/390 and z/OS (DB2 for OS/390 and z/OS), 5675-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 OS/390 and 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/os390/library.html). Check these resources to ensure that you are using the most current information.

Contents

About this book . . . vii
Who should read this book . . . vii
Product terminology and citations . . . vii
How to read the syntax diagrams . . . vii
How to send your comments . . . ix

Summary of changes to this book . . . xi

Chapter 1. JDBC application support . . . 1
What is JDBC? . . . 1
  JDBC background information . . . 1
  Advantages of using DB2 JDBC . . . 1
DB2's JDBC implementation . . . 1
  Connecting to a data source . . . 2
  DB2 for OS/390 and z/OS SQLJ/JDBC driver . . . 8
  JDBC API . . . 9
Running a JDBC application . . . 9
  Getting started . . . 9

Chapter 2. Writing SQLJ programs for DB2 for OS/390 and z/OS . . . 15
Executing SQL statements in an SQLJ program . . . 16
  Including SQL statements in an SQLJ program . . . 16
  Using Java variables and expressions as host expressions . . . 16
  Including comments . . . 17
  Handling SQL errors and warnings . . . 17
Including code to access SQLJ support . . . 19
Connecting to a data source . . . 19
Using result set iterators to retrieve rows from a result table . . . 23
  Using positioned iterators . . . 24
  Using named iterators . . . 25
  Using iterators for positioned UPDATE and DELETE operations . . . 27
  Using JDBC result sets in SQLJ applications . . . 28
Controlling the execution of SQL statements . . . 30
Retrieving multiple result sets from a stored procedure . . . 31
Setting the isolation level for a transaction . . . 32
Setting the read-only mode for a transaction . . . 33
An SQLJ sample program . . . 33
Running SQLJ programs . . . 35
Diagnosing SQLJ problems . . . 35
  Formatting trace data. . . . 36
  Running diagnosis utilities . . . 37

Chapter 3. SQLJ statement reference . . . 39
Common elements . . . 40
  host-expression . . . 40
  implements-clause . . . 41
  with-clause . . . 42

connection-declaration-clause . . . 44
  Syntax . . . 44
  Description . . . 44
  Usage notes . . . 44
iterator-declaration-clause . . . 45
  Syntax . . . 45
  Description . . . 45
  Usage notes . . . 46
executable-clause . . . 47
  Syntax . . . 47
  Usage notes . . . 47
context-clause . . . 47
  Syntax . . . 47
  Description . . . 47
  Usage notes . . . 48
statement-clause . . . 48
  Syntax . . . 48
  Description . . . 48
  Usage notes . . . 49
assignment-clause . . . 50
  Syntax . . . 50
  Description . . . 50
  Usage notes . . . 50
iterator-conversion-clause . . . 50
  Syntax . . . 51
  Description . . . 51
  Usage notes . . . 51
SET-TRANSACTION-clause . . . 51
  Syntax . . . 51
  Description . . . 51
  Usage notes . . . 52

Chapter 4. Creating Java stored procedures and user-defined functions . . . 53
Setting up the environment for Java routines . . . 53
  The stored procedure address space for Java routines . . . 53
  Setting environment variables for Java routines . . . 54
| Defining a Java routine to DB2 . . . 56
| Defining a JAR file for a Java routine to DB2 . . . 62
| Calling SQLJ.INSTALL_JAR . . . 62
| Calling SQLJ.REPLACE_JAR . . . 63
| Calling SQLJ.REMOVE_JAR . . . 63
Writing a Java routine . . . 64
  Differences between Java routines and stand-alone Java programs . . . 64
  Differences between Java routines and other routines . . . 64
  Writing a Java stored procedure to return result sets . . . 65
Testing a Java routine . . . 66

Chapter 5. Preparing Java programs . . . 69
Steps in the SQLJ program preparation process . . . 69
  Translating and compiling SQLJ source code . . . 70

Customizing an SQLJ serialized profile . . . 72
Binding a plan for an SQLJ program. . . . 75
Customizing SQLJ and JDBC to work together . . . 76
Preparing Java routines for execution . . . 76
| Preparing interpreted Java routines for execution . . . 76
Preparing compiled Java stored procedures for execution . . . 78
Building an SQLJ or JDBC program under VisualAge for Java . . . 80
Binding an SQLJ or JDBC program for VisualAge for Java . . . 81
Creating symbolic links for packages in a Java DLL . . . 86

Chapter 6. JDBC and SQLJ administration . . . 89
Installing JDBC and SQLJ . . . 89
Loading the JDBC and SQLJ libraries . . . 90
Setting the program control extended attribute . . . 90
Setting environment variables . . . 91
Customizing parameters in the SQLJ/JDBC run-time properties file . . . 92
Customizing the JDBC profile (optional) . . . 95
Binding the DBRMs . . . 96
Customizing the cursor properties file (optional) . . . 97
Installing VisualAge for Java support for JDBC and SQLJ . . . 97
JDBC and SQLJ security model . . . 102
How are authorization IDs established? . . . 102
DB2 attachment types . . . 102
JDBC and SQLJ multiple OS/390 context support . . . 103
Connecting when multiple OS/390 context support is not enabled . . . 103
Connecting when multiple OS/390 context support is enabled . . . 103
Enabling multiple OS/390 context support . . . 104
Multiple context performance . . . 104
Connection sharing . . . 104
JDBC and SQLJ connection pooling support . . . 105
JDBC and SQLJ global transaction support . . . 105

Appendix A. Selected sqlj.runtime classes and interfaces . . . 109
sqlj.runtime.ExecutionContext class . . . 109
sqlj.runtime.ConnectionContext interface . . . 110
sqlj.runtime.ForUpdate interface . . . 110
sqlj.runtime.NamedIterator interface . . . 111
sqlj.runtime.PositionedIterator interface . . . 111
sqlj.runtime.ResultSetIterator interface . . . 111

Appendix B. Special considerations for CICS applications . . . 113
Choosing parameter values for the SQLJ/JDBC run-time properties file . . . 113
Choosing parameter values for the db2genJDBC utility . . . 113
Choosing the number of cursors for JDBC result sets . . . 114
Setting environment variables for the CICS environment. . . . 114
Choosing VisualAge for Java bind parameters for better performance . . . 114
Connecting to DB2 in the CICS environment . . . 114
Commit and rollback processing in CICS SQLJ and JDBC applications . . . 115
Abnormal terminations in the CICS attachment facility . . . 115
Running traces in a CICS environment . . . 115

Notices 117
Programming interface information 118
Trademarks 119

Glossary 121

Bibliography 139

Index 147

About this book

This book describes DB2® for OS/390® and z/OS Java™ Edition, a feature of DB2 for OS/390 and z/OS that lets you access relational databases from Java application programs.

Who should read this book

This book is for DB2 for OS/390 application developers who are familiar with Structured Query Language (SQL) and who know the Java programming language.

Product terminology and citations

In this book, DB2 Universal Database™ Server for OS/390 and z/OS is referred to as "DB2 for OS/390 and z/OS." In cases where the context makes the meaning clear, DB2 for OS/390 and z/OS is referred to as "DB2." When this book refers to other books in this library, a short title is used. (For example, "See *DB2 SQL Reference*" is a citation to *IBM® DATABASE 2™ Universal Database Server for OS/390 and z/OS SQL Reference*.)

When referring to a DB2 product other than DB2 for OS/390 and z/OS, this book 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.
C and C language	Represent the C programming language.
CICS®	Represents CICS/ESA® and CICS Transaction Server for OS/390.
IMS™	Represents IMS or IMS/ESA®.
MVS	Represents the MVS element of OS/390.
OS/390	Represents the OS/390 or z/OS operating system.
RACF®	Represents the functions that are provided by the RACF component of the SecureWay® Security Server for OS/390 or by the RACF component of the OS/390 Security Server.

How to read the syntax diagrams

The following rules apply to the syntax diagrams 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).

```
►►—required_item————————————————————————————————————————————————►◄
```

- Optional items appear below the main path.

```
►►—required_item——┬————————————————┬————————————————————————————►◄
                  └—optional_item—┘
```

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.

```
                  ┌—optional_item—┐
►►—required_item——┴————————————————┴————————————————————————————►◄
```

- 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.

```
►►—required_item——┬—required_choice1—┬——————————————————————————►◄
                  └—required_choice2—┘
```

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

```
►►—required_item——┬———————————————————┬—————————————————————————►◄
                  ├—optional_choice1—┤
                  └—optional_choice2—┘
```

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

```
                  ┌—default_choice——┐
►►—required_item——┼——————————————————┼——————————————————————————►◄
                  ├—optional_choice—┤
                  └—optional_choice—┘
```

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

```
                  ┌————————————————————┐
                  ▼                    │
►►—required_item———repeatable_item———┴——————————————————————————►◄
```

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

```
                 ┌─,──────────────┐
                 ▼                │
►►──required_item───repeatable_item─┴──────────────────────────────►◄
```

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.

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 for OS/390 and z/OS documentation. You can use any of 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).
- Send your comments from the Web. Visit the Web site at:

  http://www.ibm.com/software/db2os390

  The Web site has a feedback page that you can use to send comments.
- Complete the readers' comment form at the back of the book and return it by mail, by fax (800-426-7773 for the United States and Canada), or by giving it to an IBM representative.

Summary of changes to this book

The principle changes to this book are:

- Chapter 4. Creating Java stored procedures and user-defined functions contains information on writing and running Java routines and on preparing programs for execution under VisualAge® for Java.
- Appendix B. Special considerations for CICS applications contains information on running JDBC™ and SQLJ programs in the CICS environment.

Chapter 1. JDBC application support

This chapter explains DB2 for OS/390 and z/OS's support for applications using JavaSoft JDBC interfaces to access DB2 data. It provides an overview that explains what JDBC is, more detailed information about DB2 for OS/390 and z/OS's implementation of JDBC, and guidelines for writing a JDBC program.

What is JDBC?

JDBC is a Java application programming interface (API) that Java applications use to access any relational database. DB2 for OS/390 and z/OS's support for JDBC enables you to write Java applications that access local DB2 data or remote relational data on a server that supports DRDA®. DB2 for OS/390 and z/OS is fully compliant with the JavaSoft JDBC 1.2 specification.

JDBC background information

To understand JDBC, knowing about its purpose and background is helpful. Sun Microsystems developed the specifications for a set of APIs that allow Java applications to access relational data. The purpose of the APIs is to provide a generic interface for writing platform-independent applications that can access any SQL database. The APIs are defined within 16 classes that support basic SQL functionality for connecting to a database, executing SQL statements, and processing results. Together, these interfaces and classes represent the JDBC capabilities by which a Java application can access relational data.

Advantages of using DB2 JDBC

DB2 JDBC offers a number of advantages for accessing DB2 data:

- JDBC combines the benefit of running your applications in an OS/390 environment with the portability and ease of writing Java applications. Using the Java language, you can write an application on any platform and execute it on any platform to which the Java Development Kit (JDK™) is ported.
- The ability to develop an application once and execute it anywhere offers the potential benefits of reduced development, maintenance, and systems management costs, and flexibility in supporting diverse hardware and software configurations.
- The JDBC interface offers the ability to change between drivers and access a variety of databases without recoding your Java program.
- JDBC applications do not require precompiles.

DB2's JDBC implementation

DB2 for OS/390 and z/OS provides the following implementations of JDBC:

- The ***SQLJ/JDBC driver with JDBC 1.2 support***, which is fully compliant with the JDBC 1.2 and SQLJ – Part 0 specification

  To use this version of JDBC, you need the JDK for OS/390, Version 1.1.6 or higher.

- # The ***SQLJ/JDBC driver with JDBC 2.0 support***, which is fully compliant with the
JDBC 1.2 and SQLJ – Part 0 specification and includes most of the functions of the
JDBC 2.0 specification

  # To use this version of JDBC, you need the JDK for OS/390, Version 1.3 or higher.

  # You need this JDBC driver if you use any of the following functions:
  - # The JDBC 2.0 DataSource function
  - # Global transactions that run under WebSphere Application Server Version 4.0 and
above

You select a version of the JDBC driver by specifying the associated file name in your CLASSPATH environment variable. See “Setting environment variables” on page 91 for details.

You can find the JDBC 1.2 and JDBC 2.0 specifications at the JDBC Web site: http://java.sun.com/products/jdbc. You should familiarize yourself with the specifications to understand how to use the JDBC APIs. Documentation that includes detailed information about each of the JDBC API interfaces, classes, and exceptions is also available at this Web site.

The JDK includes a Java compiler, Java Virtual Machine (JVM), and Java Debugger. You can learn more about the JDK from the Java for OS/390 Web site: http://www.ibm.com/s390/java.

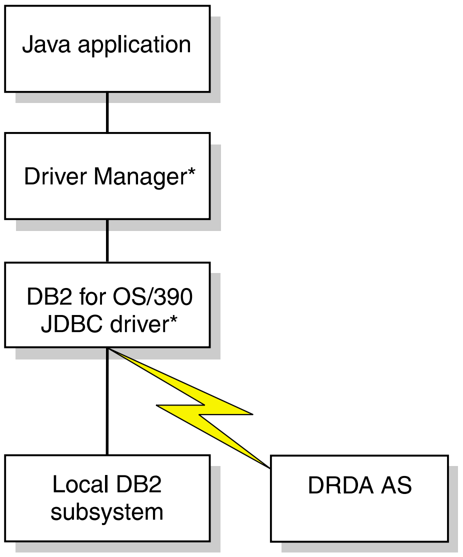
Connecting to a data source

There are two ways to connect to a data source. One method is available when you use the JDBC 1.2 driver or later. The other method is available only when you use the JDBC 2.0 driver or later.

Connecting to a data source using the JDBC 1.2 driver

Under the JDBC 1.2 specification, a JDBC application establishes a connection to a data source using the JDBC `DriverManager` interface, which is part of the `java.sql` package.

Figure 1 on page 3 shows how a Java application uses the `DriverManager` interface to connect to the DB2 for OS/390 and z/OS SQLJ/JDBC driver.



*Java byte code executed under JVM

Figure 1. Java application flow

The Java application first loads the JDBC driver (by invoking the `Class.forName()`method), in this case the DB2 for OS/390 and z/OS SQLJ/JDBC driver, and subsequently connects to the local DB2 subsystem or a remote DRDA database server (by invoking the `DriverManager.getConnection` method, described in "Getting started" on page 9).

The Java application identifies the target data source it wants to connect to by passing a database Uniform Resource Locator (URL) to the `DriverManager`.

The basic structure for the URL is:

```
jdbc:<subprotocol>:<subname>
```

Specify either of the following URL values for a DB2 for OS/390 and z/OS data source:

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

Each format results in the same behavior. Both subprotocols are provided for compatibility with existing DB2 for OS/390 and z/OS JDBC applications.

If *location-name* is not the local site, *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 for OS/390 and z/OS data source, there are two URL values that have special meaning for the DB2 for OS/390 and z/OS SQLJ/JDBC driver.

- If a URL value does not specify a location-name, for example, ″jdbc:db2os390:″, you will be connected to the local DB2 site. This format of the URL value is a DB2 for OS/390 and z/OS-defined extension. By using this URL value, a DB2 for OS/390 and z/OS JDBC application does not need to know the location-name of the local DB2 subsystem that the driver is using.
- The SQLJ specification defines the following URL:

  ```
  jdbc:default:connection
  ```

  When you use this URL value, your application is connected to the local DB2 site.

For information on how using the default URL affects a CICS connection, see "Connecting to DB2 in the CICS environment" on page 114.

The JDBC driver that is used to create the `java.sql.Connection` must first be registered with the JDBC driver manager. The driver manager searches all of the registered `java.sql.Driver` implementations for a driver that is capable of accepting the database URL. The driver manager then invokes the first of the registered JDBC drivers that supports the subprotocol that is specified in the URL. The application can then use the `DriverManager.getConnection` method to obtain a `java.sql.Connection`. For example:

```
include java.sql.*;
Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
⋮
Connection conn = DriverManager.getConnection("jdbc:db2os390:db2loc1");
```

This method lacks 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 and driver implementation. If your applications need to be portable among data sources, you should use the JDBC 2.0 driver.

Connecting to a data source using the JDBC 2.0 driver

The DB2 for OS/390 and z/OS SQLJ/JDBC driver with JDBC 2.0 support includes
JDBC 2.0 data source support, Java Naming and Directory Interface (JNDI) support,
and connection pooling support. (See "JDBC and SQLJ connection pooling support" on
page 105 for more information.) When you connect to a data source using the
SQLJ/JDBC driver with JDBC 2.0 support, your application can reference a data source
using a logical name, rather than an explicit driver class name and URL. In addition,
you can define or modify the data source attributes without changing the JDBC
application program.

JDBC 2.0 data source support is a complete replacement for the previous JDBC driver
manager support. You can use both types of support in the same application, but it is
recommended that you use data source support to obtain connections, regardless of
whether you use connection pooling or distributed transactions.

The application uses JNDI to associate a logical name with a specific data source implementation. This data source object contains all of the information that is necessary to determine the correct JDBC driver and return a `java.sql.Connection` object to the specified data source.

Creating a DB2DataSource instance: A `DataSource` object is the Java representation of a data source. The `DataSource` interface provides methods for connecting to the data source.

A `DataSource` object works with a JNDI naming service. A JNDI naming service provides a way for an application to access remote services over a network. Before you can use a `DataSource` object, the object needs to be created and registered with JNDI. In general, creating and managing `DataSource` objects is not done by the applications that use those objects.

If you use an application server, such as WebSphere, to write your JDBC applications, the application server creates and registers `DataSource` objects for you. This section contains information that you need if you create the `DataSource` objects yourself.

Two DB2 for OS/390 JDBC classes implement the `javax.sql.DataSource` interface:

- `com.ibm.db2.jcc.DB2DataSource` includes the methods that are necessary for a JDBC application to establish a connection to a data source using JDBC 2.0 support. This class also contains support for connection pooling.
- `com.ibm.db2.jcc.DB2SimpleDataSource` includes the same methods and properties as `com.ibm.db2.jcc.DB2DataSource`, but this class does *not* contain support for connection pooling. Because CICS does not support connection pooling, you need to use this class for CICS applications.

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 data source object. If the location name is not the local site (see the description of the DB2SQLJSSID property in "Customizing parameters in the SQLJ/JDBC run-time properties file" on page 92), 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 `database` 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. The methods to get and set the value of this property are:

getDatabaseName

Format:

```
java.lang.String getDatabaseName()
```

Returns the location name of the data source.

setDatabaseName

Format:

```
void setDatabaseName(java.lang.String dbname)
```

Sets the location name of the data source.

description

Describes the data source object. This property has data type String. The default value is null. The methods to get and set the value of this property are:

getDescription

Format:

```
java.lang.String getDescription()
```

Returns a description of the data source.

setDescription

Format:

```
void setDescription(java.lang.String description)
```

Sets the description of the data source.

user

Specifies the OS/390 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 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. For more information on how DB2 validates the user ID and password, see "JDBC and SQLJ security model" on page 102.

This property has data type String. The default value is null. The methods to get and set the value of this property are:

getUser

Format:

```
java.lang.String getUser()
```

Returns the user ID to be used when connecting to the data source.

setUser

Format:

```
void setUser(java.lang.String user)
```

Sets the user ID that is to be used for connecting to the data source.

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. The method to set the value of this this property is:

setPassword
Format:

```
void setPassword(java.lang.String password)
```

Sets the password that is to be used for connecting to the data source.

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. The methods to get and set the value of this property are:

getPlanName
Format:

```
 java.lang.String getPlanName()
```

Gets the name of the plan that DB2 allocates for connections that are established using the data source object.

setPlanName
Format:

```
void setPlanName(java.lang.String planName)
```

Sets the name of the plan that DB2 allocates for connections that are established using the data source object.

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. The methods to get and set the value of this property are:

getLoginTimeout
Format:

```
int getLoginTimeout()
```

Gets the maximum time in seconds to wait for a connection.

setLoginTimeout
Format:

```
void setLoginTimeout(int seconds)
```

Sets the maximum time in seconds to wait for a connection.

Examples of creating and deploying a DB2DataSource instance: Example: Creating a DB2DataSource instance and registering it with JNDI:

```
#     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
#
#               DB2DataSource db2ds = new com.ibm.db2.jcc.DB2DataSource();  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
```

1 Creates an instance of the DB2DataSource class.
2 This statement and the next three statements set values for properties of this
`DB2DataSource` instance.
3 Creates a context for use by JNDI.
4 Associates `DB2DataSource` object db2ds with the logical name jdbc/sampledb. An
application that uses this object can refer to it by the name jdbc/sampledb.
#

Example: Retrieving a DB2DataSource instance and obtaining a connection:

```
#               import java.sql.*;
#               import javax.naming.*;
#               import javax.sql.*;
#
#               Context ctx=new InitialContext();                         1
#               DataSource ds=(DataSource)ctx.lookup("jdbc/sampledb");    2
#               Connection con=ds.getConnection();                        3
```

1 Creates a context for use by JNDI.
2 Calls the lookup method to get the `DataSource` object that is associated with logical
name jdbc/sampledb. This object is the same as object db2ds in the previous example.
3 Establishes a connection to the data source that `DataSource` object ds represents.
#

DB2 for OS/390 and z/OS SQLJ/JDBC driver

The DB2 for OS/390 and z/OS SQLJ/JDBC driver is implemented as a type 2 driver, one of four types of JDBC drivers defined by JavaSoft. The type 2 driver translates JDBC calls into calls to a DB2 language interface module.

Several packages are included with the DB2 for OS/390 and z/OS SQLJ/JDBC driver. These packages represent the DB2 for OS/390 and z/OS implementation of the `java.sql` JDBC API. The driver packages include all of the JDBC classes, interfaces, and exceptions that comply with the JDBC 1.2 specification and some support for the JDBC 2.0 specification.

The DB2 for OS/390 and z/OS SQLJ/JDBC driver is available under two different Java class names. The preferred driver name is:

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

However, to maintain compatibility with existing DB2 for OS/390 and z/OS JDBC applications, the following driver name is also supported:

```
ibm.sql.DB2Driver
```

The `ibm.sql.DB2Driver` class will automatically forward all driver API calls to the `COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`.

JDBC API

The JDBC API consists of the abstract Java interfaces that an application program uses to access databases, execute SQL statements, and process the results. Like ODBC, JDBC is a dynamic SQL interface. Writing a JDBC application is similar to writing a C application using ODBC to access a database. The main interfaces that perform these functions are:

- The `DriverManager` and `DB2DataSource` classes create database connections. `DB2DataSource` is available in JDBC 2.0 and above. `DriverManager` is available in JDBC 1.2 and above.
- The `Connection` interface supports the connection to a specific database.
- The `Statement` interface supports all SQL statement execution. This interface has two underlying interfaces:
  - The `PreparedStatement` interface supports any SQL statement containing input parameter markers.
  - The `CallableStatement` interface supports the invocation of a stored procedure and allows the application to retrieve output parameters.
- The `ResultSet` interface provides access to the results that a query generates. The `ResultSet` interface is similar to the cursor that is used in SQL applications in other languages.

Running a JDBC application

When you create a Java application that uses the JDBC interfaces, you import the `java.sql` package and invoke methods according to the JDBC specification.

Getting started

Figure 2 on page 10 demonstrates a simple JDBC application. Assuming the JDBC driver is installed in `/usr/lpp/db2`, you can find source code for a similar sample program in the following path:

```
/usr/lpp/db2/db2710/samples/sample01.java
```

For additional information on preparing and running the sample program, see sample01.readme in the same directory.

```
import java.sql.*;                                                  1
import com.ibm.db2.jcc.DB2Diagnosable;
import com.ibm.db2.jcc.DB2Sqlca;

public class sample01 {

 static {
   try {
     // register the SQLJ/JDBC driver with DriverManager
     Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");    2
   } catch (ClassNotFoundException e) {
      e.printStackTrace();
   }
 }

 public static void main(String args[]) {

  String URLprefix = "jdbc:db2os390sqlj:";
  String url;
  try {
    System.out.println("**** JDBC Entry within class sample01.");

    // If an alternate URL is passed, then use it
    if (args.length > 0)
      url = new String(URLprefix + args[0]);
    else
      url = new String(URLprefix);   //else use "local" DB2 location
    // Create the connection                                        3
    String userid="myname";           //set user ID
    String password="mypass";         //set password
    Connection con = DriverManager.getConnection (url,userid,password);
    System.out.println("**** JDBC Connection to DB2 for OS/390.");
    // Create the Statement                                         4
    Statement stmt = con.createStatement();
    System.out.println("**** JDBC Statement Created");
```

Figure 2. Sample Java application (Part 1 of 4)

```
  // Execute a Query and generate a ResultSet instance        5
  // The Query is a Select from SYSIBM.SYSTABLES
  ResultSet rs = stmt.executeQuery("SELECT NAME FROM SYSIBM.SYSTABLES");
  System.out.println("**** JDBC Result Set Created");
  // Print all of the table names to sysout                   6
  while (rs.next()) {
    String s = rs.getString(1);
    System.out.println("Table NAME = " + s);
  }
  System.out.println("**** JDBC Result Set output completed");
  // Close the resultset                                      7
  rs.close();
  // Close the statement                                      8
  stmt.close();
  System.out.println("**** JDBC Statement Closed");
  // Close the connection                                     9
  con.close();
  System.out.println("**** JDBC Disconnect from DB2 for OS/390.");
  System.out.println("**** JDBC Exit from class sample01 - no  Errors.");
   } catch( SQLException sqle ) {
       System.out.println ("SQLException: " + sqle +
         ".  SQLSTATE=" + sqle.getSQLState() +
         " SQLCODE=" + sqle.getErrorCode());
         sqle.printStackTrace();
         //=====> Optional DB2-only error processing
         processDB2Diagnosable ( sqle );
   } catch( Exception e ) {
      System.out.println ("Exception: " + e );
      e.printStackTrace();
   }
 }
```

Figure 2. Sample Java application (Part 2 of 4)

```
// Retrieve DB2-only diagnosis information                          10
public static void processDB2Diagnosable (java.sql.SQLException sqle) {
  // Check whether SQLException object has DB2-only diagnostics
  if (sqle instanceof DB2Diagnosable) {
    // A DB2Diagnosable object has method getSqlca()
    DB2Sqlca sqlca = ((DB2Diagnosable)sqle).getSqlca();
    // For future portability, confirm that DB2Sqlca is not null
    if (sqlca != null) {
      // Retrieve SQLCODE
      int sqlCode = sqlca.getSqlCode();
      // Retrieve SQLERRMC
      // SQLERRML can be obtained from String length() method
      String sqlErrmc = sqlca.getSqlErrmc();
      // Obtain the tokens from SQLERRMC
      // Retrieve each token as a separate String in an array
      String[] sqlErrmcTokens = sqlca.getSqlErrmcTokens();
      // Retrieve SQLERRP
      String sqlErrp = sqlca.getSqlErrp();
      // Retrieve SQLERRD fields
      int[] sqlErrd = sqlca.getSqlErrd();
      // Retrieve SQLWARN flags
      char[] sqlWarn = sqlca.getSqlWarn();
      // Retrieve SQLSTATE
      String sqlState = sqlca.getSqlState();
      // Print all SQLCA fields
      System.err.println ( "--------------- DB2Sqlca ---------------" );
      System.err.println ( "SQLCODE: " + sqlCode );
      System.err.println ( "SQLERRMC: " + sqlErrmc );
      for (int i=0; i< sqlErrmcTokens.length; i++ ) {
        System.err.println ("  token " + i + ": " + sqlErrmcTokens[i]);
      }
```

Figure 2. Sample Java application (Part 3 of 4)

```
          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);
        }
      }
    }
}
```

Figure 2. Sample Java application (Part 4 of 4)

Notes to Figure 2 on page 10:

1 The first statement imports the appropriate Java package, `java.sql`. The second and third statements import optional DB2 packages for extended error processing.

2 The `Class.forName` method loads the appropriate JDBC driver, in this case, DB2 for OS/390 and z/OS SQLJ/JDBC driver (`COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`) and registers it with the `DriverManager`.

3 The `getConnection` method creates a `Connection` instance to connect to the database, specifying the location with a URL and using the DB2 subprotocol (as defined in the JDBC specification and explained in "Connecting to a data source using the JDBC 1.2 driver" on page 2). You must modify the URL in the sample01.java application to match the location name of your local DB2 for OS/390 and z/OS. When the application connects to DB2, DB2 checks that the user ID and password in the `getConnection` call are authorized to connect to DB2. DB2 does not authenticate connections through CICS or local connections.

4 The `createStatement` method creates a `Statement` instance.

5 The `executeQuery` method executes a query and generates a `ResultSet` instance.

6 The `next()` method on the `ResultSet` instance advances the iterator to successive rows of the result set. For each row, the `getString` method is called to retrieve column 1.

7 close() closes the result set.

8 close() closes the statement and frees all resources associated with the statement.

9 close() closes the connection and frees all resources associated with the connection.

10 The processDB2Diagnosable method calls methods in the DB2Diagnosable class to retrieve SQLCA fields. See "Handling SQL errors and warnings" on page 17 for more information.

After writing your program, compile it as you would any other Java program. No precompile or bind steps are required to run a Java program.

Chapter 2. Writing SQLJ programs for DB2 for OS/390 and z/OS

SQLJ provides support for embedded static SQL in Java applications and servlets. SQLJ was initially developed by Oracle, Tandem, and IBM 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 includes JDBC 1.2 or JDBC 2.0, an application program can create a JDBC connection and then use that connection to execute dynamic SQL statements through JDBC and embedded static SQL statements through SQLJ.

The SQLJ specification consists of three parts:

- **Database Languages – SQL – Part 10: Object Language Bindings (SQL/OLB)** is also known as SQLJ Part 0. It was approved by ANSI in 1998, and it specifies the SQLJ language syntax and semantics for embedded SQL statements in a Java application.
- **Database Languages – SQLJ – Part 1: SQL Routines using the Java Programming Language** was approved by ANSI in 1999, and it specifies extensions that define:
  - Installation of Java classes in an SQL database
  - Invocation of static methods as stored procedures
- **Database Languages – SQLJ – Part 2: SQL Types using the Java Programming Language** is under development. It specifies extensions for accessing Java classes as SQL user-defined types.

The DB2 for OS/390 and z/OS implementation of SQLJ includes support for the following portions of the specification:

- Part 0
- The ability to invoke a Java static method as a stored procedure, which is in Part 1

Some of the major differences between SQLJ and JDBC are:

- SQLJ follows the static SQL model, and JDBC follows the dynamic SQL model.
- SQLJ source programs are smaller than equivalent JDBC programs, because certain code that the programmer must include in JDBC programs is generated automatically by SQLJ.
- SQLJ can do data type checking during the program preparation process to determine whether table columns are compatible with Java host expressions. JDBC passes values to and from SQL tables without compile-time data type checking.
- In SQLJ programs, you can embed Java host expressions in SQL statements. JDBC requires a separate call statement for each bind variable and specifies the binding by position number.
- SQLJ provides the advantages of static SQL authorization checking. With SQLJ, the authorization ID under which SQL statements execute is the plan or package owner. DB2 checks table privileges at bind time. Because JDBC uses dynamic SQL, the authorization ID under which SQL statements execute is not known until run time, so no authorization checking of table privileges can occur until run time.

This chapter and the following two chapters explain DB2 for OS/390 and z/OS support for SQLJ. This chapter gives you the information that you need to write SQLJ programs that run on DB2 for OS/390 and z/OS. Subsequent chapters describe how to prepare SQLJ programs for execution and provide detailed syntax for the components of SQLJ.

The following topics are discussed in this chapter:

- "Executing SQL statements in an SQLJ program"
- "Including code to access SQLJ support" on page 19
- "Connecting to a data source" on page 19
- "Using result set iterators to retrieve rows from a result table" on page 23
- "Controlling the execution of SQL statements" on page 30
- "Retrieving multiple result sets from a stored procedure" on page 31
- "Setting the isolation level for a transaction" on page 32
- "Setting the read-only mode for a transaction" on page 33
- "An SQLJ sample program" on page 33
- "Running SQLJ programs" on page 35
- "Diagnosing SQLJ problems" on page 35

Executing SQL statements in an SQLJ program

This section discusses the following basic information about writing an SQLJ program:
- How to include SQL statements, host variables, and comments in the program
- Which SQL statements are valid in an SQLJ program
- How to do error handling

Including SQL statements in an SQLJ program

In an SQLJ program, all statements that are used for database access are in *SQLJ clauses*. SQLJ clauses that contain SQL statements are called *executable clauses*. An executable clause begins with the characters #sql and contains an SQL statement that is enclosed in curly brackets. The SQL statement itself has no terminating character. An example of an executable clause is:

```
#sql {DELETE FROM EMP};
```

"Controlling the execution of SQL statements" on page 30 contains a list of the SQL statements that you can include in an SQLJ program. An executable clause can appear anywhere in a program that a Java statement can appear.

Using Java variables and expressions as host expressions

To pass data between a Java application program and DB2, use host expressions. A Java host expression is a Java simple identifier or complex expression, preceded by a colon. The result of a complex expression must be a single value. An array element is considered to be a complex expression. A complex expression must be surrounded by parentheses. When you use a host expression as a parameter in a stored procedure call, you can follow the colon with the IN, OUT, or INOUT parameter, which indicates whether the host expression is intended for input, output, or both. The IN, OUT, or INOUT value must agree with the value you specify in the stored procedure definition in catalog table SYSIBM.SYSPROCEDURES.

The following SQLJ clause uses a host expression that is a simple Java variable named empname:

```
#sql {SELECT LASTNAME INTO :empname FROM EMP WHERE EMPNO='000010'};
```

The following SQLJ clause calls stored procedure A and uses a simple Java variable named EMPNO as an input or output parameter:

```
#sql {CALL A (:INOUT EMPNO)};
```

SQLJ evaluates host expressions from left to right before DB2 processes the SQL statements that contain them. For example, suppose that the value of i is 1 before the following SQL clause is executed:

```
#sql {SET :(z[i++]) = :(x[i++]) + :(y[i++])};
```

The array index that determines the location in array z is 1. The array index that determines the location in array x is 2. The array index that determines the location in array y is 3. The value of i in the Java space is now 4. The statement is then executed. After statement execution, the output value is assigned to z[1].

In an executable clause, host expressions, which are Java tokens, are case sensitive.

Including comments

To include comments in an SQLJ program, use either Java comments or SQL 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, use Java comments in host expressions.
- SQL comments are denoted by * at the beginning of a line or -- anywhere on a line in an SQL statement. You can use SQL comments in executable clauses, anywhere except in host expressions.

Handling SQL errors and warnings

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 SQLCODE
- When a SELECT INTO SQL statement returns a +100 SQLCODE

You can use the `getErrorCode` method to retrieve SQLCODEs 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 {SELECT LASTNAME INTO :empname
    FROM EMP WHERE EMPNO='000010'};
}
catch(SQLException e) {
  System.out.println("SQLCODE returned: " + e.getErrorCode());
}
```

Other than a +100 SQLCODE on a SELECT INTO statement, DB2 warnings do not throw SQLExceptions. To handle DB2 warnings, you need to import the `java.sql.SQLWarning` class. To check for a DB2 warning, invoke the `getWarnings` method after you execute an SQL clause. `getWarnings` returns the first warning code that an SQL statement generates. Subsequent SQL warning codes are chained to the first SQL warning code.

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” on page 30 for information on how to set up an execution context. The following example demonstrates how to retrieve an SQL warning code for an SQL clause with execution context ExecCtx:

```
SQLWarning SQLWarn;
#sql [ExecCtx] {SELECT LASTNAME INTO :empname
  FROM EMP WHERE EMPNO='000010'};
if (SQLWarn = ExecCtx.getWarnings() != null) then
System.out.println("SQLWarning " + SQLWarn);
```

If your SQLJ or JDBC application runs only on DB2 for OS/390 and z/OS, you can retrieve the contents of the SQLCA when an SQL statement generates an `SQLWarning` or `SQLException`. To retrieve the SQLCA, import the `com.ibm.db2.jcc.DB2Diagnosable` and `db2.jcc.DB2Sqlca` classes. Then, in the catch block for an `SQLException` or after the `getWarnings` call for an `SQLWarning`, call the `DB2Diagnosable.getSQLCA` method to retrieve the SQLCA and the `DB2Sqlca` methods to retrieve the fields in the SQLCA. For example:

```
import com.ibm.db2.jcc.DB2Diagnosable;
import com.ibm.db2.jcc.DB2Sqlca;
⋮

try {
  #sql {SELECT LASTNAME INTO :empname
    FROM EMP WHERE EMPNO='000010'};
}
catch(SQLException e) {
  System.out.println("SQLCODE returned: " + e.getErrorCode());
  if (e instanceof DB2Diagnosable) {                // If SQLException object has
                                                    // DB2-only diagnostics
    DB2Sqlca sqlca = ((DB2Diagnosable)e).getSqlca(); // Get the SQLCA
    if (sqlca != null) {
      int sqlCode = sqlca.getSqlCode();             // Get the SQLCA fields
⋮

    }
  }
}
```

# For a complete example of using DB2Diagnosable and DB2Sqlca, see sample SQLJ
# program sample02.sqlj, or sample JDBC program sample01.java, which are shipped
# with the SQLJ/JDBC driver.

Including code to access SQLJ support

Before you can execute any SQLJ clauses in your application program, you must import the Java packages for SQLJ run-time support and the JDBC interfaces that are used by SQLJ.

To import the Java packages for SQLJ and JDBC, include these lines in your application program:

```
import sqlj.runtime.*;         // SQLJ runtime support
import java.sql.*;             // JDBC interfaces
```

If you use JDBC 2.0 data source support to create connections, you also need to import these packages:

```
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
```

Connecting to a data source

In an SQLJ application, as in any other DB2 application, you must be connected to a data source before you can execute SQL statements. A data source in DB2 for OS/390 and z/OS is a DB2 location name.

To connect to a data source, you need to define a connection context class and then set up a connection context, using one of the following methods. The first two methods are available with either version of the SQLJ/JDBC driver. The third method is available only with version 2.0 of the SQLJ/JDBC driver.

- Connection method 1:
  1. Execute a type of SQLJ clause called a *connection declaration clause* to generate a connection context class.
  2. Load the DB2 for OS/390 and z/OS SQLJ/JDBC driver and register it with the DriverManager.

     To do that, invoke method Class.forName with an argument of COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver.
  3. Invoke the constructor for the connection context class with the following arguments:
     – A string that specifies the location name that is associated with the data source. That argument has the form:

       jdbc:db2os390sqlj:*location-name*

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

  – Two optional strings that specify the user ID that is making the connection to DB2, and the password for that user ID. If you do not specify these parameters, the application connects to DB2 without doing user ID and password authentication. Each string must be 255 bytes or less. If you do not specify these parameters, DB2 does not authenticate the connection. For a CICS connection or a local connection, DB2 never does authentication.
  – A boolean that specifies whether autoCommit is on or off for the connection.

  For example, suppose that you want to use the first method to set up connection context myconn to access data at a data source that is associated with location NEWYORK. For this connection, you want autoCommit to be off. You want to pass a user ID and password to DB2, which are in variables userid and password. First, execute a connection declaration clause to generate a connection context class:

  ```
  #sql context Ctx;
  ```

  Next, load the SQLJ/JDBC driver and register it with the `DriverManager`:

  ```
  try {
    Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
  }
  catch (ClassNotFoundException e) {
     e.printStackTrace();
  }
  ```

  Then invoke the constructor for generated class Ctx with arguments jdbc:db2os390sqlj:NEWYORK, userid, password, and `false`:

  ```
  Ctx myconn=new Ctx("jdbc:db2os390sqlj:NEWYORK",userid,password,false);
  ```

- Connection method 2:
  1. Execute a connection declaration clause to generate a connection context class.
  2. Load the DB2 for OS/390 and z/OS SQLJ/JDBC driver and register it with the `DriverManager`.

     To do that, invoke method `Class.forName` with an argument of `COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver`. For example:

     ```
     try {
       Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
     }
     catch (ClassNotFoundException e) {
        e.printStackTrace();
     }
     ```
  3. Invoke the JDBC `java.sql.DriverManager.getConnection` method.

     The first argument for java.sql.DriverManager.getConnection is a string that specifies the location name that is associated with the data source. That argument has the form:

     `jdbc:db2os390sqlj:`*`location-name`*

If *location-name* is not the local site, *location-name* must be defined in SYSIBM.LOCATIONS. 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. The invocation returns an instance of class `Connection`, which represents a JDBC connection to the data source.

| The second and third arguments, which are optional, are a user ID and password
| that DB2 uses to authenticate the connection to DB2. Both arguments are Java
| String variables. Each string must be 255 bytes or less. If you do not specify
| these parameters, DB2 does not authenticate the connection. For a CICS
| connection or a local connection, DB2 never does authentication.

4. If necessary, invoke the setAutoCommit method to enable or disable AutoCommit.

   For environments other than the CICS environment, the default state of autoCommit for a JDBC connection is on. To disable autoCommit, invoke the setAutoCommit method with an argument of `false`.
5. Invoke the constructor for the connection context class. For the argument of the constructor, use the JDBC connection that results from invoking `java.sql.DriverManager.getConnection`.

For example, suppose you want to use the second method to set up connection context myconn to access data at the data source associated with location NEWYORK, and you want to pass a user ID and password. First, execute a connection declaration clause to generate a connection context class:

```
#sql context Ctx;
```

Next, load the SQLJ/JDBC driver and register it with the `DriverManager`:

```
try {
  Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");
}
catch (ClassNotFoundException e) {
   e.printStackTrace();
}
```

Then invoke `java.sql.Driver.getConnection` with the arguments jdbc:db2os390sqlj:NEWYORK, userid, and password:

```
Connection jdbccon=
  DriverManager.getConnection("jdbc:db2os390sqlj:NEWYORK",userid,password);
```

Next, to set autoCommit off for the connection, invoke setAutoCommit with an argument `false`:

```
jdbccon.setAutoCommit(false);
```

Finally, invoke the constructor for class Ctx using the JDBC connection as the argument:

```
Ctx myconn=new Ctx(jdbccon);
```

• Connection method 3:

1. Execute a connection declaration clause to generate a connection context class.

2. Invoke the JNDI `javax.naming.InitialContext` method to create a context for use by JNDI.
3. Invoke the JNDI `javax.naming.lookup` method to get the `DataSource` object that is associated with a logical name.

   The `DataSource` object is created separately by an application server such as WebSphere.
4. Invoke the JDBC `java.sql.DataSource.getConnection` method to connect the `DataSource` object to the data source.
5. If necessary, invoke the setAutoCommit method to enable or disable AutoCommit.

   For environments other than the CICS environment, the default state of autoCommit for a JDBC connection is on. To disable autoCommit, invoke the setAutoCommit method with an argument of `false`.
6. Invoke the constructor for the connection context class. For the argument of the constructor, use the JDBC connection that results from invoking `java.sql.DataSource.getConnection`.

For example, suppose you want to use the second method to set up connection context myconn to access data at the data source with logical name jdbc/sampledb. First, execute a connection declaration clause to generate a connection context class:

```
#sql context Ctx;
```

Next, establish a JNDI context and find the DataSource object that is associated with the logical data source name:

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

Then establish a connection for the `DataSource` object:

```
Connection jdbccon=ds.getConnection();
```

Next, set autoCommit off for the connection:

```
jdbccon.setAutoCommit(false);
```

Finally, invoke the constructor for class Ctx using the JDBC connection as the argument:

```
Ctx myconn=new Ctx(jdbccon);
```

To execute an SQL statement at a data source, use one of the following methods:

- Use an *explicit connection.*

  Specify a *connection context*, enclosed in square brackets, at the beginning of the execution clause that contains the SQL statement. A connection context is an instance of a connection context class.

  For example, the following SQL clause executes an UPDATE statement at the data source that is associated with connection context myconn:

  ```
  #sql [myconn] {UPDATE DEPT SET MGRNO=:hvmgr WHERE DEPTNO=:hvdeptno};
  ```

- Use a *default connection.*

  When you specify an execution clause without a connection context, SQLJ uses the default context to access a data source. If you create a connection context object for accessing a remote data source, you can use the setDefaultContext method to install that connection context object as the default connection. If you do not use setDefaultContext to override the default connection, the default connection is to the local DB2 subsystem.

Important: Always use an explicit connection context for a program that runs in a multithreaded environment. For example, you should always use an explicit connection context in a Java servlet.

SQLJ uses the JDBC `java.sql.Connection` class to connect to data sources. Your application can invoke any method in the `java.sql.Connection` class.

Using result set iterators to retrieve rows from a result table

In DB2 application programs that are written in traditional host languages, you use a cursor to retrieve individual rows from the result table that is generated by a SELECT statement. The SQLJ equivalent of a cursor is a *result set iterator*. 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.

You define a result set iterator using an *iterator declaration clause*. The iterator declaration clause specifies the following information:

- 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. Table 1 shows each Java data type that you can specify in a result set iterator declaration and the equivalent JDBC and SQL data types.

Table 1. Equivalent Java, JDBC, and SQL data types

#	Java data type	JDBC data type	SQL data type
# #	java.lang.String	CHAR, VARCHAR, LONGVARCHAR	CHAR, VARCHAR, GRAPHIC, VARGRAPHIC, CLOB, DBCLOB
# #	java.math.BigDecimal	NUMERIC	NUMERIC, INTEGER, DECIMAL, SMALLINT, FLOAT, REAL, DOUBLE
#	boolean	BIT	INTEGER, SMALLINT
# #	int, Integer	INTEGER	SMALLINT, INTEGER, DECIMAL, NUMERIC, FLOAT, DOUBLE
# #	float, Float	REAL	SMALLINT, INTEGER, DECIMAL, NUMERIC, FLOAT, DOUBLE

Table 1. Equivalent Java, JDBC, and SQL data types (continued)

#	Java data type	JDBC data type	SQL data type
#	double, Double	DOUBLE	SMALLINT, INTEGER, DECIMAL, NUMERIC, FLOAT, DOUBLE
#	byte[][1]	BINARY, VARBINARY, LONGVARBINARY	CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, BLOB
#	java.sql.Date[2]	DATE	DATE
#	java.sql.Time[2]	TIME	TIME
#	java.sql.Timestamp[2]	TIMESTAMP	TIMESTAMP

Note:

1. Because this data type is equivalent to a DB2 data type with a subtype of BIT, SQLJ performs no conversion for
data of this type.

2. This class is part of the JDBC API.
#

If you declare an iterator without the public modifier, you can declare and use the iterator in the same file. If you declare the iterator as public, you can declare and use the iterator in one of the following ways:

- Declare the iterator in one file, and use it in a different file. The name of the file in which you declare the iterator must match the iterator name.
- Declare and use the iterator in the same file. If you do this, you need to declare the iterator with the public and static modifiers, and declare the iterator in the class that uses it.

Examples in this chapter that use a public iterator declare the iterator in a different file from the file in which it is used.

An iterator must have a public modifier when the iterator has a with clause, such as with UpdateColumns.

The two types of result set iterators are *positioned iterators* and *named iterators*. The type of result set iterator that you choose depends on the way that you plan to use that result set iterator. The following sections explain how to use each type of iterator.

Using positioned iterators

For a positioned 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 declare positioned iterators to execute FETCH statements.

For example, the following iterator declaration clause defines a positioned iterator named ByPos with two columns. The first column is of type `String`, and the second column is of type `Date`.

```
#sql iterator ByPos(String,Date);
```

When SQLJ encounters an iterator declaration clause for a positioned iterator, it generates a *positioned iterator class* with the name that you specify in the iterator declaration clause. You can then declare an object of the positioned iterator class to retrieve rows from a result table.

For example, suppose that you want to retrieve rows from a result table that contains the values of the LASTNAME and HIREDATE columns from the DB2 sample employee table. Figure 3 shows how you can declare an iterator named ByPos and use an object of the generated class ByPos to retrieve those rows.

```
{
  #sql iterator ByPos(String,Date);
                             // Declare positioned iterator class ByPos
  ByPos positer;             // Declare object of ByPos class
  String name = null;
  Date hrdate;
  #sql positer = { SELECT LASTNAME, HIREDATE FROM EMP };              1
  #sql { FETCH :positer INTO :name, :hrdate };                        2
                            // Retrieve the first row
  while ( !positer.endFetch() )                                       3
  { System.out.println(name + " was hired in " +
      hrdate);
    #sql { FETCH :positer INTO :name, :hrdate };
                           // Retrieve the rest of the rows
  }
}
```

Figure 3. Retrieving rows using a positioned iterator

Notes to Figure 3:

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 positer.

2 The DB2 for OS/390 and z/OS customizer can validate that the iterator type is compatible with the SQL data type of the corresponding column.

3 Method endFetch(), which is a method of the generated iterator class ByPos, returns a value of `true` when all rows have been retrieved from the iterator. The first FETCH statement needs to be executed before endFetch() is called.

Using named iterators

Using named iterators is an alternative way to select rows from a result table. 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.

When SQLJ encounters a named iterator declaration, it generates a *named iterator class* with the same name that you use in the iterator declaration clause. In the named iterator class, SQLJ generates an *accessor method* for each column name in the iterator declaration clause. The accessor method name is the same name as the column name in the iterator declaration clause. The data type that is returned by the accessor method is the same as the data type of the corresponding column in the iterator declaration clause.

When you execute an SQL clause that has a named iterator, SQLJ matches the name of each iterator column to the name of a column in the result table.

The following iterator declaration clause defines the named iterator ByName, which has two columns. The first column of the iterator is named LastName and is of type `String`. The second column is named HireDate and is of type `Date`.

```
#sql iterator ByName(String LastName, Date HireDate);
```

To use a named iterator, you use an SQLJ *assignment clause* to assign the result table from a SELECT statement to an instance of a named iterator class. Then you use the accessor methods to retrieve the data from the iterator.

Figure 4 shows how you can use a named iterator to retrieve rows from a result table that contains the values of the LASTNAME and HIREDATE columns of the employee table.

```
{
  #sql iterator ByName(String LastName, Date HireDate);          1
  ByName nameiter;           // Declare object of ByName class
  #sql nameiter={SELECT LASTNAME, HIREDATE FROM EMP};            2
  while (nameiter.next())                                        3
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate());
  }
}
```

Figure 4. Retrieving rows using a named iterator

Notes to Figure 4:

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 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 nameiter.

3 next(), which is a method of the generated class ByName, advances the iterator to successive rows of the result set. `next` returns a value of `true` when a next row is available, and a value of `false` when all rows have been fetched from the iterator.

The column names for named iterators must be valid Java identifiers. The column names must also match the column names in the result table from which the iterator retrieves rows. If a SELECT statement that uses a named iterator selects data from columns with names that are not valid Java identifiers, you need to use SQL AS clauses in the SELECT statement to give the columns of the result table acceptable names.

For example, suppose you want to use a named iterator to retrieve the rows that are specified by this SELECT statement:

```
SELECT PUBLIC FROM GOODTABLE
```

The iterator column name must match the column name of the result table, but you cannot specify an iterator column name of `PUBLIC` because `PUBLIC` is a reserved Java keyword. You must therefore use an AS clause to rename `PUBLIC` to a valid Java identifier in the result table. For example:

```
SELECT PUBLIC AS IS_PUBLIC FROM GOODTABLE
```

You can then declare a named iterator with a column name that is a valid Java identifier and matches the column name of the result table:

```
#sql iterator ByName(String IS_PUBLIC);
ByName nameiter;
#sql nameiter={SELECT PUBLIC AS IS_PUBLIC FROM GOODTABLE};
```

Using iterators for positioned UPDATE and DELETE operations

When you declare an iterator for a positioned UPDATE or DELETE statement, you must use an SQLJ *implements clause* to implement the `sqlj.runtime.ForUpdate` interface. You must also declare the iterator as public. For example, suppose that you declare the iterator ByPos for use in a positioned DELETE statement. The declaration looks like this:

```
#sql public iterator ByPos implements sqlj.runtime.ForUpdate
  with(updateColumns="EmpNo") (String);
```

Because you declare the iterator as public but not static, you need to use the iterator in a different source file. To use the iterator:

1. Import the generated iterator class.
2. Declare an instance of the generated iterator class.
3. Assign the SELECT statement for the positioned UPDATE or DELETE to the iterator instance.
4. Execute positioned UPDATE or DELETE statements using the iterator.

After the iterator is created, 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 is the authorization ID under which the DB2 package that contains the UPDATE or DELETE executes.

For example, suppose that you declare iterator UpdByName like this in UpdByName.sqlj:

```
#sql public iterator UpdByName implements sqlj.runtime.ForUpdate
  with(updateColumns="SALARY") (String EMPNO, BigDecimal SALARY);
```

To use UpdByName for a positioned UPDATE in another file, execute statements like those in Figure 5.

```
import UpdByName;                                                           1
{
  UpdByName upditer;         // Declare object of UpdByName class
  String enum;
  #sql upditer = { SELECT EMPNO, SALARY FROM EMP                            2
                     WHERE WORKDEPT='D11'};
  while (upditer.next())                                                    3
  {
    enum = upditer.EmpNo();  // Get value from result table
    #sql { UPDATE EMP SET SALARY=SALARY*1.05 WHERE CURRENT OF :upditer };  4
                             // Update row where cursor is positioned
    System.out.println("Updating row for " + enum);
  }
  #sql {COMMIT};             // Commit the changes
}
```

Figure 5. Updating rows using a positioned iterator

Notes to Figure 5:

1 This statement imports named iterator class UpdByName, which was created by the iterator declaration clause for UpdByName in UpdByName.sqlj. The import command is not needed if UpdByName is in the same package as the Java source file that references it.

2 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 upditer.

3 This statement positions the iterator to the next row to be updated.

4 This SQLJ clause performs the positioned UPDATE.

Using JDBC result sets in SQLJ applications

You can combine SQLJ clauses and JDBC calls in a single program to take advantage of the flexibility of JDBC and the type checking of SQLJ. To do this effectively, you need to be able to use SQLJ iterators to retrieve data from JDBC result sets or generate JDBC result sets from SQLJ iterators.

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 and must be declared in a separate file from
the file in which it is used.

- 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.

In a JDBC program, when you execute the `close` method on a `ResultSet` object that is
generated from a `Statement` object, you free the `ResultSet` object, as well as the
underlying DB2 cursor resources. However, when you close an iterator that is
generated from a `ResultSet` object, you free only the underlying DB2 cursor resources.
To make the iterator and the `ResultSet` object available for reuse, you need to close the
`PreparedStatement` object from which the `ResultSet` object is generated.

The code in Figure 6 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.

```
public void hireDates(Connection conn, String whereClause)
{
  #sql iterator ByName(String LastName, Date HireDate);          1
  ByName nameiter;           // Declare object of ByName class
  PreparedStatement stmt = conn.prepareStatement();              2
  String query = "SELECT LASTNAME, HIREDATE FROM EMP";
  query+=whereClause;   // Build the query
  ResultSet rs = stmt.executeQuery(query);                       3
  #sql nameiter = {CAST :rs};                                    4
  while (nameiter.next())
  {
    System.out.println( nameiter.LastName() + " was hired on "
      + nameiter.HireDate());

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

Figure 6. Converting a JDBC result set to an SQLJ iterator

Notes to Figure 6:

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 result set RS to SQLJ iterator nameiter, and the following statements use nameiter to retrieve values from the result table.

5 The close() method closes the SQLJ iterator and JDBC result set rs.

Generating JDBC result sets from SQLJ iterators

Use the getResultSet method to generate a JDBC result set 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 7 generates a positioned iterator for a query, converts the iterator to a result set, and uses JDBC methods to fetch rows from the table.

```
{
  sqlj.runtime.ResultSetIterator unTyped;
  #sql unTyped = { SELECT LASTNAME, HIREDATE FROM EMP };      1
  ResultSet rs = unTyped.getResultSet();                      2
  while (rs.next())                                           3
  { System.out.println(rs.getString(1) + " was hired in " +
      rs.getDate(2));
  }
  unTyped.close();                                            4
}
```

Figure 7. Converting an SQLJ iterator to a JDBC result set

Notes to Figure 7:

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 unTyped.

2 The getResultSet() method converts iterator unTyped to result set rs.

3 The JDBC getString() and getDate() methods retrieve values from the result set. The next() method moves the cursor to the next row in the result set.

4 The close() method closes the SQLJ iterator.

Controlling the execution of SQL statements

You can use selected methods of the SQLJ `ExecutionContext` class to query and modify the characteristics of SQL statements during execution. “Appendix A. Selected sqlj.runtime classes and interfaces” on page 109 describes those methods.

To execute `ExecutionContext` methods for an SQL statement, you must create an *execution context* and associate that execution context with the SQL statement.

To create an execution context, invoke the constructor for `ExecutionContext` and assign the result to a variable of type `ExecutionContext`. For example:

```
ExecutionContext ExecCtx=new ExecutionContext();
```

To associate an execution context with an SQL statement, specify the name of the execution context, enclosed in square brackets, at the beginning of the execution clause that contains the SQL statement. For example:

```
#sql [ExecCtx] {DELETE FROM EMP WHERE SALARY > 10000};
```

You can associate a different execution context with each SQL statement. If you also use an explicit connection context for an SQL statement, specify the connection context, followed by the execution context in the execution clause for the SQL statement. For example:

```
#sql [ConnCtx, ExecCtx] {DELETE FROM EMP WHERE SALARY > 10000};
```

If you do not specify an execution context for an execution clause, SQLJ uses the execution context that is associated with the connection context for the execution clause.

After you associate an execution context with an SQL statement, you can execute ExecutionContext methods for that SQL statement. For example, you can use method `getUpdateCount` to count the number of rows that are deleted by a DELETE statement:

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

Retrieving multiple result sets from a stored procedure

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:

- Create an execution context that is used to retrieve the result set from the stored procedure.

  If you plan to cast the result set from the stored procedure to an SQLJ iterator, create a second execution context for that purpose. You cannot use the same execution context to retrieve a result set and to cast that result set to an iterator.
- Associate the execution context with the CALL statement for the stored procedure.
- For each result set:
  - Use the ExecutionContext method getNextResultSet to retrieve the result set.
  - Use an iterator or JDBC ResultSet to retrieve the rows from the result set.

Each call to getNextResultSet closes the previous result set and advances to the next 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.

The code in Figure 8 on page 32 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.

```
#sql context ConnCtx;                                        1
Connection Connjdbc=
  DriverManager.getConnection("jdbc:db2os390sqlj:SANJOSE");
Connjdbc.setAutoCommit(false);
ConnCtx myconn=new ConnCtx(Connjdbc);
#sql [myconn] {CALL MULTRSSP()};                             2
ExecutionContext ExecCtx=myconn.getExecutionContext();       3
ResultSet rs;                                                4
while ((rs = ExecCtx.getNextResultSet()) != null)            5
{
  ResultSetMetaData rsmeta=rs.getMetaData();                 6
  int numcols=rsmeta.getColumnCount();
  while (rs.next())                                          7
  {
    for (int i=1; i<=numcols; i++)
    {
      String colval=rs.getString(i);
      System.out.println("Column " + i + "value is " + colval);
    }
  }
  rs.close();
}
```

Figure 8. Retrieving multiple result sets from a stored procedure

Notes to Figure 8:

1	This statement and the following three statements set the connection context for the program that calls the stored procedure.
2	MULTRSSP is a stored procedure that returns multiple result sets.
3	This statement gets the execution context from the connection that is used to call the stored procedure.
4	Result set rs is used to retrieve rows from each result set that is returned from the stored procedure.
5	Each invocation of the getNextResultSet method returns a result set from the stored procedure. When there are no more result sets to retrieve, getNextResultSet returns null.
6	Because the caller does not know the contents of the result sets that are returned from the stored procedure, JDBC ResultSetMetaData methods are used to obtain this information.
7	The statements in this loop retrieve rows from a result set and print out the contents of each column.

Setting the isolation level for a transaction

To set the isolation level for a unit of work within an SQLJ program, use the SET TRANSACTION ISOLATION LEVEL clause. Table 2 on page 33 shows the values that you can specify in the SET TRANSACTION ISOLATION LEVEL clause and their DB2 for OS/390 and z/OS equivalents.

Table 2. Equivalent SQLJ and DB2 isolation levels

SET TRANSACTION value	DB2 for OS/390 and z/OS isolation level
READ COMMITTED	Cursor stability
READ UNCOMMITTED	Uncommitted read
REPEATABLE READ	Read stability
SERIALIZABLE	Repeatable read

You can set the isolation level only at the beginning of a transaction.

Setting the read-only mode for a transaction

To set the read-only mode for a unit of work within an SQLJ program, use the SET TRANSACTION READ ONLY or SET TRANSACTION READ WRITE clause. SET TRANSACTION READ ONLY puts a connection into read-only mode so that DB2 can optimize execution of SQL statements for read-only access. If you execute SET TRANSACTION READ WRITE, DB2 does not optimize for read-only access.

You can set the read-only mode only at the beginning of a transaction.

An SQLJ sample program

Figure 9 on page 34 contains an example of an SQLJ program that prints the names and salaries of employees with salaries that exceed the average for the company. The program uses the DB2 sample employee table.

For another sample SQLJ program, see sample02.sqlj. Assuming the SQLJ/JDBC driver is installed in `/usr/lpp/db2`, you can find source code for the sample SQLJ program in the following path:

```
/usr/lpp/db2/db2710/samples/sample02.sqlj
```

For additional information on preparing and running sample program sample02, see sample02.readme in the same directory.

```
import sqlj.runtime.*;                                                  1
import java.sql.*;
import java.math.*;
#sql context HSCtx;                                                     2
#sql iterator HSByName(String LastName, BigDecimal Salary);             3
public class HighSalary
{
  public static void main (String[] args) // Main entry point
    throws SQLException
  {
    try {
      Class.forName("COM.ibm.db2os390.sqlj.jdbc.DB2SQLJDriver");        4
    }
    catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
    String userid="myid";
    String password="mypass";
    Connection HSjdbccon=                                               5
    DriverManager.getConnection("jdbc:db2os390sqlj:SANJOSE",userid,password);
    HSjdbccon.setAutoCommit(false);
    HSCtx myconn=new HSCtx(HSjdbccon);
    BigDecimal AvgSal;
    #sql [myconn] {SELECT AVG(SALARY) INTO :AvgSal FROM EMP};
    printSalary(AvgSal,myconn);
    HSjdbccon.close();
  }

  static void printSalary(BigDecimal AvgSalary, HSCtx hsconn)
    throws SQLException                   // Method to get high salaries
  {
    HSByName nameiter;                                                  6
    #sql [hsconn] nameiter =                                            7
      {SELECT LASTNAME, SALARY FROM EMP
         WHERE SALARY >= :AvgSalary
         ORDER BY SALARY DESC};
    while (nameiter.next())                                             8
      System.out.println( nameiter.LastName() + " " +                   9
        nameiter.Salary());
    nameiter.close();                                                   10
  }
}
```

Figure 9. SQLJ sample program

Notes to Figure 9:

1	The first two statements import the JDBC and SQLJ packages that are used by SQLJ.
2	This connection declaration clause declares connection context HSCtx, which will be used to connect to location SANJOSE. When you prepare the application program, SQLJ generates a class named HSCtx. You must therefore ensure that HSCtx is a valid Java class name that is unique within its scope.

3 This iterator declaration clause declares named iterator HSByName, which will be used to select rows from the employee table. When you prepare the application program, SQLJ generates a class named HSByName. You must therefore ensure that HSByName is a valid Java class name that is unique within its scope.

4 The `Class.ForName` method loads the DB2 for OS/390 and z/OS SQLJ JDBC driver and registers it with the `DriverManager`.

5 This statement and the two statements that follow it set up the connection to the data source at location SANJOSE for the user ID specified by userid with a password specified by password, and set autoCommit for the connection to off. Executable clauses that specify the connection instance myconn will be executed at location SANJOSE.

6 This statement declares nameiter as an instance of the named iterator class HSByName.

7 This assignment 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 nameiter.

8 `next`, which is a method of the generated class HSByName, advances the iterator to successive rows of the result set. `next` returns a value of `true` when a next row is available and a value of `false` when all rows have been fetched from the iterator.

9 Accessor methods nameiter.LastName and nameiter.Salary retrieve the values of the LASTNAME and SALARY column from the current row of the result table.

10 `close`, which is a method of generated iterator class HSByName, closes the iterator to free any database resources that the iterator holds.

Running SQLJ programs

After you have set the environmental variables discussed in “Setting environment variables” on page 91 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:

- 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 SQLJ program, execute the `java` command from the OS/390 UNIX System Services command line:

```
java program-name
```

Diagnosing SQLJ problems

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 SQLCODEs and SQLSTATEs. See “Handling SQL errors and warnings” on page 17 for information on how to write your application program to retrieve SQLCODEs and SQLSTATEs.

All SQLSTATEs except FFFFF are documented in Part 1 of *DB2 Messages and Codes*. FFFFF is a special SQLSTATE that indicates an internal error in the SQLJ/JDBC driver.

- 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 three 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

Before you can format SQLJ trace data, you must set several environmental variables. You must also set several parameters in the run-time properties file that you name in environmental variable DB2SQLJPROPERTIES. “Customizing parameters in the SQLJ/JDBC run-time properties file” on page 92 describes these variables and parameters.

In the CICS environment, configuring for traces is somewhat different than in other environments. See “Appendix B. Special considerations for CICS applications” on page 113 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 SQLJ/JDBC driver 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:

```
►►──db2sqljtrace──┬─fmt─┬──input-file-name──────────────────────────────────►◄
                  └─flw─┘
```

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 environmental variable DB2SQLJ_TRACE_FILENAME.

Running diagnosis utilities

If an SQLJ application program receives a recoverable, internal error (SQLSTATE FFFFF) or a repeatable, non-recoverable error, run diagnosis utilities `profp`, `profdb`, and `db2profp`, which are provided with SQLJ, to obtain additional information about the error.

The `profp` utility captures information about each SQLJ clause in a serialized profile. The format of the `profdb` utility is:

```
►►──profp──serialized-profile-name─────────────────────────────────────►◄
```

Run the `profp` utility on the serialized profile for the connection in which the error occurs. If an exception is thrown, a Java stack trace is generated. You can determine which serialized profile was in use when the exception was thrown from the stack trace.

The `db2profp` utility captures information about each SQLJ clause in a customized serialized profile. A customized serialized profile is a serialized profile on which the DB2 for OS/390 and z/OS SQLJ customizer has been run. The format of the `db2profp` utility is:

```
►►──db2profp──customized-serialized-profile-name──────────────────────►◄
```

Run the `db2profp` utility on the customized serialized profile for the connection in which the error occurs.

The `profdb` utility customizes serialized profiles so that SQLJ captures extra information about run-time calls. The syntax of the `profdb` utility is:

```
                ┌─────────────────────────┐
                ▼                         │
►►──profdb──────serialized-profile-name───┴────────────────────────────►◄
```

Run the `profdb` utility on every serialized profile that is associated with the SQLJ application program that received the internal error. After you run `profdb`, rerun the application program to gather the diagnostic information.

Chapter 3. SQLJ statement reference

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

```
►►──#sql──┬─connection-declaration-clause─┬─;──────────────────────────────────────►◄
          ├─iterator-declaration-clause───┤
          └─executable-clause─────────────┘
```

This chapter describes each of the three clauses that can appear in an SQLJ clause and the elements that you can include in each of those clauses. Elements that are subcomponents of several other elements are discussed first.

For more information and examples of using the clauses described in this chapter, see “Chapter 2. Writing SQLJ programs for DB2 for OS/390 and z/OS” on page 15.

Common elements

This section describes the elements that are common to several SQLJ clauses.

host-expression

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

Syntax

```
►►──:──┬───────┬──┬─simple-variable──────┬──────────────────────────────────────►◄
       ├─IN────┤  └─(complex-expression)─┘
       ├─OUT───┤
       └─INOUT─┘
```

Description

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

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

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.
- 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.

implements-clause

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

Syntax

```
              ┌─,──────────────┐
              ▼                │
►►──implements───interface-element─┴──────────────────────────────────────►◄
```

interface-element:

```
►►──┬─sqlj.runtime.ForUpdate───────────┬─────────────────────────────────►◄
    └─user-specified-interface-class─┘
```

Description

interface-element
: Specifies a user-defined Java interface, or the SQLJ interface `sqlj.runtime.ForUpdate`.

  You must implement `sqlj.runtime.ForUpdate` when you declare an iterator for a positioned UPDATE or positioned DELETE operation. See "Using iterators for positioned UPDATE and DELETE operations" on page 27 for information on performing a positioned UPDATE or positioned DELETE operation in SQLJ.

with-clause

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

Syntax

```
                ┌─,────────────┐
                ↓              │
►►──with──(──────with-element──┴──)──────────────────────────────────────────────►◄
```

with-element:

```
►►──┬─holdability=──┬─true──┬─────────────────┬──────────────────────────────────►◄
    │               └─false─┘                 │
    ├─returnability=──┬─true──┬───────────────┤
    │                 └─false─┘               │
    │                 ┌─,──────────────┐      │
    │                 ↓                │      │
    ├─updateColumns=─"───column-name───┴──"───┤
    └─Java-ID=Java-constant-expression────────┘
```

Description

holdability
: Specifies whether an iterator keeps its position in a table after a COMMIT is executed. The value for holdability must be true or false.

returnability
: Specifies whether an iterator can return result sets from a stored procedure call. The value for returnability must be true or false.

updateColumns
: 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
: Specifies a column of the result table that is to be updated using the iterator.

Java-ID
: Specifies a Java variable that identifies a user-defined attribute of an iterator or connection context. The value of *Java-constant-expression* is also user-defined.

Usage notes

- The value on the left side of a with element must be unique within its with clause.
- For a connection declaration clause, only user-defined attributes (*Java-ID=Java-constant-expression*) can be specified in a 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.

connection-declaration-clause

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

Syntax

```
►►──┬────────────────┬──context──Java-class-name──┬───────────────────┬──┬─────────────┬──────────►◄
    └─Java-modifiers─┘                            └─implements-clause─┘  └─with-clause─┘
```

Description

Java-modifiers
Specifies modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
Specifies a valid Java identifier. During the program preparation process, SQLJ generates a connection context class whose name is this identifier.

implements-clause
See “implements-clause” on page 41 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 “with-clause” on page 42 for a description of this clause. In a connection declaration clause, all attributes in a with clause must be user defined.

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.

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. See “Using result set iterators to retrieve rows from a result table” on page 23 for information on how to use each type of iterator.

Syntax

```
►►─┬────────────────┬─iterator─Java-class-name─┬────────────────────┬─┬─────────────┬──────────►
   └─Java-modifiers─┘                          └─implements-clause─┘ └─with-clause─┘

►─(─┬─positioned-iterator-column-declarations─┬─)─────────────────────────────────────────────►◄
    └─named-iterator-column-declarations──────┘
```

positioned-iterator-column declarations:

```
     ┌─,──────────────┐
     ▼                │
►►───Java-data-type───┴───────────────────────────────────────────────────────►◄
```

named-iterator-column-declarations:

```
     ┌─,─────────────────────────┐
     ▼                           │
►►───Java-data-type──Java-ID─────┴────────────────────────────────────────────►◄
```

Description

Java-modifiers
: Any modifiers that are valid for Java class declarations, such as static, public, private, or protected.

Java-class-name
: Any valid Java identifier. During the program preparation process, SQLJ generates an iterator class whose name is this identifier.

implements-clause
See "implements-clause" on page 41 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`.

with-clause
See "with-clause" on page 42 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 Table 1 on page 23 for a list of compatible data types. A positioned iterator can be used only for FETCH statements.

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 Table 1 on page 23 for a list of compatible data types. A named iterator cannot be used for a FETCH statement.

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.

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 section first describes the executable clause in general. The next sections describe each of the components of an executable clause.

Syntax

```
►►──┬────────────────┬──┬─statement-clause──┬──────────────────────────────────────►◄
    └─context-clause─┘  └─assignment-clause─┘
```

Usage notes

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

  If SQLJ raises a run-time exception during the execution of an executable clause, the value of any host expression of type OUT or INOUT is undefined.

context-clause

A context clause specifies a connection context or an execution context. You use a connection context to connect to a data source. You use an execution context to monitor and modify SQL statement execution. See “Connecting to a data source” on page 19 for information on using a connection context. See “Controlling the execution of SQL statements” on page 30 for information on using an execution context.

Syntax

```
►►──[──┬─connection-context──────────────────────────┬──]──────────────────────────►◄
       ├─execution-context───────────────────────────┤
       └─connection-context──,──execution context────┘
```

Description

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

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

Usage notes

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

statement-clause

A statement clause contains an SQL statement or a SET TRANSACTION clause. All SQL statements are described in Chapter 5 of *DB2 SQL Reference*. The SET TRANSACTION clause is described in “SET-TRANSACTION-clause” on page 51.

Syntax

```
►►─{──┬─SQL-statement──────────┬─}──────────────────────────────────────────►◄
      └─SET-TRANSACTION-clause─┘
```

Description

SQL-statement
You can include the statements in Table 3 in a statement clause.

SET-TRANSACTION-clause
Sets the isolation level for SQL statements in the program and the access mode for the connection. The SET TRANSACTION clause is equivalent to the SET TRANSACTION statement, which is described in the ANSI/ISO SQL standard of 1992 and is supported in some implementations of SQL. See “SET-TRANSACTION-clause” on page 51 for more information.

Table 3. Valid SQL statements in an SQLJ statement clause

ALTER DATABASE
ALTER FUNCTION
ALTER INDEX
ALTER PROCEDURE
ALTER STOGROUP
ALTER TABLE
ALTER TABLESPACE
CALL
COMMIT
CREATE ALIAS
CREATE DATABASE
CREATE DISTINCT TYPE

Table 3. Valid SQL statements in an SQLJ statement clause (continued)

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
EXPLAIN
FETCH
GRANT
INSERT
LOCK TABLE
RENAME
REVOKE
ROLLBACK
SELECT
SET CURRENT DEGREE
SET CURRENT LOCALE LC_CTYPE
SET CURRENT OPTIMIZATION HINT
SET CURRENT PRECISION
SET CURRENT RULES
SET CURRENT SQLID
SIGNAL SQLSTATE
UPDATE
VALUES

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 obtain rows from a result table. See “Using result set iterators to retrieve rows from a result table” on page 23 for more information on iterators.

assignment-clause

The assignment clause assigns the result table from a SELECT statement to an iterator.

Syntax

```
▶▶──Java-ID=──{──┬─subselect──────────────────┬─}──────────────────────────────────────▶◀
                 └─iterator-conversion-clause─┘
```

Description

Java-ID
Identifies an iterator that was declared previously as an instance of an iterator class.

subselect
Generates a result table. The syntax of the subselect is defined in Chapter 4 of *DB2 SQL Reference*.

iterator-conversion-clause
See “iterator-conversion-clause” for a description of this clause.

Usage notes

- 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.
- 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.

iterator-conversion-clause

The iterator conversion clause converts a JDBC result set to an iterator.

Syntax

```
▸▸──CAST──host-expression──────────────────────────────────────────────────────▸◂
```

Description

host-expression
Identifies the JDBC result set that is to be converted to an SQLJ iterator.

Usage notes

- 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.
- When an iterator that is generated through the iterator conversion clause is closed, the result set from which the iterator is generated is also closed.

SET-TRANSACTION-clause

The SET TRANSACTION clause performs one of the following functions:

- Sets the isolation level for the current unit of work. For a detailed discussion of isolation levels, see Part 5 (Volume 2) of *DB2 Administration Guide*.
- Sets or disables read-only mode for a connection.

Syntax

```
▸▸──SET TRANSACTION──┬─ISOLATION LEVEL──┬─READ COMMITTED───┬─┬──────────────────────────▸◂
                     │                  ├─READ UNCOMMITTED─┤ │
                     │                  ├─REPEATABLE READ──┤ │
                     │                  └─SERIALIZABLE─────┘ │
                     ├─READ ONLY─────────────────────────────┤
                     └─READ WRITE────────────────────────────┘
```

Description

ISOLATION LEVEL
Specifies one of the following DB2 for OS/390 and z/OS 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.

READ ONLY
Set the connection object to read-only mode. Executing `SET TRANSACTION READ ONLY;` is equivalent to invoking the JDBC method *connection*`.setReadOnly(true);`.

READ WRITE
Set the connection object to read-write mode. Executing `SET TRANSACTION READ WRITE;` is equivalent to invoking the JDBC method *connection*`.setReadOnly(false);`.

Usage notes

You can execute SET TRANSACTION only at the beginning of a transaction.

Chapter 4. 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 chapter 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 "Defining a JAR file for a Java routine to DB2" on page 62.

In this chapter, the following terminology is used:

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

This chapter covers the following topics:

- "Setting up the environment for Java routines"
- "Defining a Java routine to DB2" on page 56
- "Defining a JAR file for a Java routine to DB2" on page 62
- "Writing a Java routine" on page 64
- "Testing a Java routine" on page 66

Setting up the environment for Java routines

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

1. Ensure that your operating system and Java are at the correct levels. ″See Program requirements and optional programs″ in *DB2 Release Planning Guide* for a list of requirements.
2. Set up a WLM-managed address space for running the routines. See "The stored procedure address space for Java routines".
3. Set environment variables that are required by Java routines. See "Setting environment variables for Java routines" on page 54.

The stored procedure address space for Java routines

A Java routine must run in a WLM-established stored procedure address space. You need to run Java stored procedures and user-defined functions in a different WLM environment from other stored procedures and user-defined functions. You also need to run compiled Java stored procedures in a different WLM environment from interpreted Java routines. The address space startup procedure for Java routines requires extra DD statements that other routines do not need. Figure 10 on page 54 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. If the JAVAENV data set contains the environment variable JAVA_HOME, it indicates to DB2 that the WLM environment is for interpreted Java routines.

```
  //DSNWLM   PROC RGN=0K,APPLENV=WLMCJAV,DB2SSN=DSN,NUMTCB=1
  //IEFPROC EXEC PGM=DSNX9WLM,REGION=&RGN,TIME=NOLIMIT,
  //        PARM='&DB2SSN,&NUMTCB,&APPLENV'
  //STEPLIB  DD  DISP=SHR,DSN=DSN710.RUNLIB.LOAD
1 //         DD  DISP=SHR,DSN=HPJSP.PDSE1
  //         DD  DISP=SHR,DSN=CEE.SCEERUN
  //         DD  DISP=SHR,DSN=DSN710.SDSNEXIT
  //         DD  DISP=SHR,DSN=DSN710.SDSNLOAD
2 //         DD  DISP=SHR,DSN=HPJ.SQLJ,DISP=SHR
3 //         DD  DISP=SHR,DSN=HPJ.SHPJMOD
  //         DD  DISP=SHR,DSN=HPJ.SHPOMOD
  //         DD  DISP=SHR,DSN=VAJAVA.V2R0M0.SHPOMOD
4 //JAVAENV  DD  DISP=SHR,DSN=WLMCJAV.JSPENV
5 //JSPDEBUG DD  SYSOUT=A
  //CEEDUMP  DD  SYSOUT=A
  //SYSPRINT DD  SYSOUT=A
```

Figure 10. Startup procedure for a WLM address space in which a Java routine runs

Notes to Figure 10:

1 This DD statement specifies the PDSE that contains the Java program objects for compiled Java stored procedures.

2 This DD statement specifies the PDSE that contains Java program objects for Java classes that are referenced by the stored procedure.

3 This DD statement and the following DD statement specify the PDSEs that contains the VisualAge for Java compiler and run-time library.

4 JAVAENV specifies a data set that contains environment variables that specify system properties for the Java routine execution environment. The presence of this DD statement indicates to DB2 that the WLM environment is for Java routines. See “Setting environment variables for Java routines” for more information.

5 JSPDEBUG specifies a data set into which DB2 puts information that you can use to debug your stored procedure.

If you specify SYSOUT=A in the JSPDEBUG DD statement, the debug information is written to your SYSOUT data set. If you specify a data set name, you also need to specify the WORK_DIR environment variable in your JAVAENV data set. If you specify the WORK_DIR environment variables, DB2 creates files server_stdin.txt, server_stdout.txt, and server_stderr.txt, which contain debugging information.

Setting environment variables for Java routines

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 environment variables for the VisualAge for Java execution environment. You need to set the following environment variables in the JAVAENV data set:

CLASSPATH
Modify CLASSPATH to include the following HFS directories:

- The directories that contain the external links to the routines that run in the WLM-established stored procedure address space
- The directory in which the VisualAge for Java links are defined ($IBMHPJ_HOME/lib)

For example:

```
CLASSPATH=.:/u/sysadm/links:/usr/lpp/hpj/lib
```

LIBPATH and LD_LIBRARY_PATH
Modify LIBPATH and LD_LIBRARY_PATH to include the following HFS directories:

- The path for the ET/390 code ($IBMHPJ_HOME/lib)
- The path for the SQLJ/JDBC driver code, if the stored procedures contain SQL

For example:

```
LIBPATH=/u/sysadm/links:/usr/lpp/hpj/lib:/u/hpjsp/lib
LD_LIBRARY_PATH=/u/sysadm/links:/usr/lpp/hpj/lib:/u/hpjsp/lib
```

JAVA_HOME
For an interpreted Java routine, used to determine Java.home. For example:

```
JAVA_HOME=/usr/local/Java
```

DB2 uses this environment variable to determine whether the WLM environment is for a compiled Java stored procedure or an interpreted Java routine. Therefore, if the startup procedure is for a WLM environment for interpreted Java routines, you need to include this environment variable.

LANG
Modify LANG to change the locale to use for the locale categories when neither the LC_ALL environment variable nor the individual locale environment variables specify locale information. For example:

```
LANG="En_US.IBM-037"
```

LC_ALL
Modify LC_ALL to change the locale to be used to override any values for locale categories specified by the settings of the LANG environment variable or any individual locale environment variables. For example:

```
LC_ALL="S370"
```

LC_CTYPE
Modify LC_CTYPE to change the locale for character classification, case conversion, and other character attributes. This value should match the DB2 for OS/390 and z/OS installation default. For example:

```
LC_CTYPE="En_US.IBM-037"
```

WORK_DIR
Specify WORK_DIR to direct diagnostic output to an HFS directory. When you specify the JSPDEBUG DD statement in your WLM address space startup

procedure, DB2 creates three data sets in this directory for diagnostic information. Those data sets are server_stdin.txt, server_stdout.txt, and server_stderr.txt.

TZ Modify TZ to change the local timezone. For example:

```
TZ="PST08"
```

The default is GMT.

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

```
ENVAR("CLASSPATH=.:/u/sysadm/links:/usr/lpp/hpj/lib",
 "TZ=PST08",
 "LIBPATH=/u/sysadm/links:/usr/lpp/hpj/lib:/u/hpjsp/lib",
 "LD_LIBRARY_PATH=/u/sysadm/links:/usr/lpp/hpj/lib:/u/hpjsp/lib"),
 MSGFILE(JSPDEBUG)
```

For information on environment variables that are related to locales, see *OS/390 C/C++ Programming Guide*.

Defining a Java routine to DB2

Defining a Java routine to DB2 involves one or two steps, depending on how you prepare the routine for execution:

- 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 IBM DB2 Stored Procedure Builder, 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 62.
- For all types of Java routines, you need to define the routine to DB2.

  If you prepare the Java routine for execution without IBM DB2 Stored Procedure Builder, 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.

If you use the IBM DB2 Stored Procedure Builder to prepare your Java routine for execution, the IBM DB2 Stored Procedure Builder 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.

There are two possible values for a Java stored procedure. If the stored procedure runs in a JVM, specify LANGUAGE JAVA. If the stored procedure runs under VisualAge for Java, specify LANGUAGE COMPJAVA.

For a Java user-defined function, which must run in a JVM, 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 value in that string depends on the value of LANGUAGE:

- For LANGUAGE COMPJAVA, the syntax of this parameter is:

```
►►──EXTERNAL──┬──────┬──'──┬─────────────────────┬──class-name.method-name──┬──────────────────────┬──'──────►◄
              └─NAME─┘     │                 (1) │                          └─(method-signature)──┘
                           └─package-name.───────┘
```

Notes:

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

For example if you write a Java stored procedure as method getSals in class S1Sal and package s1, the EXTERNAL NAME parameter is:

```
's1.S1Sal.getSals'
```

- For LANGUAGE JAVA, the syntax of this parameter is:

```
►►──EXTERNAL──┬──────┬────────────────────────────────────────────────────────────────────────────────────►
              └─NAME─┘

►─'──┬────────────────┬──┬─────────────────────┬──class-name.method-name──┬──────────────────────┬──'──────►◄
     │            (1) │  │                 (2) │                          └─(method-signature)──┘
     └─JAR-name:──────┘  └─package-name──.─────┘
```

Notes:

1 For compatibility with DB2 UDB for Unix, Windows, OS/2, 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 the way that you prepare your stored procedure for execution. 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, define the JAR file to DB2, and grant the USAGE privilege on the JAR before you execute the CREATE PROCEDURE or CREATE FUNCTION statement.

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 part of the Java environment, such as java.lang package or the SQLJ/JDBC driver classes do not need to be 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 programs 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 a class that is declared as an SQLJ iterator

Table 4 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.

Table 4. Valid Java data types for the method signature in a CREATE PROCEDURE or CREATE FUNCTION statement

SQL type in parameter definition[1]	**Java type in method signature**[2]
CHAR(*n*)	java.lang.String
VARCHAR(*n*)	java.lang.String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
SMALLINT	short, java.lang.Integer
INTEGER	int, java.lang.Integer
REAL	float, java.lang.Float
FLOAT	double, java.lang.Double
DOUBLE PRECISION	double, java.lang.Double
CHAR(*n*) FOR BIT DATA	byte[]
VARCHAR(*n*) FOR BIT DATA	byte[]

Table 4. Valid Java data types for the method signature in a CREATE PROCEDURE or CREATE FUNCTION statement (continued)

SQL type in parameter definition[1]	Java type in method signature[2]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Note:

1. For a stored procedure, if a parameter is declared as INOUT or OUT, the associated Java type must be a single-dimension array of the specified type.
2. The Java type name needs to be the full name, which includes the package name and class name.

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.

A Java routine must run in a WLM-established address space, so this parameter must be specified.

PROGRAM TYPE

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

You can write a Java routine as a Java main method. However, 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 or LANGUAGE COMPJAVA, 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 call 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 call 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 OS/390 UNIX System Services. The values of SECURITY and the IDs that must have access to OS/390 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 that runs under VisualAge for Java: 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
Run-time environment	VisualAge for Java
Collection ID for the stored procedure package	DSNJDBC
Java executable name	s1.S1Sal.getSals
Type of SQL statements in the program	Statements that modify DB2 tables
WLM environment name	WLMENV1
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 COMPJAVA
  EXTERNAL NAME  's1.S1Sal.getSals'
  WLM ENVIRONMENT WLMENV1
  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
Output parameter	VARCHAR(20)
Language	Java
Run-time environment	JVM
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	WLMENV1

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 WLMENV1
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. If you do this, you need to install the JAR file into the DB2 catalog. DB2 provides three built-in stored procedures that perform the following functions for the JAR file:

SQLJ.INSTALL_JAR
Installs a JAR file into the DB2 catalog.

SQLJ.REPLACE_JAR
Replaces an existing JAR file in the DB2 catalog.

SQLJ.REMOVE_JAR
Deletes a JAR file from the DB2 catalog.

You can use the IBM DB2 Stored Procedure Builder to install JAR files into the DB2 catalog, or you can write a client program that executes SQL CALL statements to invoke the SQLJ.INSTALL_JAR, SQLJ.REPLACE_JAR, and SQLJ.REMOVE_JAR stored procedures. The following sections explain 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 DB2 catalog.

SQLJ.INSTALL_JAR syntax

```
►►──CALL──SQLJ.INSTALL_JAR──(──url,──jar-name,──deploy──)──────────────────────────►◄
```

SQLJ.INSTALL_JAR parameters

url A VARCHAR(128) input parameter that identifies the HFS 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(27) 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 default schema name.

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 DB2 catalog.

SQLJ.REPLACE_JAR syntax

```
►►──CALL──SQLJ.REPLACE_JAR──(──url,──jar-name──)──────────────────────────────────►◄
```

SQLJ.REPLACE_JAR parameters

url A VARCHAR(128) input parameter that identifies the HFS 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(27) 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 default schema name.

Calling SQLJ.REMOVE_JAR

Use SQLJ.REMOVE_JAR to delete a JAR file from the DB2 catalog.

SQLJ.REMOVE_JAR syntax

```
►►──CALL──SQLJ.REMOVE_JAR──(──jar-name,──undeploy──)──────────────────────────────►◄
```

SQLJ.REMOVE_JAR parameters

JAR-name
A VARCHAR(27) 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 default schema name.

undeploy
An INTEGER input parameter that indicates whether additional actions should be performed after the JAR file is removed. Additional actions are not supported, so this value should always be 0.

Writing a Java routine

A Java routine is a JDBC or SQLJ application program that runs in a stored procedure address space. 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:

- A Java routine does not establish its own connection to the local data source. Instead, the routine uses the default RRS connection to the data source that processes the CALL statement or the statement that contains the user-defined function invocation. If you want to execute SQLJ clauses or JDBC methods at another location, use the same methods to connect to those locations as you do in SQLJ or JDBC application programs.
- A Java routine must be declared as static and public.
- 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 B 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.
- Java main programs must have a signature of String[]. If your Java routine is a main method, it must be possible to map all the parameters to Java variables of type java.lang.String.
- You cannot make 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 Table 1 on page 23.
- 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 65 for information on how to cause a Java stored procedure to return result sets.

Writing a Java stored procedure to return result sets

Your stored procedure can return multiple query result sets to a DRDA client if the following conditions are satisfied:

- The client supports the DRDA code points that are used to return query result sets.
- The value of 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 a class that is declared as an SQLJ iterator 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. 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 the java.sql.ResultSet[] object that is in the parameter list.
- 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 11 on page 66 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.*;
1   #sql iterator NameSal(String LastName, BigDecimal Salary);
    public class S1Sal
    {
2     public static void getSals(BigDecimal[] AvgSalParm, java.sql.ResultSet[] rs)
        throws SQLException
      {
        NameSal iter1;
        try
        {
3         #sql iter1 = {SELECT LASTNAME, SALARY FROM EMP
            WHERE SALARY>0 ORDER BY SALARY DESC};
4         #sql {SELECT AVG(SALARY) INTO :(AvgSalParm[0]) FROM EMP};
        }
        catch (SQLException e)
        {
          System.out.println("SQLCODE returned: " + e.getErrorCode());
          throw(e);
        }
5       rs[0] = iter1.getResultSet();
      }
    }
```

Figure 11. Java stored procedure that returns a result set

Notes to Figure 11:

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.

When you are ready to invoke 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.

Chapter 5. Preparing Java programs

DB2 for OS/390 and z/OS Java programs run in the OS/390 UNIX System Services environment. These applications can run in a JVM or under VisualAge for Java. This chapter explains program preparation techniques for SQLJ programs and Java stored procedures. The following topics are discussed:

- “Steps in the SQLJ program preparation process”
- “Preparing Java routines for execution” on page 76
- “Building an SQLJ or JDBC program under VisualAge for Java” on page 80

Steps in the SQLJ program preparation process

After you write an SQLJ application, you must generate an executable form of the application. To prepare the application to run in a JVM, follow these steps:

1. Translate the source code to produce modified Java source code and serialized profiles, and compile the modified source code to product Java bytecodes.
2. Customize the serialized profiles to produce DBRMs.
3. Bind the DBRMs into packages and bind the packages into a plan, or bind the DBRMs directly into a plan.

Figure 12 on page 70 shows the steps of the program preparation process.

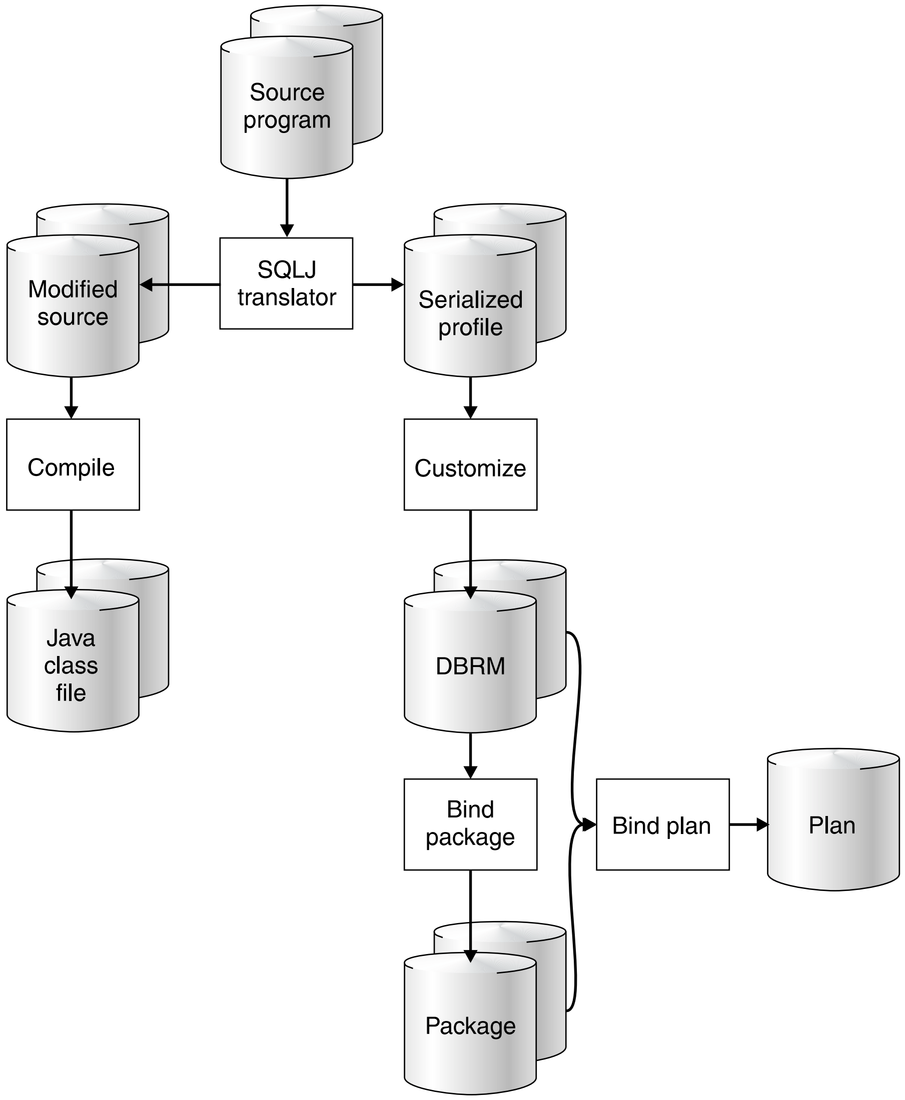


Figure 12. The SQLJ program preparation process

This section discusses each of those steps. See "Building an SQLJ or JDBC program under VisualAge for Java" on page 80 for information on preparing an application to run under VisualAge for Java.

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. Executing the `sqlj` command from the OS/390 UNIX

System Services command line invokes the DB2 for OS/390 and z/OS SQLJ translator. The SQLJ translator runs without connecting to DB2.

Syntax

```
►►──sqlj──┬───────┬──┬─────────────────┬──┬───────────────────────────┬──┬─────────────────────────┬──────────►
          └─-help─┘  └─-dir=directory──┘  └─-props=properties-file────┘  │              ┌─true──┐  │
                                                                         └─-compile=──┴─false─┴──┘

►──┬───────────────────────────┬──file-list──────────────────────────────────────────────────────────────────►◄
   │          ┌─all─────────┐  │
   └─-warn=───┼─none────────┼──┘
              ├─verbose─────┤
              ├─nonverbose──┤
              ├─portable────┤
              └─nonportable─┘
```

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 output from the translator. This output consists of Java source files and serialized profile files. The default directory is the current directory.

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. Then the translator puts the serialized profiles and Java source file for file1.sqlj in directory /src and puts the serialized profiles and Java source file 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 modified Java source into bytecodes.

true
The translator compiles the modified Java source code. This is the default.

false
The translator does not compile the modified 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.

Output from the SQLJ translator

For each source file, *program-name*.sqlj, the SQLJ translator produces the following files:

- The modified source program

  The modified source file is named *program-name*.java.
- A serialized profile file for each connection declaration clause in the program, and one serialized profile for the default context, if it is used

  A serialized profile file is named *program-name*_SJProfile*n*.ser, where *n* is 0 for the first serialized profile generated for the program, 1 for the second serialized profile generated, and so on.

  You must run the SQLJ customizer on each serialized profile file to produce a standard DB2 for OS/390 and z/OS DBRM. See “Customizing an SQLJ serialized profile” for information on how to customize a serialized profile.

Customizing an SQLJ serialized profile

After you use the SQLJ translator to generate serialized profiles for an SQLJ program, customize each serialized profile to produce a standard DB2 for OS/390 and z/OS DBRM and a serialized profile that is customized for DB2 for OS/390 and z/OS. Executing the `db2profc` on the OS/390 UNIX System Services command line customizes a serialized profile.

Syntax

```
►►──db2profc──┬───────┬──┬──────────┬──┬────────────────┬──┬────────────────┬──┬────────────────┬──────────►
              └─-help─┘  └─-version─┘  │       ┌─ISO─┐  │  │       ┌─ISO─┐  │  │      ┌─ALL─┐   │
                                       └─-date=─┼─USA─┼──┘  └─-time=─┼─USA─┼──┘  └─-sql=─┴─DB2─┴───┘
                                                ├─EUR─┤              ├─EUR─┤
                                                └─JIS─┘              └─JIS─┘

►──┬────────────────────────────────────────────────────────────────────────────────────────────────┬──────►
   └─-online=location-name──┬──────────────────────────────────────────────────────────────────────┤
                            │                                ┌─-inform──=──YES─┐  ┌─-validate=CUSTOMIZE─┐ │
                            └─┬────────────────────────────┬─┴─-inform──=──NO──┴──┴─-validate=RUN───────┴─┘
                              └─-schema=authorization-ID───┘

►──-pgmname=DBRM-member-name──serialized-profile-name───────────────────────────────────────────────────────►◄
```

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 for OS/390 and z/OS recognizes.

  `ALL`, which is the default, indicates that the SQL statements in the program are not necessarily for DB2 for OS/390 and z/OS. Use `ALL` for application programs whose SQL statements must execute on a server other that DB2 for OS/390 and z/OS.

  `DB2` indicates that the SQLJ customizer should interpret SQL statements and check syntax for use by DB2 for OS/390 and z/OS. Use `DB2` when the database server is DB2 for OS/390 and 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 95 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.

-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 a plan for an SQLJ program" on page 75 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. Serialized profiles are generated by the SQLJ translator and have names of the form

```
program-name_SJProfilen.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.

Usage notes
Online checking is always recommended: It is highly recommended that you use
online checking (the `-online` option) when you customize your serialized profiles.
Online checking determines information about the data types and lengths of DB2 host
variables and stores that information in the DBRM. Online checking is especially
important in the following situations:

• 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 making the length of each host variable available to DB2.

- 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

  The SQLJ/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.

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.

Output from the SQLJ customizer

When the SQLJ customizer runs, it creates a DBRM and a modified serialized profile.

Binding a plan for an SQLJ program

After you have customized the serialized profiles for your SQLJ application program, you must 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 5 shows the name of each DBRM and the isolation level that you need to specify when you bind that DBRM.

Table 5. 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)

DB2 executes SQLJ positioned update and delete operations dynamically and executes all other SQLJ statements statically. In addition, you might write an SQLJ program that includes JDBC methods. It is therefore recommended that you bind your SQLJ plans with option DYNAMICRULES(BIND). 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*.

Customizing SQLJ and JDBC to work together

With interoperability established, a Java application can contain both static and dynamic SQL. The application can execute SQLJ clauses and invoke JDBC methods.

Complete the following steps to establish interoperability:

- When you bind a plan for SQLJ, include JDBC packages in the PKLIST of that SQLJ plan. The default names of the JDBC packages are:
  - DSNJDBC.DSNJDBC1
  - DSNJDBC.DSNJDBC2
  - DSNJDBC.DSNJDBC3
  - DSNJDBC.DSNJDBC4
- Make sure that the JDBC profile is accessible in a directory specified in the CLASSPATH environment variable. “Customizing the JDBC profile (optional)” on page 95 explains how to create the JDBC profile.

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 that run in a JVM are referred to in this section as interpreted Java routines. Java stored procedures that run under VisualAge for Java are referred to as compiled Java stored procedures. This section explains how to prepare Java routines for execution.

See “Steps in the SQLJ program preparation process” on page 69 for detailed information on program preparation steps that are common to all Java programs. See “Defining a Java routine to DB2” on page 56 for information on defining Java routines and JAR files to DB2. See “Building an SQLJ or JDBC program under VisualAge for Java” on page 80 for detailed information about preparing programs to run under VisualAge for Java.

Preparing interpreted Java routines for execution

This section 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.
3. Call the INSTALL_JAR stored procedure to define the JAR file to DB2.

4. Execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR to the user who defines the routine to DB2.
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 on page 76.

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. Include the HFS directory that contains the class file for your routine in your CLASSPATH. You set the CLASSPATH by specifying the CLASSPATH environment variable 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.
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.

Procedure 3:

Use IBM DB2 Stored Procedure Builder 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 modified Java source code and serialized profiles, and to compile the Java program to produce Java bytecodes.
2. Run the `db2profc` command to produce serialized profiles that are customized for DB2 for OS/390 and z/OS and DBRMs.
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 into a JAR file.
4. Call the INSTALL_JAR stored procedure to define the JAR file to DB2.
5. Execute the SQL GRANT USAGE ON JAR statement to grant the privilege to use the JAR to the user who defines the routine to DB2.
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. Run the DB2 `BIND` command to bind the DBRMs that you created in step 2 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 modified Java source code and serialized profiles, and to compile the Java program to produce Java bytecodes.
2. Run the `db2profc` command to produce serialized profiles that are customized for DB2 for OS/390 and z/OS and DBRMs.
3. Include the HFS directory that contains the class file for your routine in your CLASSPATH. You set the CLASSPATH by specifying the CLASSPATH environment variable 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.
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. Run the DB2 `BIND` command to bind the DBRMs that you created in step 2 into packages.

Procedure 3:

Use IBM DB2 Stored Procedure Builder to perform all of the program preparation steps.

Preparing compiled Java stored procedures for execution

This section outlines the program preparation steps for compiled Java stored procedures. Those steps vary, depending on whether your routine contains embedded SQL statements.

Preparing compiled Java stored procedures with no SQLJ statements

If the program contains only JDBC methods or no SQL statements, use the following procedure for program preparation:

1. Run the `javac` command to compile the Java program to produce Java bytecodes.
2. Run the VisualAge for Java hpj command to bind the Java bytecode file for the stored procedure and for any packages that are used by the stored procedure into Java DLLs in PDSEs. See "Using VisualAge for Java to produce a Java DLL for a stored procedure" on page 79 for more information.
3. Execute the SQL CREATE PROCEDURE statement to define the stored procedure to DB2.

Preparing compiled Java stored procedures with SQLJ statements

If the program contains SQLJ clauses, the program preparation process includes these steps:

1. Run the `sqlj` command to translate and compile the source code. This process produces modified Java source code, serialized profiles, and Java bytecodes.
2. Run the `db2profc` command to produce serialized profiles that are customized for DB2 for OS/390 and z/OS and DBRMs.
3. Run the jar tool to create a JAR file for the serialized profiles.
4. Run the VisualAge for Java hpj command to bind the Java bytecode file for the stored procedure and for any packages that are used by the stored procedure into

Java DLLs in PDSEs. See “Using VisualAge for Java to produce a Java DLL for a stored procedure” for more information.

5. Execute the SQL CREATE PROCEDURE statement to define the stored procedure to DB2.
6. Bind the DBRMs into packages and plans, or directly into plans, using the DB2 `BIND` command.

Using VisualAge for Java to produce a Java DLL for a stored procedure

To convert Java bytecodes into a compiled Java program that runs as a stored procedure, you need to execute the hpj command under OS/390 UNIX System Services. You need to specify hpj options that create the Java DLLs in a PDSE and create an external link to the DLLs.

The input to the hpj command needs to include all classes that are used by the stored procedure. Those classes include:

- The class for the stored procedure program
- Generated classes for any iterators that are used by the stored procedure
- Internal classes that are generated by the SQLJ translator

You should also create JAR files from the `.ser` files that are generated by the SQLJ translator, and include those JAR files in your Java DLL as resources. See “Building an SQLJ or JDBC program under VisualAge for Java” on page 80 for more information on executing the hpj command.

Example: Create an executable Java stored procedure: Suppose that you have written the Java stored procedure shown in Figure 11 on page 66 and stored the source code in a file named s1sal.sqlj. Prepare the stored procedure for execution.

The steps that you need to perform are:

1. Run the sqlj command to produce Java bytecodes and a serialized profile:
   ```
   sqlj s1/s1Sal.sqlj
   ```
2. Run the db2profc command to produce DBRM S1SAL from serialized profile s1Sal_SJProfile0.ser in path s1:
   ```
   db2profc -pgmname=S1SAL s1/s1Sal_SJProfile0.ser
   ```
3. Run the jar tool to create a JAR file for serialized profile s1Sal_SJProfile0.ser. As is required for VisualAge for Java, specify the -M option so that you do not include a manifest in the JAR file.
   ```
   jar -Mcv0f sqljprofile.jar s1Sal_SJProfile0.ser
   ```

   The JAR file for the SQLJ profile needs to be in the HFS directory structure at the same level as the package.
4. Run the hpj command to convert the Java bytecodes that were produced by the SQLJ translator into a compiled Java program that runs as a stored procedure. Include the following input files:

s1/s1Sal
The class for the stored procedure program

s1/NameSal
The generated class for iterator NameSal, which is used by the program

s1/s1Sal_SJProfileKeys
An internal class that is generated by the SQLJ translator

sjprofile.jar
The JAR file for serialized profile that is generated by the SQLJ translator. This file needs to be included as a resource.

Save the link-edit information in a file called s1.map.

```
hpj -o="//'HPJSP.PDSE1(S1)'" -alias=s1.jll -O  \
-classpath=.:/usr/lpp/hpj/lib/classes.zip -jll -nofollow \
s1/NameSal s1/s1Sal_SJProfileKeys \
-resource sjprofile.jar \
-t=/u/sysadm/links s1/s1Sal \
> s1.map
```

This example uses three options that are useful but not required for compiled Java stored procedures:

-O Specifies that hpj produces optimized code.

-classpath
Overrides the default classpath.

-nofollow
Specifies that referenced classes are not bound into the Java DLL for the stored procedure.

Building an SQLJ or JDBC program under VisualAge for Java

With VisualAge for Java, you can prepare an SQLJ or JDBC program for execution as a Java executable or as a Java DLL. In general, you build executables for stand-alone programs. You build Java DLLs when the source code contains either of the following items:
- Supporting class packages for executables
- Compiled Java stored procedures

A Java executable must have at least one class that contains a main method. A Java DLL does not need to contain a main method.

Follow these steps to build an SQLJ or JDBC program for execution under VisualAge for Java:
1. For an SQLJ program, use the `sqlj` command to translate and compile the source code. This process produces modified Java source code, serialized profiles, and Java bytecodes.
2. For an SQLJ program, customize the serialized profiles using the `db2profc` command to produce DBRMs.

3. For a Java program other than an SQLJ program, compile the Java program using the `javac` command to produce Java bytecodes.
4. Create JAR files for any Java resource files that you want to bind with the application, such as SQLJ profiles and the JDBC profile.
5. Use the VisualAge for Java hpj command to bind the Java bytecode files for the program.
6. If the output from the hpj command is a Java DLL in an HFS file, create HFS links for all packages in the DLL.
7. For an SQLJ program, use the DB2 BIND command to bind the DBRMs into packages and plans, or directly into plans, using the DB2 `BIND` command.

Steps 1, 2, 3, and 7 are common to all Java applications. They are discussed in "Chapter 5. Preparing Java programs" on page 69. "Binding the JDBC and SQLJ profiles as VisualAge for Java resource files" on page 100 discusses step 4. This section discusses steps 5 and 6.

Binding an SQLJ or JDBC program for VisualAge for Java

Use the VisualAge for Java hpj command to bind a Java program into a Java executable or DLL. The VisualAge for Java documentation contains a complete explanation of the hpj command. The following options are commonly used for SQLJ and JDBC applications:

-jll Indicates that the output from the hpj invocation is a Java DLL.

-exe
Indicates that the output from the hpj invocation is a Java executable.

-nofollow
Indicates that hpj should bind the Java executable using *only* the classes that are in the input list. No referenced classes are automatically included.

-resource
Indicates that when hpj binds the Java executable or DLL, it does the following things with Java resource files that it finds in input JAR or zip files:

- Binds the resource files into the Java executable or DLL
- If the output from the hpj command is a PDSE member, creates aliases for the resource files.

-alias=*alias-name*
Indicates that when hpj binds a Java DLL or executable and puts it in a PDSE, hpj creates an alias for the PDSE member. If you also specify the -t option, hpj creates an HFS external link that matches the alias name.

You must specify the -alias option when you bind Java DLLs into PDSE members. At run time, VisualAge for Java uses the alias name to find the DLL.

alias-name is the Java package name with a .jll extension. For example, if you build a Java DLL from a package named a.b.c, you specify -alias=a/b/c.jll.

Java DLLs for compiled Java stored procedures and CICS applications must reside in PDSE members. Therefore, you need to use the -alias option when you bind stored procedures and CICS applications.

-o=*output-file-name*
Specifies the output file name. The format of the file name indicates whether the file is an HFS file or a PDS member name. For an HFS file, specify only the qualified or unqualified file name. For a PDSE member, specify the name in the format "//'*data-set-name(member-name)*'". For example, if you want to create member S1 in data set HPJSP.PDSE1, the -o parameter that you specify is -o="//'HPJSP.PDSE1(S1)'".

-t=*directory-name*
Creates links that are required to access Java DLLs in PDSEs or to access resource files. hpj creates the links in *directory-name*.

Use the -t option with the -alias option when you create Java DLLs in PDSE members, and the applications that access the DLLs use HFS file access. The links that hpj creates for DLLs in PDSE members are HFS external links.

Use the -t option with the -resource option to create links for resource files. The links that hpj creates are HFS symbolic links if the resources are bound into an HFS file. The links are HFS external links if the resources are bound into a PDSE member.

Before you run an application that uses the links, you need to put *directory-name* in your CLASSPATH concatenation.

Do not use the -t option for CICS applications, which do not use HFS file access.

Use the -t option for compiled Java stored procedures.

Example: Using hpj to create an SQLJ executable: Suppose that you have completed the first three steps in the SQLJ program preparation process, and you want your SQLJ program to run under VisualAge for Java. This example creates an Java application executable that also contains the JDBC profile, the SQLJ profile, and the run-time properties file. The example uses the following data sets:

a.b.c
The package that contains all classes for the application. The corresponding relative directory structure for the classes is a/b/c.

App1.class
The application class that contains the main method.

App1Conn.class
The SQLJ connection context that the application uses.

App1Iter.class
The SQLJ iterator that the application uses.

App1_SJProfileKeys.class
An internal class that the SQLJ translator creates.

App1_SJProfile0.ser
The SQLJ profile that the SQLJ translator creates.

DSNJDBC_JDBCProfile.ser
The JDBC profile. This file is not in the same path as the application classes.

db2sqljjdbc.properties.ebcdic
The run-time properties file, in EBCDIC format. This file is not in the same path as the application classes.

The first step that you need to perform is to create the run-time properties file in ASCII format. To do that, execute the `iconv` utility:

```
iconv -f ibm-1047 -t utf-8 db2sqljjdbc.properties.ebcdic > db2sqljjdbc.properties
```

The ASCII run-time properties file is now in file db2sqljjdbc.properties.

Next, run the jar tool to create JAR files for the Java resource files. As is required for VisualAge for Java, specify the -M option so that you do not include a manifest in any of the JAR files.

```
jar -Mcv0f sqljprofile.jar a/b/c/App1_SJProfile0.ser
jar -Mcv0f jdbcprofile.jar DSNJDBC_JDBCProfile.ser
jar -Mcv0f properties.jar  db2sqljjdbc.properties
```

The JAR file for the SQLJ profile needs to be in the HFS directory structure at the same level as the package.

Next, include the directory path that contains a/b/c in your CLASSPATH concatenation. Then you do not need to refer to files by their full path name when you execute the hpj command. For example, suppose that the full path name for App1.class is `/usr/myname/project1/a/b/c/App1.class`. If you put `/usr/myname/project1` in the CLASSPATH, you can refer to `/usr/myname/project1/a/b/c/App1.class` as `a.b.c.App1.class`.

Now you are ready to use the hpj command to create the executable:

```
hpj -o app1.exe -exe -nofollow \
  a.b.c.App1 \
  a.b.c.App1Conn \
  a.b.c.App1Iter \
  a.b.c.App1_SJProfileKeys \
  -resource \
  sqljprofile.jar jdbcprofile.jar properties.jar \
        -t /usr/lpp/db2710/vajlinks
```

The meanings of the parameters in the hpj invocation are:

a.b.c.App1, a.b.c.App1Conn, a.b.c.App1Iter, a.b.c.App1_SJProfileKeys, sqljprofile.jar, jdbcprofile.jar, properties.jar	The input files. Input files can appear anywhere within the command string. In this example, the JAR files are separated from the other input files to emphasize that they are resources.
-exe	Indicates that the output file is a Java executable.
-o app1.exe	Indicates the name and format of the output file. The format of the file name indicates that the output is an HFS file.
-nofollow	Indicates that hpj should bind the Java executable using *only* the classes that are in the input list and the JDK classes.

-resource	Indicates that when hpj binds the Java executable, it binds the three resource files into the executable.
-t /usr/lpp/db2710/vajlinks	Causes hpj to create symbolic links for the three resource files that are included in the DLL. hpj creates the links in the /usr/lpp/db2710/vajlinks directory.

Example: Using hpj to create a Java DLL: Suppose that you have completed the first three steps in the SQLJ program preparation process on each of two Java classes. The classes are bound into two different packages. This example creates a Java DLL that contains those two packages. The example also creates symbolic links for the packages that are in the DLL. The example uses the following data sets:

a.b.c
The package that contains all classes for application App1. The corresponding relative directory structure for the classes is a/b/c.

x.y.z
The package that contains all classes for application App2. The corresponding relative directory structure for the classes is x/y/z.

App1.class
The application class for application App1.

App2.class
The application class for application App2.

App1Conn.class
The SQLJ connection context that the App1 uses.

App2Conn.class
The SQLJ connection context that the App2 uses.

App2Iter.class
The SQLJ iterator that the App1 uses.

App2Iter.class
The SQLJ iterator that App2 uses.

App1_SJProfileKeys.class
An internal class that the SQLJ translator creates for App1.

App2_SJProfileKeys.class
An internal class that the SQLJ translator creates for App2.

App1_SJProfile0.ser
The SQLJ profile that the SQLJ translator creates for App1.

App2_SJProfile0.ser
The SQLJ profile that the SQLJ translator creates for App2.

DSNJDBC_JDBCProfile.ser
The JDBC profile. This file is not in the same path as the application classes.

db2sqljjdbc.properties.ebcdic
The run-time properties file, in EBCDIC format. This file is not in the same path as the application classes.

The first step that you need to perform is to create the run-time properties file in ASCII format. To do that, execute the `iconv` utility:

```
iconv -f ibm-1047 -t utf-8 db2sqljjdbc.properties.ebcdic > db2sqljjdbc.properties
```

The ASCII run-time properties file is now in file db2sqljjdbc.properties.

Next, run the jar tool to create JAR files for the Java resource files. Include both of the SQLJ profiles in the same JAR file. As is required for VisualAge for Java, specify the -M option so that you do not include a manifest in any of the JAR files.

```
jar -Mcv0f sqljprofile.jar a/b/c/App1_SJProfile0.ser x/y/z/App2_SJProfile0.ser
jar -Mcv0f jdbcprofile.jar DSNJDBC_JDBCProfile.ser
jar -Mcv0f properties.jar  db2sqljjdbc.properties
```

Now you are ready to use the hpj command to create the Java DLL:

```
hpj -o allApps.jll -jll -nofollow \
  a.b.c.App1 \
  a.b.c.App1Conn \
  a.b.c.App1Iter \
  a.b.c.App1_SJProfileKeys \
  x.y.z.App2 \
  x.y.z.App2Conn \
  x.y.z.App2Iter \
  x.y.z.App2_SJProfileKeys \
  -resource \
  sqljprofile.jar jdbcprofile.jar properties.jar \
  -t /usr/lpp/db2710/vajlinks
```

The meanings of the parameters in the hpj invocation are:

a.b.c.App1, a.b.c.App1Conn, a.b.c.App1Iter, a.b.c.App1_SJProfileKeys, x.y.z.App2, x.y.z.App2Conn, x.y.z.App2Iter, x.y.z.App2_SJProfileKeys, sqljprofile.jar, jdbcprofile.jar, properties.jar	The input files. Input files can appear anywhere within the command string. In this example, the JAR files are separated from the other input files to emphasize that they are resources.
-jll	Indicates that the output file is a Java DLL.
-o allApps.jll	Indicates the name and format of the output file. The format of the file name indicates that the output is an HFS file.
-nofollow	Indicates that hpj should bind the Java executable using *only* the classes that are in the input list and the JDK classes.
-resource	Indicates that when hpj binds the Java DLL, it binds the four resource files into the DLL.

-t /usr/lpp/db2710/vajlinks	Causes hpj to create symbolic links for the four resource files that are included in the DLL. hpj creates the links in the /usr/lpp/db2710/vajlinks directory.

Creating symbolic links for packages in a Java DLL

When an application that runs under VisualAge for Java references a class that is not in the Java DLL or executable that is running, that reference is called a *nonlocal reference.* VisualAge for Java needs to be able to associate a nonlocal reference with a Java DLL. Unless the package that contains the referenced class is in the set of directories that VisualAge for Java searches by default, VisualAge for Java needs a link to find the DLL.

When you bind a Java DLL into a PDSE member, you can specify the -t and -alias options to make hpj automatically create a link between a package name and the PDSE member that contains the DLL. However, if the Java DLL is in an HFS file, you need to create the link between the package name and the HFS file yourself. You use the HFS ln command to create the link.

Example: Creating a symbolic link for a Java DLL: In the previous example, the Java DLL that you create contains two packages: `a.b.c` and `x.y.z`. By default, when VisualAge for Java looks for a method in `a.b.c`, it looks for the method in DLL files with one of these relative path names:

```
a/b/c.jll
a/b.jll
a.jll
```

Similarly, when VisualAge for Java looks for a method in `x.y.z`, it looks for the method in DLL files with one of these relative path names:

```
x/y/z.jll
x/y.jll
x.jll
```

Because the real DLL that contains the packages is called allApps.jll, you need to create symbolic links from at least one of the DLL names that VisualAge for Java looks for to the real DLL name.

Suppose that you created allApps.jll in directory /usr/myname/project1/jlls. Use OS/390 UNIX System Services commands similar to these to create symbolic links from the names a/b/c.jll and x/y/z.jll, where VisualAge for Java looks for methods, to /usr/myname/project1/jlls/allApps.jll, where the methods really are:

```
cd /usr/lpp/db2x10/vajlinks
mkdir a
cd a
mkdir b
cd b
ln -s /usr/myname/project1/jlls/allApps.jll c.jll
cd /usr/lpp/db2x10/vajlinks
mkdir x
cd x
```

```
mkdir y
cd y
ln -s /usr/myname/project1/jlls/allApps.jll z.jll
```

Chapter 6. JDBC and SQLJ administration

This chapter contains the following topics about the administration of JDBC and SQLJ:

- "Installing JDBC and SQLJ"
- "JDBC and SQLJ security model" on page 102
- "JDBC and SQLJ multiple OS/390 context support" on page 103
- "JDBC and SQLJ connection pooling support" on page 105
- "JDBC and SQLJ global transaction support" on page 105

Installing JDBC and SQLJ

The procedures in this section describe what you need to do to install DB2 JDBC and SQLJ. All procedures, except for those noted differently, are for both JDBC and SQLJ installation.

To install JDBC and SQLJ, follow these steps:

1. 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" on page 90 for details.
2. Log on to TSO.

   Specify a maximum region size of at least 200000.

   Ensure that you have superuser authority (UID 0).
3. ***Optional:*** Set the program control extended attribute for SQLJ/JDBC driver DLLs. This step is necessary only if a product that uses the SQLJ/JDBC driver requires it. See "Setting the program control extended attribute" on page 90 for details.
4. In OS/390 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" on page 91 for details.
5. ***Optional:*** In OS/390 UNIX System Services, customize the SQLJ/JDBC run-time properties file. See "Customizing parameters in the SQLJ/JDBC run-time properties file" on page 92 for details.

   The default path name is /usr/lpp/db2/db2710/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.
6. ***Optional:*** In OS/390 UNIX System Services, customize the JDBC cursor properties file. You do not need to perform this step unless you need to alter the number or properties of JDBC cursors. See "Customizing the cursor properties file (optional)" on page 97 for details.
7. ***Optional:*** Run the db2genJDBC utility in OS/390 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 95 for details.
8. Prepare the JDBC DBRMs for execution.

If you did not run the db2genJDBC utility, these are the DBRMs in the DSN710.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 DSN710.SDSNSAMP. See “Binding the DBRMs” on page 96 for details.

In TSO, grant EXECUTE authority on the packages and plan to PUBLIC.

9. If you plan to run compiled Java programs, follow the steps in “Installing VisualAge for Java support for JDBC and SQLJ” on page 97.

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 Server for OS/390 and z/OS Program Directory* for information on allocating and loading DB2 data sets.

Setting the program control extended attribute

After you load the JDBC and SQLJ libraries, you might need to execute the OS/390 UNIX System Services extattr command to set the program control extended attribute for all files in the /usr/lpp/db2/db2710/lib directory. The command looks like this:

```
extattr +p /usr/lpp/db2/db2710/lib/*
```

Although the SQLJ/JDBC driver does not require any program control settings, some programs that use the SQLJ/JDBC driver, such as WebSphere Application Server, include privileged code that requires a program-controlled environment. See the documentation for these programs to determine whether they require a program-controlled environment.

You must reset the program control extended attribute when you apply maintenance to the SQLJ/JDBC driver.

Setting environment variables

The environment variables that you must set are:

STEPLIB

Modify STEPLIB to include the `SDSNEXIT`, `SDSNLOAD`, and `SDSNLOD2` data sets. For example:

```
export STEPLIB=DSN710.SDSNEXIT:DSN710.SDSNLOAD:DSN710.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/db2`, modify PATH as follows:

```
export PATH=/usr/lpp/db2/db2710/bin:$PATH
```

LIBPATH and LD_LIBRARY_PATH

The DB2 for OS/390 and z/OS SQLJ/JDBC driver contains several dynamic load libraries (DLLs).

Modify LIBPATH and LD_LIBRARY_PATH to include the directory that contains these DLLs. If SQLJ and JDBC are installed in `/usr/lpp/db2`, modify LIBPATH and LD_LIBRARY_PATH, respectively, as follows:

```
export LIBPATH=/usr:/usr/lib:/usr/lpp/db2/db2710/lib:$LIBPATH
```

```
export LD_LIBRARY_PATH=/usr/lpp/db2/db2710/lib:$LD_LIBRARY_PATH
```

CLASSPATH

If you plan to use the *JDBC 1.2* driver, modify CLASSPATH to include one of the following class files:

`db2sqljclasses.zip`

Contains all of the classes necessary to prepare and run JDBC and SQLJ programs with the JDBC 1.2 driver. Assuming that JDBC and SQLJ are installed in `/usr/lpp/db2`, modify CLASSPATH as follows:

```
export CLASSPATH=/usr/lpp/db2/db2710/classes/db2sqljclasses.zip:$CLASSPATH
```

`db2sqljruntime.zip`

Contains only the classes that are needed to run JDBC and SQLJ programs with the JDBC 1.2 driver. This file is smaller than the `db2sqljclasses.zip` file, which contains files for program preparation and execution. Specify this class file only if you do not plan to prepare SQLJ programs on your system. Assuming that JDBC and SQLJ are installed in `/usr/lpp/db2`, modify CLASSPATH as follows:

```
export CLASSPATH=/usr/lpp/db2/db2710/classes/db2sqljruntime.zip:$CLASSPATH
```

`db2jdbcclasses.zip`

The `db2jdbcclasses.zip` file is provided to maintain compatibility with existing DB2 for OS/390 and z/OS JDBC applications. The contents of `db2jdbcclasses.zip` are equivalent to the contents of `db2sqljclasses.zip`

If you plan to use the *JDBC 2.0* driver, 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/db2`, modify CLASSPATH as follows:

```
export CLASSPATH=/usr/lpp/db2/db2710/classes/db2j2classes.zip:$CLASSPATH
```

DB2SQLJPROPERTIES

Specifies the fully-qualified name of the run-time properties file for the DB2 for OS/390 and z/OS SQLJ/JDBC driver. The run-time properties file contains various entries of the form *parameter*=*value* that specify program preparation and run-time options that the DB2 for OS/390 and z/OS SQLJ/JDBC driver 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/db2/db2710/classes` directory, specify:

```
export DB2SQLJPROPERTIES=/usr/lpp/db2/db2710/classes/db2sqljjdbc.properties
```

If you use Java stored procedures, you need to set additional environment variables in a JAVAENV data set. See “Setting environment variables for Java routines” on page 54 for more information.

Customizing parameters in the SQLJ/JDBC run-time properties file

To customize the SQLJ/JDBC run-time environment, you need to provide information about your environment in the SQLJ/JDBC run-time properties file.

For the CICS environment, the settings for some of the environment variables and run-time properties parameters are different than for other environments. See “Appendix B. Special considerations for CICS applications” on page 113 for information that is specific to CICS.

The parameters that you can set in the run-time properties file are:

DB2SQLJDBRMLIB

Specifies the fully-qualified name of the MVS 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 DSN710.SDSNMACS. The DCB attributes of the DBRM data set are RECFM FB and LRECL 80.

See “Customizing the JDBC profile (optional)” on page 95 and “Customizing an SQLJ serialized profile” on page 72 for more information on serialized profile customization.

DB2SQLJPLANNAME
Specifies the name of the plan that is associated with a JDBC or an SQLJ application. The plan is created by the DB2 for OS/390 and z/OS bind process. For example:

```
DB2SQLJPLANNAME=SQLJPLAN
```

The default name is DSNJDBC.

DB2SQLJJDBCPROGRAM
Specifies the name of a connected profile that is used by the DB2 for OS/390 and z/OS SQLJ/JDBC driver. For example:

```
DB2SQLJJDBCPROGRAM=CONNPROF
```

The default connected profile name is DSNJDBC.

See “Customizing the JDBC profile (optional)” on page 95 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
```

The default is the subsystem name that was specified during installation of the local DB2 subsystem.

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=NO
```

The default is YES.

For DB2SQLJMULTICONTEXT=YES to be in effect, the OS/390 system must be OS/390 Version 2 Release 5 or later. If this condition is not met, SQLJ operates as if DB2SQLJMULTICONTEXT=NO.

See “JDBC and SQLJ multiple OS/390 context support” on page 103 for more information on multiple OS/390 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.

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 /tmp/jdbctrace and /tmp/jdbctrace.JTRACE:

```
DB2SQLJ_TRACE_FILENAME=/tmp/jdbctrace
```

See “Formatting trace data” on page 36 for more information on the SQLJ/JDBC trace.

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
```

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 the parameters for diagnostic traces (`DB2SQLJ_TRACE_FILENAME`, `DB2SQLJ_TRACE_BUFFERSIZE`, and `DB2SQLJ_TRACE_WRAP`) only under the direction of your IBM service representative. See “Formatting trace data” on page 36 for information on formatting trace data.

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

```
►►──db2genJDBC──┬──────────────────────────────────┬──┬──────────────────────────────┬──────────────►
                │              ┌─DSNJDBC──────┐    │  │                ┌─150─────┐   │
                └─- pgmname=───┴─program-name─┴────┘  └─- statements=──┴─integer─┴───┘

►──┬─────────────────────────────────────────────┬──┬───────────────────────┬──────────────────────►◄
   │              ┌─db2jdbc.cursors───────────┐  │  │           ┌─5───────┐ │
   └─- cursors=───┴─cursor-properties-file────┴──┘  └─- calls=───┴─integer─┴─┘
```

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 “Appendix B. Special considerations for CICS applications” on page 113 for more information.

-cursors

Specifies the name of the cursor properties file, described in “Customizing the cursor properties file (optional)” on page 97. 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 specified in the CLASSPATH environment variable, described in “Setting environment variables” on page 91.

-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 the directory that is specified in the CLASSPATH environment variable.

The JDBC profile name is in the following format:

program-name`_JDBCProfile.ser`

Binding the DBRMs

Bind the DBRMs into packages and include those packages in the plan for JDBC. For JDBC applications, you need to bind the packages into the DSNJDBC plan. You can use sample job DSNTJJCL to bind the packages and the plan. The sample bind job is shipped in DSN710.SDSNSAMP(DSNTJJCL).

The DBRM names and isolation levels are as follows:

- *program-name*1: Bind with transaction isolation level = UR
- *program-name*2: Bind with transaction isolation level = CS
- *program-name*3: Bind with transaction isolation level = RS
- *program-name*4: Bind with transaction isolation level = 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 packages in the plan that each SQLJ application accesses at run time. This step is necessary for SQLJ applications to interoperate with JDBC. There is one DBRM for each transaction isolation level.

For more information about SQLJ and JDBC interoperability, see “Customizing SQLJ and JDBC to work together” on page 76. For information on binding packages and plans, see Chapter 2 of *DB2 Command Reference*.

Customizing the cursor properties file (optional)

JDBC result sets require a valid DB2 cursor. 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 125 cursors with hold and 125 cursors without hold.

For CICS applications, you should not use the default value. See “Appendix B. Special considerations for CICS applications” on page 113 for more information.

To customize the cursor properties file, edit the file that you specify in the `-cursors` option of the db2genJDBC utility (described in “Customizing the JDBC profile (optional)” on page 95), and define an entry for each JDBC cursor.

Syntax

```
                          ┌─nohold─┐
►►──cursor=──cursorname───┤        ├────────────────────────────────────────────►◄
                          └─hold───┘
```

Parameter descriptions

cursorname
Specifies the cursor name. This name must be 18 or fewer characters in length.

hold|nohold
Specifes hold attribute for the cursor. If a hold attribute is not specified, the cursor is assigned the `nohold` attribute. The default is `nohold`.

For example:

```
cursor=SAMPLECURSORA:hold
cursor=SAMPLECURSORB:nohold
```

Installing VisualAge for Java support for JDBC and SQLJ

If you want to run your JDBC and SQLJ applications as compiled Java applications, you need to install VisualAge for Java, Enterprise Edition for OS/390 and use the Enterprise Toolkit for OS/390. This section explains how to customize the SQLJ/JDBC driver to work with VisualAge for Java. See the documentation that is distributed with VisualAge for Java and the following web sites for detailed information on how to install and use VisualAge for Java and how to prepare and run VisualAge for Java programs.

VisualAge for Java web site:
`www.ibm.com/software/ad/vajava`

VisualAge Developer Domain web site:
`www.ibm.com/software/ad/vadd`

Installing and accessing SQLJ/JDBC DLLs for VisualAge for Java support

After you install SQLJ, JDBC, and VisualAge for Java, you need to install two DLLs that make SQLJ and JDBC work with VisualAge for Java. This section explains how to install those DLLs and how your applications can access them.

To help you install the DLLs, use the installVAJDLLs script, which is located in the root installation directory for SQLJ and JDBC support.

Install DLLs that support VisualAge for Java: To make SQLJ and JDBC work with VisualAge for Java, you need the following DLLs:

- DSNAQDLL

  This is the native C/C++ DLL for VisualAge for Java. The SQLJ/JDBC driver loads this DLL when an SQLJ or JDBC application is running in a VisualAge for Java environment.

  For SQLJ and JDBC applications that do not support HFS file access, such as CICS applications, an alias named `libdb2os390vaj.so` is required. The installVAJDLLs script creates this alias.

  For SQLJ and JDBC applications that require HFS file access for DLL resolution, such as compiled Java stored procedures, an external link to the PDSE member that contains DSNAQDLL is required. The installVAJDLLs script also creates this external link.

- DSNAQJLL

  This DLL provides Java run-time classes for the SQLJ/JDBC driver.

  For SQLJ and JDBC applications that do not support HFS file access, such as CICS applications, three aliases for DSNAQJLL are required. Those aliases are named `sqlj.jll`, `ibm/sql.jll`, and `COM/ibm/db2os390/sqlj.jll`. The installVAJDLLs script creates the aliases.

  For SQLJ and JDBC applications that require HFS file access for DLL resolution, such as compiled Java stored procedures, an external link to the PDSE member that contains DSNAQJLL is required. The installVAJDLLs script also creates that external link.

To install the DLLs, follow these steps:

1. Create a PDSE for the DLLs.

   The PDSE needs to have a primary extent of at least 25 cylinders and secondary extents of at least five cylinders of DASD.

2. Customize and run the installVAJDLLs script to install the DLLs in your PDSE. See the prolog of installVAJDLLs for information on how to customize it.

Accessing DLLs for VisualAge for Java support at run time: The way in which you access the DLLs that let SQLJ and JDBC work with VisualAge for Java depends on whether your applications run in the CICS environment.

Accessing the DLLs outside of the CICS environment: For applications that do not run under CICS, you need to make the following modifications to your LIBPATH, STEPLIB, and CLASSPATH concatenations so that the applications can access the DLLs for VisualAge for Java support:

LIBPATH
Include the directory that contains the external link to the DSNAQDLL DLL.

STEPLIB
Include the PDSE that contains the DSNAQDLL and DSNAQJLL DLLs.

CLASSPATH
Include the directory that contains the external link to the DSNAQJLL DLL.

Accessing the DLLs in the CICS environment: For applications that run under CICS, you do not modify the STEPLIB, CLASSPATH, or LIBPATH concatenations. Instead, you perform the following steps:

1. Include the PDSE that contains the DSNAQDLL and DSNAQJLL DLLs in the DFHRPL concatenation.
2. Define the DSNAQDLL and DSNAQJLL DLLs to CICS as as programs. For example, you might use statements like these to define DSNAQDLL and DSNAQJLL to CICS and add them to a CICS group named JDBCSQLJ:

```
DEFINE PROGRAM(DSNAQDLL)
  DESCRIPTION(JDBC AND SQLJ NATIVE RUNTIME)
  EXECKEY(CICS)
  GROUP(JDBCSQLJ)

DEFINE PROGRAM(DSNAQJLL)
  DESCRIPTION(JDBC AND SQLJ JAVA RUNTIME)
  EXECKEY(CICS)
  GROUP(JDBCSQLJ)
```

Accessing SQLJ and JDBC profiles and the run-time properties file under VisualAge for Java

JDBC and SQLJ support includes the JDBC profile and the SQLJ/JDBC run-time properties file, which all of your SQLJ and JDBC applications need to access. In addition, SQLJ applications must access their SQLJ profiles. When you run your applications under VisualAge for Java, you can use one of two techniques to access these files. The technique that you use depends on whether you are using HFS file access:

- Technique 1 (for VisualAge for Java applications that support HFS file access *only*):

  Include the directory that contains the JDBC profile or SQLJ profiles in your CLASSPATH concatenation. Include the directory that contains the run-time properties file in your DB2SQLJPROPERTIES environment variable.
- Technique 2 (for any VisualAge for Java applications):

  Using VisualAge for Java, bind the file as a Java *resource file*. In VisualAge for Java, a resource file is defined as a non-code file that you can refer to from your Java program.

  CICS applications do not support HFS file access, so you must use this technique for CICS applications.

The following sections explain how to bind your profiles and run-time properties file as resource files and how to access those resource files at run time.

Binding the JDBC and SQLJ profiles as VisualAge for Java resource files: You can bind the JDBC and SQLJ profiles as resource files in one of two ways:

- Bind the profile into individual application executables.

  This is the preferred technique for SQLJ profiles because an SQLJ profile is generally associated with a single application.
- Bind the profile into its own Java DLL.

  This is the preferred technique for JDBC profiles because a JDBC profile is generally associated with many or all applications.

  If one profile works with all your applications, you can bind the profile with the SQLJ/JDBC driver.

  If one profile works for some, but not all, applications, you can bind each version of the profile into its own Java DLL, and make each DLL visible to only those applications with which it works.

You can include several instances of a resource file in a single Java DLL. To distinguish between the instances of the resource, you give them different names. At run time, you need to set the appropriate environment variable or run-time properties file parameter to specify the correct resource name. Because CICS VisualAge for Java applications do not use the SQLJ/JDBC environment variables, you cannot rename the SQLJ/JDBC run-time properties file for CICS applications. This means that for CICS applications, you can have only one copy of the SQLJ/JDBC run-time properties file in a Java DLL. For more information on CICS restrictions, see “Appendix B. Special considerations for CICS applications” on page 113.

Follow these steps to bind a profile as a resource file:

1. Use the jar tool to create a JAR file that contains the JDBC profile.

   An example of invoking the jar tool is:

   ```
   jar -Mcv0f jdbcprofile.jar DSNJDBC_JDBCProfile.ser
   ```

   Always specify the -M option when you create a JAR file from a resource file. The -M option prevents the creation of a manifest.
2. Use the hpj command to bind the profile.

   Include the -resource and -t options when you execute the hpj command so that hpj processes the profile as a resource file and creates the appropriate links for it. See “Building an SQLJ or JDBC program under VisualAge for Java” on page 80 for more information on these options.

Accessing a JDBC or SQLJ profile as a resource file at run time: The way in which you access a JDBC or SQLJ profile in VisualAge for Java applications depends on whether the applications run in a CICS environment.

Accessing the profile outside of the CICS environment: For applications that do not run under CICS, you need to make the following modifications to your STEPLIB and CLASSPATH concatenations so that the applications can access the profiles:

STEPLIB
If the executable or DLL that contains the profile is a PDSE member, include the PDSE that contains the member.

CLASSPATH
Include the directory that contains the external link to the profile.

Accessing the profile in the CICS environment: For applications that run under CICS, perform the following steps:

1. Include the PDSE that contains the profile in the DFHRPL concatenation.
2. Define the executable or DLL that contains the profile to CICS as a program.

Binding the SQLJ/JDBC run-time properties file as a VisualAge for Java resource file: Follow these steps to bind the run-time properties file as a resource file:

1. Convert the run-time properties file to ASCII.

   Use the OS/390 C/C++ `iconv` utility or the OS/390 UNIX System Services `iconv` shell utility to do this. For example, to convert an EBCDIC run-time properties file named db2sqljjdbc.properties.ebcdic to an ASCII file named db2sqljjdbc.properties, execute a command like this:

   ```
   iconv -f ibm-1047 -t utf-8 db2sqljjdbc.properties.ebcdic > db2sqljjdbc.properties
   ```
2. Create a JAR file that contains the converted run-time properties file.
3. Bind the JAR file into an application executable or into its own DLL.

   Binding the run-time properties file into a DLL is usually the best approach for CICS applications. If all CICS applications that run in the same address space use the same set of properties, those applications can access the same DLL. For Java stored procedures, which might need different run-time properties than the default properties, accessing a DLL that contains the run-time properties file is also most practical.

Accessing the SQLJ/JDBC run-time properties file as a resource file at run time: The way in which you access the SQLJ/JDBC run-time properties file in VisualAge for Java applications depends on whether the applications run in a CICS environment.

Accessing the run-time properties file outside of the CICS environment: For applications that do not run under CICS, you need to make the following modifications to your STEPLIB and CLASSPATH concatenations so that the applications can access the run-time properties file:

STEPLIB
If the executable or DLL that contains the run-time properties file is a PDSE member, include the PDSE that contains the member.

CLASSPATH
Include the directory that contains the external link to the run-time properties file.

Accessing the run-time properties file in the CICS environment: For applications that run under CICS, perform the following steps:

1. Include the PDSE that contains the run-time properties file in the DFHRPL concatenation.

2. Define the executable or DLL that contains the run-time properties file to CICS as a program.

JDBC and SQLJ security model

This section describes the JDBC and SQLJ security model. It explains how authorization IDs are determined and the use of attachment facilities.

How are authorization IDs established?

The method that DB2 uses to establish an authorization ID for a process depends on the level of the JDBC driver.

Determining an authorization ID with the JDBC 1.2 driver

With the JDBC 1.2 driver, the security environment created by an external security product, such as OS/390 Security Server (RACF ACEE), determines the DB2 authorization ID that is used for a thread. A user ID and password is not specified for an SQLJ connection context or a JDBC connection.

DB2 does not create the security environment. The application or server that provides the point of entry into the OS/390 system, such as a TSO logon, Telnet logon, or Web Server, typically creates the security environment.

Determining an authorization ID with the JDBC 2.0 driver

With the JDBC 2.0 driver, 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 provide a user ID and password, the JDBC driver passes these values to DB2 for validation and uses these 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.

DB2 attachment types

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 “Appendix B. Special considerations for CICS applications” on page 113 for information on using the CICS attachment facility.

Using the RRS 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.

JDBC and SQLJ multiple OS/390 context support

An OS/390 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.

Connecting when multiple OS/390 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 OS/390 context support is enabled

With multiple OS/390 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
- OS/390 Context Services, available in OS/390 Version 2, Release 5 or higher

With multiple OS/390 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`s.
  - 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.

For information about using JDBC connections for SQLJ operations, see “Customizing SQLJ and JDBC to work together” on page 76.

Enabling multiple OS/390 context support

The DB2SQLJMULTICONTEXT parameter in the run-time properties file enables multiple context support. See “Customizing parameters in the SQLJ/JDBC run-time properties file” on page 92 for information about setting the DB2SQLJMULTICONTEXT parameter.

Multiple context performance

Setting the DB2SQLJMULTICONTEXT parameter to YES enhances SQLJ and JDBC performance if the operating system is OS/390 Release 6 or higher and OS/390 APAR OW41492 is applied.

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 OS/390, 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.

JDBC and SQLJ connection pooling support

Connection pooling is part of JDBC 2.0 data source support. 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. Connection pooling is a built-in part of JDBC 2.0 data source support. Connection pooling support is completely transparent to the JDBC application.

Without connection pooling, each `java.sql.Connection` object represents a physical connection to the database. 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 the physical data source connection.

The JDBC 2.0 connection pooling framework lets a single physical data source connection 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.

JDBC and SQLJ 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 for OS/390 and z/OS relational data within global transactions. Global transaction support is available with the JDBC 2.0 driver. 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 13 on page 106 shows, with local transactions, you call tbe `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. 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 13. Example of a local transaction

In contrast, applications that participate in global transactions cannot call the `commit`, `rollback`, or `setAutoCommit` methods on the `Connection` object. 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 for OS/390 and z/OS does not implement the JTA specification, the methods for delimiting transaction boundaries are available with the JDBC 2.0 driver. Figure 14 demonstrates the use of the JTA interfaces to indicate global transaction boundaries.

```
// 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 14. 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 15 on page 107 illustrates this point.

```
// 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 15. Example of a global transaction in a multi-threaded environment

Appendix A. Selected sqlj.runtime classes and interfaces

The `sqlj.runtime` package defines the run-time classes and interfaces that SQLJ uses. This appendix describes:

- Each class of `sqlj.runtime` that contains methods that you can invoke in your SQLJ application programs
- Each of the interfaces that you might need to implement in your SQLJ application programs

sqlj.runtime.ExecutionContext class

The `sqlj.runtime.ExecutionContext` class is defined for execution contexts. You can use an execution context to control the execution of SQL statements. After you declare an execution context and create an instance of that execution context, you can use the following methods.

getMaxFieldSize

Format:

```
public int getMaxFieldSize()
```

Returns the maximum number of bytes that are returned for any character column in queries that use the given execution context. A value of 0 means that the maximum number of bytes is unlimited.

getMaxRows

Format:

```
public int getMaxRows()
```

Returns the maximum number of rows that are returned for any query that uses the given execution context. A value of 0 means that the maximum number of rows is unlimited.

getNextResultSet

Format:

```
public ResultSet getNextResultSet()
```

After a stored procedure call, returns a result set from the stored procedure. Each call to getNextResultSet closes the result set that was retrieved by the previous call. A value of null means that there are no more result sets to be returned.

getUpdateCount

Format:

```
public abstract int getUpdateCount() throws SQLException
```

Returns the number of rows that were updated by the last SQL operation that was executed using this context.

getWarnings

Format:

```
public SQLWarning getWarnings()
```

Returns the first warning that was reported by the last SQL operation that was executed using this context. Subsequent warnings are chained to the first warning.

Use this method to retrieve positive SQLCODEs.

setMaxFieldSize
Format:

```
public void setMaxFieldSize(int max)
```

Specifies the maximum number of bytes that are returned for any character column in queries that use the given execution context. The default is 0, which means that the maximum number of bytes is unlimited.

setMaxRows
Format:

```
public void setMaxRows(int max)
```

Specifies the maximum number of rows that are returned for any query that uses the given execution context. The default is 0, which means that the maximum number of rows returned is unlimited.

sqlj.runtime.ConnectionContext interface

`sqlj.runtime.ConnectionContext` is an interface that SQLJ implements when you execute a connection declaration clause and thereby create a connection context class.

Suppose that you declare a connection named Ctx. You can then use the following methods to determine or change the default context.

getDefaultContext
Format:

```
public static Ctx getDefaultContext()
```

Returns the default connection context object for the Ctx class.

SetDefaultContext
Format:

```
public static void Ctx setDefaultContext(Ctx default-context)
```

Sets the default connection context object for the Ctx class.

sqlj.runtime.ForUpdate interface

Implement the `sqlj.runtime.ForUpdate` interface for positioned UPDATE or DELETE operations. You implement `sqlj.runtime.ForUpdate` in an SQLJ iterator declaration clause. For positioned UPDATE and DELETE operations, you must declare an iterator in one source file and use the iterator in a different source file. See “Using iterators for positioned UPDATE and DELETE operations” on page 27 for more information.

sqlj.runtime.NamedIterator interface

`sqlj.runtime.NamedIterator` is an interface that SQLJ implements when you declare a named iterator. When you declare an instance of a named iterator, SQLJ creates an accessor method for each column in the expected 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.

In addition to the accessor methods, SQLJ generates the following methods that you can invoke in your SQLJ application.

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.

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

`sqlj.runtime.PositionedIterator` is an interface that SQLJ implements when you declare a positioned iterator. After you declare and create an instance of a positioned iterator, you can use the following method.

endFetch

Format:

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

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

sqlj.runtime.ResultSetIterator interface

`sqlj.runtime.ResultSetIterator` is an interface that SQLJ implements when you declare an iterator. After you declare and create an instance of an iterator, you can use the following methods.

clearWarnings

Format:

```
public abstract void clearWarnings() throws SQLException
```

Returns null until a new warning is reported for this iterator.

close

Format:

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

Releases database resources that the iterator uses.

getResultSet

Format:

```
public abstract ResultSet getResultSet() throws SQLException
```

Returns a JDBC result set representation of an SQLJ iterator.

getWarnings

Format:

```
public abstract SQLWarning getWarnings() throws SQLException
```

Returns the first warning that is reported by calls on this iterator. Subsequent iterator warnings are be chained to this SQLWarning. The warning chain is automatically cleared each time a new row is read.

isClosed

Format:

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

Returns a value of `true` if the `close` method has 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.

Appendix B. 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. This appendix outlines those differences and explains what you need to do about them.

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.

For the VisualAge for Java environment, you need to specify a fully-qualified path name.

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 OS/390 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. For VisualAge for Java SQLJ or JDBC applications that run in a CICS environment, large JDBC profiles can degrade performance.

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 SQLJ/JDBC driver uses to process JDBC result sets. The default cursor properties file, db2jdbc.cursors, defines 125 cursors with hold and 125 cursors without hold. 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 SQLJ or JDBC applications in a CICS environment, the way that you specify configuration information differs depending on whether you run in the JVM environment or the VisualAge for Java environment.

For CICS Java programs that run in the JVM environment, you specify the environment variables that are listed in "Setting environment variables" on page 91 in the DFHJVM member of the SDFHENV data set.

For CICS Java programs that run in the VisualAge for Java environment, the environment variables that are listed in "Setting environment variables" on page 91 are not used. Put DB2 code in a PDSE that you specify in the CICS DFHRPL concatenation. If you use an SQLJ/JDBC run-time properties file other than the default file, bind the properties file into its own DLL in a PDSE, and include the name of that PDSE in the CICS DFHRPL concatenation. Because you cannot use an environment variable to name the run-time properties file, you must use the default name for the file: `db2sqljjdbc.properties`. See "Installing VisualAge for Java support for JDBC and SQLJ" on page 97 for more information on binding the run-time properties file.

Choosing VisualAge for Java bind parameters for better performance

To improve performance of an SQLJ or JDBC program that runs in the CICS and VisualAge for Java environment, specify this parameter when you execute the hpj command:

```
-lerunopts="(envar('IBMHPJ_OPTS=-Xskipgc'))"
```

This parameter causes Language Environment® to turn off Java garbage collection routines at run time. For more information on recommended hpj options for CICS, see *CICS Application Programming 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.

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 SQLJ/JDBC driver 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 SQLJ/JDBC driver, 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, the trace output goes to different locations, depending on whether the application runs in a JVM or under VisualAge for Java.

- The program runs in a JVM

The output goes to *trace-file* (the binary trace) and *trace-file*.JTRACE (the readable trace), as described in “Formatting trace data” on page 36.

- The program runs in the ET/390 Java execution environment

  The trace data that is in a proprietary, binary format goes to *trace-file*, as described in “Formatting trace data” on page 36. The readable trace data is routed by Language Environment to the CICS transient data destination CESE.

Notices

Programming interface information

This book is intended to help the customer write applications that use Java to access IBM DB2 for OS/390 and z/OS servers. This book primarily documents General-use Programming Interface and Associated Guidance Information provided by IBM DATABASE 2 Universal Database Server for OS/390 and z/OS (DB2 for OS/390 and z/OS).

General-use programming interfaces allow the customer to write programs that obtain the services of DB2 for OS/390 and z/OS.

Trademarks

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. A complete list of DB2 abend reason codes and their explanations is contained in *DB2 Messages and Codes*.

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 path. The path that is used to locate data that is specified in SQL statements. An access path can be indexed or sequential.

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*.

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.

ambiguous cursor. A database cursor that is not defined with the FOR FETCH ONLY clause or the FOR UPDATE OF clause, is not defined on a read-only result table, is not the target of a WHERE CURRENT clause on an SQL UPDATE or DELETE statement, and is in a plan or package that contains either PREPARE or EXECUTE IMMEDIATE SQL statements.

API. Application programming interface.

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.

ASCII. An encoding scheme that is used to represent strings in many environments, typically on PCs and workstations. Contrast with *EBCDIC* and *Unicode*.

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.

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.

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*.

binary integer. A basic data type that can be further classified as small integer or large integer.

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, because they could not be bound during the bind process, and VALIDATE(RUN) was specified.
>
> **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 is not associated with a coded character set.

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 only applies to cursors that do not update data.

BMP. Batch Message Processing (IMS).

built-in function. A function that DB2 supplies. Contrast with *user-defined function*.

C

CAF. Call attachment facility.

call attachment facility (CAF). A DB2 attachment facility for application programs that run in TSO or MVS batch. The CAF is an alternative to the DSN command processor and provides greater control over the execution environment.

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.

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.

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 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 constraints are also called *table check constraints*.

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 some SQL statements because of rows that violate referential constraints, table check constraints, or both.

CICS. Represents (in this publication) one of the following products:
CICS Transaction Server for OS/390: Customer Information Control System Transaction Server for OS/390
CICS/ESA: Customer Information Control System/Enterprise Systems Architecture
CICS/MVS: Customer Information Control System/Multiple Virtual Storage

CICS attachment facility. A DB2 subcomponent that uses the MVS subsystem interface (SSI) and cross storage linkage to process requests from CICS to DB2 and to coordinate resource commitment.

clause. In SQL, a distinct part of a statement, such as a SELECT clause or a WHERE clause.

client. See *requester*.

CLIST. Command list. A language for performing TSO tasks.

clustering index. An index that determines how rows are physically ordered in a table space.

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.** A set of assignments of characters
| to code points. In EBCDIC, for example, the
| character 'A' is assigned code point X'C1', 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.

collection. A group of packages that have the same qualifier.

column function. An operation that derives its result by using values from one or more rows. Contrast with *scalar function*.

command. A DB2 operator command or a DSN subcommand. A command is distinct from an SQL statement.

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.

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 =, >, <) that is used to specify a relationship between two values.

composite key. An ordered set of key columns of the same table.

concurrency. The shared use of resources by more than one application process at the same time.

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.

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*, *table check constraint*, and *uniqueness constraint*.

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 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.

CP. See *central processor (CP)*.

CS. Cursor stability.

current data. Data within a host structure that is current with (identical to) the data within the base table.

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.

D

DASD. Direct access storage device.

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 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, and relationships.

database management system (DBMS). A software system that controls the creation, organization, and modification of a database and the access to the data 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.

DATABASE 2 Interactive (DB2I). The DB2 facility that provides for the execution of SQL statements, DB2 (operator) commands, programmer commands, and utility invocation.

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 Language/I (DL/I). The IMS data manipulation language; a common high-level interface between a user application and IMS.

data partition. A VSAM data set that is contained within a partitioned table space.

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 type. An attribute of columns, literals, host variables, special registers, and the results of functions and expressions.

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.

DBCS. Double-byte character set.

DBD. Database descriptor.

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 allowing a user 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.

DB2I. DATABASE 2 Interactive.

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.

default value. A predetermined value, attribute, or option that is assumed when no other is explicitly specified.

degree of parallelism. The number of concurrently executed operations that are initiated to process a query.

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 syntax diagrams.

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 *parent row*, *parent table*, *parent table space*.

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*.

direct access storage device (DASD). A device in which access time is independent of the location of the data.

distributed data facility (DDF). A set of DB2 components through which DB2 communicates with another RDBMS.

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.

DL/I. Data Language/I.

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*.

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 *date duration*, *labeled duration*, and *time duration*.

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.

E

EBCDIC. Extended binary coded decimal interchange code. An encoding scheme that is used to represent character data in the OS/390,

| MVS, VM, VSE, and OS/400® environments.
| Contrast with *ASCII* and *Unicode*.

embedded SQL. SQL statements that are coded within an application program. See *static SQL*.

equijoin. A join operation in which the join-condition has the form *expression* = *expression*.

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 (').

EUR. IBM European Standards.

execution context. In SQLJ, a Java object that can be used to control the execution of SQL statements.

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.

expression. An operand or a collection of operators and operands that yields a single value.

F

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.

fixed-length string. A character or graphic string whose length is specified and cannot be changed. Contrast with *varying-length string*.

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.

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*.

function. A mapping, embodied as a program (the function body), invocable by means of zero or more input values (arguments), to a single value (the result). See also *column function* and *scalar function*.

Functions can be user-defined, built-in, or generated by DB2. (See *built-in function*, *cast function*, *external function*, *sourced function*, *SQL function*, and *user-defined function*.)

G

global lock. A lock that provides concurrency control within and among DB2 subsystems. The scope of the lock is across all the 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.

graphic string. A sequence of DBCS characters.

gross lock. The *shared*, *update*, or *exclusive* mode locks on a table, partition, or table space.

group name. The MVS 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.

H

help panel. A screen of information presenting tutorial text to assist a user at the terminal.

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.

I

IFP. IMS Fast Path.

IMS. Information Management System.

IMS attachment facility. A DB2 subcomponent that uses MVS subsystem interface (SSI) protocols and cross-memory linkage to process requests from IMS to DB2 and to coordinate resource commitment.

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 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 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.

inner join. The result of a join operation that includes only the matched rows of both tables being joined. See also *join*.

Interactive System Productivity Facility (ISPF). An IBM licensed program that provides interactive dialog services.

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.

internal resource lock manager (IRLM). An MVS subsystem that DB2 uses to control communication and database locking.

IRLM. Internal resource lock manager.

ISO. International Standards Organization.

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/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 Standards Committee (JISC). An organization that issues standards for coding character sets.

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.

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.

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).

key. A column or an ordered collection of columns identified in the description of a table, index, or referential constraint.

L

labeled duration. A number that represents a duration of years, months, days, hours, minutes, seconds, or microseconds.

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*.

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.

L-lock. Logical lock.

load module. A program unit that is suitable for loading into main storage for execution. The output of a linkage editor.

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*.

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 RDBMS 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.

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 parent. For explicit hierarchical locking, a lock that is held on a resource that has child locks that are lower in the hierarchy; usually the table space or partition intent locks are the parent locks.

lock promotion. The process of changing the size or mode of a DB2 lock to a higher level.

lock size. The amount of data 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.

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)*.

logical unit. 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.

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.

M

| **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.

mixed data string. A character string that can contain both single-byte and double-byte characters.

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 fails, that DB2 subsystem's modify locks are converted to retained locks.

MPP. Message processing program (in IMS).

| **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*.

multisite update. Distributed relational database processing in which data is updated in more than one location within a single unit of work.

MVS. Multiple Virtual Storage.

MVS/ESA™. Multiple Virtual Storage/Enterprise Systems Architecture.

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.

nonpartitioning index. Any index that is not a partitioning index.

NUL. In C, a single character that denotes the end of the string.

null. A special value that indicates the absence of information.

NUL-terminated host variable. A varying-length host variable in which the end of the data is indicated by the presence of a NUL terminator.

NUL terminator. In C, the value that indicates the end of a string. For character strings, the NUL terminator is X'00'.

O

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.

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*.

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*.

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.

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.

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.

partner logical unit. An access point in the SNA network that is connected to the local DB2 subsystem by way of a VTAM conversation.

PCT. Program control table (in CICS).

piece. A data set of a nonpartitioned page set.

physical consistency. The state of a page that is not in a partially changed state.

physical lock (P-lock). A lock type 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 *negotiable lock*.

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.

P-lock. Physical lock.

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*.

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 definitions.

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.

prepared SQL statement. A named object that is the executable form of an SQL statement that has been processed by the PREPARE statement.

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.

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*.

Q

QMF™. Query Management Facility.

R

RCT. Resource control table (in CICS attachment facility).

RDB. Relational database.

RDBMS. Relational database management system.

RDBNAM. Relational database name.

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.

record. The storage representation of a row or other data.

recovery. The process of rebuilding databases after a system failure.

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 upon which the referential constraints are defined.

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.

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 subsystem. Any RDBMS, 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 MVS system.

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 referenced by the program cannot be changed by other programs until the program reaches a commit point.

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 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 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.

result set. The set of rows that a stored procedure returns to a client application.

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.

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.

rollback. The process of restoring data changed by SQL statements to the state at its last commit point. All locks are freed. Contrast with *commit*.

row. The horizontal component of a table. A row consists of a sequence of values, one for each column of the table.

RRSAF. Recoverable Resource Manager Services attachment facility. RRSAF is a DB2 subcomponent that uses OS/390 Transaction Management and Recoverable Resource Manager Services to coordinate resource commitment between DB2 and all other resource managers that also use OS/390 RRS in an OS/390 system.

RS. Read stability.

S

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 *column 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.

search condition. A criterion for selecting rows from a table. A search condition consists of one or more predicates.

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.

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. Three common types of server-side programs are Common Gateway Interface (CGI) programs, Web server API programs, and Java servlets.

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*.

| **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 definition.

source program. A set of host language statements and SQL statements that is processed by an SQL precompiler.

space. A sequence of one or more blank characters.

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.

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 Processor Using File Input (SPUFI). SQL Processor Using File Input. 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.

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*.

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*.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

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 ORDER BY clause, 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).

sync point. See *commit point*.

synonym. In SQL, an alternative name for a table or view. Synonyms can be used only to refer to objects at the subsystem in which the synonym is defined.

system administrator. The person at a computer installation who designs, controls, and manages the use of the computer system.

system conversation. The conversation that two DB2 subsystems must establish to process system messages before any distributed processing can begin.

| **system-directed connection.** A connection that
| an RDBMS manages by processing SQL
| statements with three-part names.

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 check constraint. A user-defined constraint that specifies the values that specific columns of a base table can contain.

table space. A page set that is used to store the records in one or more tables.

task control block (TCB). A control block that is used to communicate information about tasks within an address space that are connected to DB2. An address space can support many task connections (as many as one per task), but only one address space connection. See also *address space connection*.

TCB. Task control block (in MVS).

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*.

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.

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.

transaction lock. A lock that is used to control concurrent execution of SQL statements.

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.

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 7, type 1 indexes are no longer supported.

type 2 indexes. Indexes that are created on a release of DB2 after Version 6 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.

| **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 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.

URL. Uniform resource locator.

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*.

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 for OS/390 and 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 for OS/390 and z/OS supports UTF-16 in graphic data fields.

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*.

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.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed- and varying-length records on direct access 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.

Virtual Telecommunications Access Method (VTAM). An IBM licensed program that controls communication and the flow of data in an SNA network.

VSAM. Virtual storage access method.

VTAM. Virtual Telecommunication Access Method (in MVS).

Z

z/OS. An operating system for the eServer product line that supports 64-bit real storage.

Bibliography

DB2 Universal Database Server for OS/390 and z/OS Version 7 product libraries:

DB2 for OS/390 and z/OS

- *DB2 Administration Guide, SC26-9931*
- *DB2 Application Programming and SQL Guide, SC26-9933*
- *DB2 Application Programming Guide and Reference for Java, SC26-9932*
- *DB2 Command Reference, SC26-9934*
- *DB2 Data Sharing: Planning and Administration, SC26-9935*
- *DB2 Data Sharing Quick Reference Card, SX26-3846*
- *DB2 Diagnosis Guide and Reference, LY37-3740*
- *DB2 Diagnostic Quick Reference Card, LY37-3741*
- *DB2 Image, Audio, and Video Extenders Administration and Programming, SC26-9947*
- *DB2 Installation Guide, GC26-9936*
- *DB2 Licensed Program Specifications, GC26-9938*
- *DB2 Master Index, SC26-9939*
- *DB2 Messages and Codes, GC26-9940*
- *DB2 ODBC Guide and Reference, SC26-9941*
- *DB2 Reference for Remote DRDA Requesters and Servers, SC26-9942*
- *DB2 Reference Summary, SX26-3847*
- *DB2 Release Planning Guide, SC26-9943*
- *DB2 SQL Reference, SC26-9944*
- *DB2 Text Extender Administration and Programming, SC26-9948*
- *DB2 Utility Guide and Reference, SC26-9945*
- *DB2 What's New? GC26-9946*
- *DB2 XML Extender for OS/390 and z/OS Administration and Programming*, SC27-9949
- *DB2 Program Directory, GI10-8182*

DB2 Administration Tool

- *DB2 Administration Tool for OS/390 and z/OS User's Guide, SC26-9847*

DB2 Buffer Pool Tool

- *DB2 Buffer Pool Tool for OS/390 and z/OS User's Guide and Reference, SC26-9306*

DB2 DataPropagator™

- *DB2 UDB Replication Guide and Reference, SC26-9920*

Net.Data®

The following books are available at this Web site: http://www.ibm.com/software/net.data/library.html

- *Net.Data Library: Administration and Programming Guide for OS/390 and z/OS*
- *Net.Data Library: Language Environment Interface Reference*
- *Net.Data Library: Messages and Codes*
- *Net.Data Library: Reference*

DB2 PM for OS/390

- *DB2 PM for OS/390 Batch User's Guide, SC27-0857*
- *DB2 PM for OS/390 Command Reference, SC27-0855*
- *DB2 PM for OS/390 Data Collector Application Programming Interface Guide, SC27-0861*
- *DB2 PM for OS/390 General Information, GC27-0852*
- *DB2 PM for OS/390 Installation and Customization, SC27-0860*
- *DB2 PM for OS/390 Messages, SC27-0856*
- *DB2 PM for OS/390 Online Monitor User's Guide, SC27-0858*
- *DB2 PM for OS/390 Report Reference Volume 1, SC27-0853*
- *DB2 PM for OS/390 Report Reference Volume 2, SC27-0854*
- *DB2 PM for OS/390 Using the Workstation Online Monitor, SC27-0859*

- *DB2 PM for OS/390 Program Directory, GI10-8223*

Query Management Facility (QMF)
- *Query Management Facility: Developing QMF Applications, SC26-9579*
- *Query Management Facility: Getting Started with QMF on Windows, SC26-9582*
- *Query Management Facility: High Peformance Option User’s Guide for OS/390 and z/OS, SC26-9581*
- *Query Management Facility: Installing and Managing QMF on OS/390 and z/OS, GC26-9575*
- *Query Management Facility: Installing and Managing QMF on Windows, GC26-9583*
- *Query Management Facility: Introducing QMF, GC26-9576*
- *Query Management Facility: Messages and Codes, GC26-9580*
- *Query Management Facility: Reference, SC26-9577*
- *Query Management Facility: Using QMF, SC26-9578*

Ada/370
- *IBM Ada/370 Language Reference, SC09-1297*
- *IBM Ada/370 Programmer's Guide, SC09-1414*
- *IBM Ada/370 SQL Module Processor for DB2 Database Manager User's Guide, SC09-1450*

APL2®
- *APL2 Programming Guide, SH21-1072*
- *APL2 Programming: Language Reference, SH21-1061*
- *APL2 Programming: Using Structured Query Language (SQL), SH21-1057*

AS/400®

The following books are available at this Web site: www.as400.ibm.com/infocenter
- *DB2 Universal Database for AS/400 Database Programming*
- *DB2 Universal Database for AS/400 Performance and Query Optimization*
- *DB2 Universal Database for AS/400 Distributed Data Management*
- *DB2 Universal Database for AS/400 Distributed Data Programming*
- *DB2 Universal Database for AS/400 SQL Programming Concepts*
- *DB2 Universal Database for AS/400 SQL Programming with Host Languages*
- *DB2 Universal Database for AS/400 SQL Reference*

BASIC
- *IBM BASIC/MVS Language Reference, GC26-4026*
- *IBM BASIC/MVS Programming Guide, SC26-4027*

BookManager® READ/MVS
- *BookManager READ/MVS V1R3: Installation Planning & Customization, SC38-2035*

SAA® AD/Cycle® C/370™
- *IBM SAA AD/Cycle C/370 Programming Guide, SC09-1841*
- *IBM SAA AD/Cycle C/370 Programming Guide for Language Environment/370, SC09-1840*
- *IBM SAA AD/Cycle C/370 User's Guide, SC09-1763*
- *SAA CPI C Reference, SC09-1308*

Character Data Representation Architecture
- *Character Data Representation Architecture Overview, GC09-2207*
- *Character Data Representation Architecture Reference and Registry, SC09-2190*

CICS/ESA
- *CICS/ESA Application Programming Guide, SC33-1169*
- *CICS External Interfaces Guide, SC33-1944*
- *CICS for MVS/ESA Application Programming Reference, SC33-1170*
- *CICS for MVS/ESA CICS-RACF Security Guide, SC33-1185*
- *CICS for MVS/ESA CICS-Supplied Transactions, SC33-1168*
- *CICS for MVS/ESA Customization Guide, SC33-1165*
- *CICS for MVS/ESA Data Areas, LY33-6083*
- *CICS for MVS/ESA Installation Guide, SC33-1163*
- *CICS for MVS/ESA Intercommunication Guide, SC33-1181*

- *CICS for MVS/ESA Messages and Codes, GC33-1177*
- *CICS for MVS/ESA Operations and Utilities Guide, SC33-1167*
- *CICS/ESA Performance Guide, SC33-1183*
- *CICS/ESA Problem Determination Guide, SC33-1176*
- *CICS for MVS/ESA Resource Definition Guide, SC33-1166*
- *CICS for MVS/ESA System Definition Guide, SC33-1164*
- *CICS for MVS/ESA System Programming Reference, GC33-1171*

CICS Transaction Server for OS/390
- *CICS Application Programming Guide, SC33-1687*
- *CICS External Interfaces Guide, SC33-1703*
- *CICS DB2 Guide, SC33-1939*
- *CICS Resource Definition Guide, SC33-1684*

IBM C/C++ for MVS/ESA
- *IBM C/C++ for MVS/ESA Library Reference, SC09-1995*
- *IBM C/C++ for MVS/ESA Programming Guide, SC09-1994*

IBM COBOL
- *IBM COBOL Language Reference, SC26-4769*
- *IBM COBOL for MVS & VM Programming Guide, SC26-4767*

IBM COBOL for OS/390 & VM Programming Guide, SC26-9049

Conversion Guide
- *IMS-DB and DB2 Migration and Coexistence Guide, GH21-1083*

Cooperative Development Environment
- *CoOperative Development Environment/370: Debug Tool, SC09-1623*

DataPropagator NonRelational
- *DataPropagator NonRelational MVS/ESA Administration Guide, SH19-5036*
- *DataPropagator NonRelational MVS/ESA Reference, SH19-5039*

Data Facility Data Set Services
- *Data Facility Data Set Services: User's Guide and Reference, SC26-4388*

Database Design
- *DB2 Design and Development Guide* by Gabrielle Wiorkowski and David Kull, Addison Wesley, ISBN 0-20158-049-7
- *Handbook of Relational Database Design* by C. Fleming and B. Von Halle, Addison Wesley, ISBN 0-20111-434-8

DataHub®
- *IBM DataHub General Information, GC26-4874*

Data Refresher
- *Data Refresher Relational Extract Manager for MVS* GI10-9927

DB2 Connect®
- *DB2 Connect Enterprise Edition for OS/2 and Windows: Quick Beginnings, GC09-2953*
- *DB2 Connect Enterprise Edition for UNIX: Quick Beginnings, GC09-2952*
- *DB2 Connect Personal Edition Quick Beginnings, GC09-2967*
- *DB2 Connect User's Guide, SC09-2954*

DB2 Red Books
- *DB2 UDB Server for OS/390 Version 6 Technical Update, SG24-6108-00*

DB2 Server for VSE & VM
- *DB2 Server for VM: DBS Utility, SC09-2394*
- *DB2 Server for VSE: DBS Utility, SC09-2395*

DB2 Universal Database for UNIX®, Windows®, OS/2®
- *DB2 UDB Administration Guide: Planning, SC09-2946*
- *DB2 UDB Administration Guide: Implementation, SC09-2944*
- *DB2 UDB Administration Guide: Performance, SC09-2945*
- *DB2 UDB Administrative API Reference, SC09-2947*
- *DB2 UDB Application Building Guide, SC09-2948*
- *DB2 UDB Application Development Guide, SC09-2949*

- *DB2 UDB CLI Guide and Reference, SC09-2950*
- *DB2 UDB SQL Getting Started, SC09-2973*
- *DB2 UDB SQL Reference Volume 1, SC09-2974*
- *DB2 UDB SQL Reference Volume 2, SC09-2975*

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 DFSMS/MVS®, DFSMSdfp™, DFSMSdss™, DFSMShsm™, and MVS/DFP™.
- *DFSMS/MVS: Access Method Services for the Integrated Catalog, SC26-4906*
- *DFSMS/MVS: Access Method Services for VSAM Catalogs, SC26-4905*
- *DFSMS/MVS: Administration Reference for DFSMSdss, SC26-4929*
- *DFSMS/MVS: DFSMShsm Managing Your Own Data, SH21-1077*
- *DFSMS/MVS: Diagnosis Reference for DFSMSdfp, LY27-9606*
- *DFSMS/MVS Storage Management Library: Implementing System-Managed Storage, SC26–3123*
- *DFSMS/MVS: Macro Instructions for Data Sets, SC26-4913*
- *DFSMS/MVS: Managing Catalogs, SC26-4914*
- *DFSMS/MVS: Program Management, SC26-4916*
- *DFSMS/MVS: Storage Administration Reference for DFSMSdfp, SC26-4920*
- *DFSMS/MVS: Using Advanced Services, SC26-4921*
- *DFSMS/MVS: Utilities, SC26-4926*
- *MVS/DFP: Using Data Sets, SC26-4749*

DFSORT™
- *DFSORT Application Programming: Guide, SC33-4035*

Distributed Relational Database Architecture™
- *Data Stream and OPA Reference, SC31-6806*
- *IBM SQL Reference, SC26-8416*
- *Open Group Technical Standard*

  The Open Group presently makes the following DRDA books available through its Web site at: www.opengroup.org
  - *DRDA Version 2 Vol. 1: Distributed Relational Database Architecture (DRDA)*
  - *DRDA Version 2 Vol. 2: Formatted Data Object Content Architecture*
  - *DRDA Version 2 Vol. 3: Distributed Data Management Architecture*

Domain Name System
- *DNS and BIND, Third Edition, Paul Albitz and Cricket Liu, O'Reilly, ISBN 1-56592-512-2*

Education
- *IBM Dictionary of Computing, McGraw-Hill, ISBN 0-07031-489-6*
- *1999 IBM All-in-One Education and Training Catalog, GR23-8105*

Enterprise System/9000® and Enterprise System/3090™
- *Enterprise System/9000 and Enterprise System/3090 Processor Resource/System Manager Planning Guide, GA22-7123*

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*

Parallel Sysplex® Library
- *OS/390 Parallel Sysplex Application Migration, GC28-1863*
- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1862*
- *OS/390 Parallel Sysplex Overview: An Introduction to Data Sharing and Parallelism, GC28-1860*
- *OS/390 Parallel Sysplex Systems Management, GC28-1861*
- *OS/390 Parallel Sysplex Test Report, GC28-1963*
- *System/390 9672/9674 System Overview, GA22-7148*

ICSF/MVS

- *ICSF/MVS General Information, GC23-0093*

IMS
- *IMS Batch Terminal Simulator General Information, GH20-5522*
- *IMS Administration Guide: System, SC26-9420*
- *IMS Administration Guide: Transaction Manager, SC26-9421*
- *IMS Application Programming: Database Manager, SC26-9422*
- *IMS Application Programming: Design Guide, SC26-9423*
- *IMS Application Programming: Transaction Manager, SC26-9425*
- *IMS Command Reference, SC26-9436*
- *IMS Customization Guide, SC26-9427*
- *IMS Install Volume 1: Installation and Verification, GC26-9429*
- *IMS Install Volume 2: System Definition and Tailoring, GC26-9430*
- *IMS Messages and Codes, GC27-1120*
- *IMS Utilities Reference: System, SC26-9441*

ISPF
- *ISPF V4 Dialog Developer's Guide and Reference, SC34-4486*
- *ISPF V4 Messages and Codes, SC34-4450*
- *ISPF V4 Planning and Customizing, SC34-4443*
- *ISPF V4 User's Guide, SC34-4484*

Language Environment
- *Debug Tool User's Guide and Reference, SC09-2137*

National Language Support
- *IBM National Language Support Reference Manual Volume 2, SE09-8002*

NetView®
- *NetView Installation and Administration Guide, SC31-8043*
- *NetView User's Guide, SC31-8056*

Microsoft® ODBC
- *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference, Microsoft Press, ISBN 1-57231-516-4*

OS/390
- *OS/390 C/C++ Programming Guide, SC09-2362*
- *OS/390 C/C++ Run-Time Library Reference, SC28-1663*
- *OS/390 C/C++ User's Guide, SC09-2361*
- *OS/390 eNetwork Communications Server: IP Configuration, SC31-8513*
- *OS/390 Hardware Configuration Definition Planning, GC28-1750*
- *OS/390 Information Roadmap, GC28-1727*
- *OS/390 Introduction and Release Guide, GC28-1725*
- *OS/390 JES2 Initialization and Tuning Guide, SC28-1791*
- *OS/390 JES3 Initialization and Tuning Guide, SC28-1802*
- *OS/390 Language Environment for OS/390 & VM Concepts Guide, GC28-1945*
- *OS/390 Language Environment for OS/390 & VM Customization, SC28-1941*
- *OS/390 Language Environment for OS/390 & VM Debugging Guide, SC28-1942*
- *OS/390 Language Environment for OS/390 & VM Programming Guide, SC28-1939*
- *OS/390 Language Environment for OS/390 & VM Programming Reference, SC28-1940*
- *OS/390 MVS Diagnosis: Procedures, LY28-1082*
- *OS/390 MVS Diagnosis: Reference, SY28-1084*
- *OS/390 MVS Diagnosis: Tools and Service Aids, LY28-1085*
- *OS/390 MVS Initialization and Tuning Guide, SC28-1751*
- *OS/390 MVS Initialization and Tuning Reference, SC28-1752*
- *OS/390 MVS Installation Exits, SC28-1753*
- *OS/390 MVS JCL Reference, GC28-1757*
- *OS/390 MVS JCL User's Guide, GC28-1758*
- *OS/390 MVS Planning: Global Resource Serialization, GC28-1759*
- *OS/390 MVS Planning: Operations, GC28-1760*
- *OS/390 MVS Planning: Workload Management, GC28-1761*
- *OS/390 MVS Programming: Assembler Services Guide, GC28-1762*
- *OS/390 MVS Programming: Assembler Services Reference, GC28-1910*
- *OS/390 MVS Programming: Authorized Assembler Services Guide, GC28-1763*

- *OS/390 MVS Programming: Authorized Assembler Services Reference, Volumes 1-4, GC28-1764, GC28-1765, GC28-1766, GC28-1767*
- *OS/390 MVS Programming: Callable Services for High-Level Languages, GC28-1768*
- *OS/390 MVS Programming: Extended Addressability Guide, GC28-1769*
- *OS/390 MVS Programming: Sysplex Services Guide, GC28-1771*
- *OS/390 MVS Programming: Sysplex Services Reference, GC28-1772*
- *OS/390 MVS Programming: Workload Management Services, GC28-1773*
- *OS/390 MVS Routing and Descriptor Codes, GC28-1778*
- *OS/390 MVS Setting Up a Sysplex, GC28-1779*
- *OS/390 MVS System Codes, GC28-1780*
- *OS/390 MVS System Commands, GC28-1781*
- *OS/390 MVS System Messages Volume 1, GC28-1784*
- *OS/390 MVS System Messages Volume 2, GC28-1785*
- *OS/390 MVS System Messages Volume 3, GC28-1786*
- *OS/390 MVS System Messages Volume 4, GC28-1787*
- *OS/390 MVS System Messages Volume 5, GC28-1788*
- *OS/390 MVS Using the Subsystem Interface, SC28-1789*
- *OS/390 Security Server External Security Interface (RACROUTE) Macro Reference, GC28-1922*
- *OS/390 Security Server (RACF) Auditor's Guide, SC28-1916*
- *OS/390 Security Server (RACF) Command Language Reference, SC28-1919*
- *OS/390 Security Server (RACF) General User's Guide, SC28-1917*
- *OS/390 Security Server (RACF) Introduction, GC28-1912*
- *OS/390 Security Server (RACF) Macros and Interfaces, SK2T-6700 (OS/390 Collection Kit), SK27-2180 (OS/390 Security Server Information Package)*
- *OS/390 Security Server (RACF) Security Administrator's Guide, SC28-1915*
- *OS/390 Security Server (RACF) System Programmer's Guide, SC28-1913*
- *OS/390 SMP/E Reference, SC28-1806*
- *OS/390 SMP/E User's Guide, SC28-1740*
- *OS/390 Support for Unicode: Using Conversion Services, SC33-7050*
- *OS/390 RMF User's Guide, SC28-1949*
- *OS/390 TSO/E CLISTS, SC28-1973*
- *OS/390 TSO/E Command Reference, SC28-1969*
- *OS/390 TSO/E Customization, SC28-1965*
- *OS/390 TSO/E Messages, GC28-1978*
- *OS/390 TSO/E Programming Guide, SC28-1970*
- *OS/390 TSO/E Programming Services, SC28-1971*
- *OS/390 TSO/E REXX Reference, SC28-1975*
- *OS/390 TSO/E User's Guide, SC28-1968*
- *OS/390 DCE Administration Guide, SC28-1584*
- *OS/390 DCE Introduction, GC28-1581*
- *OS/390 DCE Messages and Codes, SC28-1591*
- *OS/390 UNIX System Services Command Reference, SC28-1892*
- *OS/390 UNIX System Services Messages and Codes, SC28-1908*
- *OS/390 UNIX System Services Planning, SC28-1890*
- *OS/390 UNIX System Services User's Guide, SC28-1891*
- *OS/390 UNIX System Services Programming: Assembler Callable Services Reference, SC28-1899*

IBM Enterprise PL/I for z/OS and OS/390

- *IBM Enterprise PL/I for z/OS and OS/390 Language Reference, SC26-9476*
- *IBM Enterprise PL/I for z/OS and OS/390 Programming Guide, SC26-9473*

OS PL/I

- *OS PL/I Programming Language Reference, SC26-4308*
- *OS PL/I Programming Guide, SC26-4307*

Prolog

- *IBM SAA AD/Cycle Prolog/MVS & VM Programmer's Guide, SH19-6892*

RAMAC® and Enterprise Storage Server™

- *IBM RAMAC Virtual Array, SG24-4951*
- *RAMAC Virtual Array: Implementing Peer-to-Peer Remote Copy, SG24-5338*

- *Enterprise Storage Server Introduction and Planning, GC26-7294*

Remote Recovery Data Facility
- *Remote Recovery Data Facility Program Description and Operations, LY37-3710*

Storage Management
- *DFSMS/MVS Storage Management Library: Implementing System-Managed Storage, SC26-3123*
- *MVS/ESA Storage Management Library: Leading a Storage Administration Group, SC26-3126*
- *MVS/ESA Storage Management Library: Managing Data, SC26-3124*
- *MVS/ESA Storage Management Library: Managing Storage Groups, SC26-3125*
- *MVS Storage Management Library: Storage Management Subsystem Migration Planning Guide, SC26-4659*

System/370™ and System/390
- *ESA/370 Principles of Operation, SA22-7200*
- *ESA/390 Principles of Operation, SA22-7201*
- *System/390 MVS Sysplex Hardware and Software Migration, GC28-1210*

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*

VS COBOL II
- *VS COBOL II Application Programming Guide for MVS and CMS, SC26-4045*
- *VS COBOL II Application Programming: Language Reference, GC26-4047*
- *VS COBOL II Installation and Customization for MVS, SC26-4048*

VS Fortran
- *VS Fortran Version 2: Language and Library Reference, SC26-4221*
- *VS Fortran Version 2: Programming Guide for CMS and MVS, SC26-4222*

VTAM
- *Planning for NetView, NCP, and VTAM, SC31-8063*
- *VTAM for MVS/ESA Diagnosis, LY43-0069*
- *VTAM for MVS/ESA Messages and Codes, SC31-6546*
- *VTAM for MVS/ESA Network Implementation Guide, SC31-6548*
- *VTAM for MVS/ESA Operation, SC31-6549*
- *VTAM for MVS/ESA Programming, SC31-6550*
- *VTAM for MVS/ESA Programming for LU 6.2, SC31-6551*
- *VTAM for MVS/ESA Resource Definition Reference, SC31-6552*

Index

A

API
- JDBC 9

application
- Java, running 9
- JDBC support 1
- SQLJ support 15

assignment clause
- SQLJ 50

attachment facilities
- description 102
- RRSAF 103

authorization IDs, establishing 102

B

binding a plan
- SQLJ 75

C

case sensitivity
- SQLJ 17

CICS
- abends 115
- attaching to DB2 114
- autoCommit default 115
- closing JDBC connection 115
- Connection with default URL 115
- db2genJDBC parameters 113
- environment variables 114
- number of cursors 114
- run-time properties file 113
- running traces 115
- special considerations 113
- VisualAge for Java bind parameters 114

comment
- SQLJ 17

compiled Java stored procedure
- program preparation 78

configuring
- JDBC 92
- SQLJ 92

connecting to a data source
- multiple context support 103
- SQLJ 19

connection declaration clause
- SQLJ 44

connection object 103

connection pooling
- overview 105

connection sharing 104

context clause
- SQLJ 47

creating a DBRM
- SQLJ 72

customizing a serialized profile
- SQLJ 72

customizing Java environment 92

D

data source
- connecting 2
- connecting using JDBC 1.2 driver 2
- connecting using JDBC 2.0 driver 4
- identifying 3

data types
- equivalent Java and SQL 23
- equivalent JDBC and SQL 23

DB2Diagnosable class
- retrieving the SQLCA 18

db2profc command
- options 72
- parameters 72

DBINFO clause
- CREATE FUNCTION statement 60
- CREATE PROCEDURE statement 60

diagnosing SQLJ problems 35

diagnosis utilities
- SQLJ 37

driver, JDBC 8
- registering with DriverManager 19

DYNAMICRULES(BIND)
- recommended for SQLJ programs 75

E

environment
- Java stored procedure 53
- Java user-defined function 53

environment variables
- JDBC 92
- SQLJ 92

error handling
- SQLJ 17

executable clause
- SQLJ 47

execution context 30

execution control and status
- SQLJ 30

EXTERNAL
- clause of CREATE FUNCTION statement 57
- clause of CREATE PROCEDURE statement 57

F

FINAL CALL clause
- CREATE FUNCTION statement 60

formatting trace data
  SQLJ 35

G
global transaction
  JDBC and SQLJ 105

H
host expression
  SQLJ 16, 40
hpj command
  invoking the VisualAge for Java binder 79
  options for compiled Java stored procedure 79

I
implements clause
  SQLJ 41
installation
  JDBC and SQLJ 89
interpreted Java stored procedure
  program preparation 76
interpreted Java user-defined function
  program preparation 76
isolation level
  SQLJ 32
iterator
  for positioned DELETE 27
  for positioned UPDATE 27
  obtaining JDBC result sets from 28
iterator conversion clause
  SQLJ 50
iterator declaration clause
  SQLJ 45

J
JAR file
  defining to DB2 56
Java
  equivalent SQL data types 23
Java application
  customizing environment 92
  running 9
java.sql.connection class
  use by SQLJ 23
  valid methods for SQLJ 23
Java stored procedure
  defining to DB2 56
  differences from Java program 64
  differences from other stored procedures 64
  parameters specific to 56
  writing 64
Java thread 103
Java user-defined function
  defining to DB2 56
  differences from Java program 64
  differences from other user-defined functions 64
Java user-defined function *(continued)*
  parameters specific to 56
  writing 64
JDBC
  advantages 1
  API 9
  choosing a driver 1
  configuring 92
  description 1
  driver 8
  environment variables 92
  implementation 1
  installation 89
  interoperability 76
  retrieving the SQLCA 18
  sample program 9
  security 102

L
LANGUAGE
  clause of CREATE FUNCTION statement 56
  clause of CREATE PROCEDURE statement 56

M
methods, invoking 9
multiple context support
  connecting when enabled 103
  connecting when not enabled 103
  description 103
  enabling 104
multiple result sets
  retrieving from a stored procedure 31
multithreading 102

N
named iterator
  example 26
  renaming result table columns for 27
  result set iterator 25
NO SQL
  clause of CREATE FUNCTION statement 59
  clause of CREATE PROCEDURE statement 59
notices, legal 117

O
online checking
  for better optimization 74
  needed during customization 74
  restriction 74
OS/390 UNIX System Services
  authority to access 60

P
PARAMETER STYLE
  clause of CREATE FUNCTION statement 59
  clause of CREATE PROCEDURE statement 59

positioned iterator
  example 25
  result set iterator 24
problem diagnosis
  SQLJ 35
program control extended attribute
  resetting when applying maintenance 90
  setting 90
program preparation
  compiled Java stored procedure 78
  compiled Java stored procedure with no SQLJ 78
  compiled Java stored procedure with SQLJ 78
  interpreted Java stored procedure 76
  interpreted Java stored procedure with no SQLJ 76
  interpreted Java stored procedure with SQLJ 77
  interpreted Java user-defined function 76
  interpreted Java user-defined function with no SQLJ 76
  interpreted Java user-defined function with SQLJ 77
  SQLJ 69
  using the VisualAge for Java binder 79
  VisualAge for Java DLL 84
  VisualAge for Java SQLJ program 81
PROGRAM TYPE clause
  CREATE FUNCTION statement 59
  CREATE PROCEDURE statement 59
properties
  run-time
    CICS 113
    parameters 92

R

read-only mode
  SQLJ 33
result set iterator
  data types in 23
  definition and use in same file 24
  description 23
  positioned iterator 24
  public declaration in separate file 24, 28
  retrieving rows in SQLJ 23, 24, 25
retrieving rows in SQLJ
  named iterator example 26
  positioned iterator example 25
  with named iterators 25
retrieving the SQLCA
  DB2Diagnosable class 18
RRSAF 103
RUN OPTIONS clause
  CREATE FUNCTION statement 60
  CREATE PROCEDURE statement 60
run-time properties file
  CICS 113
  parameters 92
running a program
  SQLJ 35

S

sample program
  JDBC 9
  SQLJ 33
SCRATCHPAD clause
  CREATE FUNCTION statement 60
security
  environment 102
  model 102
SECURITY
  clause of CREATE FUNCTION 60
  clause of CREATE PROCEDURE 60
serialized profile
  customizing 72
SET TRANSACTION clause
  SQLJ 51
SQL
  equivalent Java data types 23
SQL statement
  format in SQLJ 16
  handling errors in SQLJ 17
SQLJ
  assignment clause 50
  basic concepts 16
  binding a plan 75
  case sensitivity 17
  comment 17
  connecting to a data source 19
  connection declaration clause 44
  context clause 47
  creating a DBRM 72
  db2profc command 72
  description 15
  environment variables 92
  error handling 17
  executable clause 16, 47
  execution control and status 30
  format of SQL statement 16
  formatting data 35
  host expression 16, 40
  implements clause 41
  importing Java packages 19
  including code to access 19
  installation 89
  installing the run-time environment 92
  interoperability 76
  isolation level 32
  iterator conversion clause 50
  iterator declaration clause 45
  loading JDBC driver 19
  problem diagnosis 35
  program preparation 69

SQLJ *(continued)*
    read-only mode 33
    result set iterator 23
    retrieving the SQLCA 18
    running a program 35
    running diagnosis utilities 35
    sample program 33
    security 102
    SET TRANSACTION clause 51
    statement clause 48
    translating source code 70
    use of java.sql.connection class 23
    valid SQL statements 16
    with clause 41, 42
SQLJ application
    writing 16
SQLJ execution context 30
sqlj.runtime.ConnectionContext
    methods called in applications 110
sqlj.runtime.ExecutionContext
    methods called in applications 109
sqlj.runtime.ForUpdate
    for positioned UPDATE and DELETE 110
sqlj.runtime.NamedIterator
    methods called in applications 111
sqlj.runtime.PositionedIterator
    methods called in applications 111
sqlj.runtime.ResultSetIterator
    methods called in applications 111
statement clause
    SQLJ 48
stored procedure
    access to OS/390 UNIX System Services 60
    Java 53
    retrieving result sets 31
    returning result set 65
syntax diagrams, how to read vii

T

target data source, identifying 3
thread, Java 103
translating source code
    SQLJ 70

U

user-defined function
    access to OS/390 UNIX System Services 60
    Java 53

V

valid SQL statements
    SQLJ 16
VisualAge for Java binder
    creating compiled Java stored procedure 79

W

WebSphere 104
with clause
    SQLJ 41, 42
with positioned iterators 24
with result set iterators 23
WLM ENVIRONMENT
    clause of CREATE FUNCTION statement 59
    clause of CREATE PROCEDURE statement 59

Readers' Comments — We'd Like to Hear from You

DB2 Universal Database for OS/390 and z/OS
Application Programming
Guide and Reference
FOR JAVA™
Version 7

Publication No. SC26-9932-01

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
SC26-9932-01

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
Department HHX/H3
PO BOX 49023
SAN JOSE CA
U. S. A.
95161-9023

Fold and Tape | **Please do not staple** | Fold and Tape

SC26-9932-01

Cut or Fold
Along Line

Program Number: 5675-DB2

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC26-9932-01