SYBASE®

Sybase® jConnect for JDBC™ Programmer's Reference

jConnect for JDBC

Version 4.2 and 5.2

Document ID: 39001-01-0520-01

Last revised: October 1999

Contents

About This Book

The *Sybase jConnect for JDBC Programmer's Reference* describes the jConnect for JDBC product and explains how to use it to access data stored in relational database management systems.

Audience

This manual is for database-application programmers who are familiar with the Java programming language, JDBC, and Transact-SQL®, the Sybase version of Structured Query Language.

Related Documents

You may find the following documents helpful:

- The *Sybase jConnect for JDBC Installation Guide*
- The *Sybase jConnect for JDBC Release Bulletin*
- The javadoc documentation of jConnect extensions to JDBC. The Java Development Kit (JDK) from JavaSoft contains a javadoc script for extracting comments from source-code files. This script has been used to extract documentation of jConnect packages, classes, and methods from jConnect source files. When you install jConnect using the full installation or javadocs option, the javadoc information is placed in the *javadocs* directory:

  Installation_directory/docs/en/javadocs

Other Sources of Information

Use the Sybase Technical Library CD and the Technical Library Web site to learn more about your product:

- The Technical Library CD contains product manuals and technical documents and is included with your software. The DynaText browser (included on the Technical Library CD) allows you to access technical information about your product in an easy-to-use format.

  Refer to the *Technical Library Installation Guide* in your documentation package for instructions on installing and starting Technical Library.

- The Technical Library Web site includes the Product Manuals site, which is an HTML version of the Technical Library CD that you can access using a standard Web browser. In addition, you'll find links to the Technical Documents Web site (formerly known as Tech Info Library), the Solved Cases page, and Sybase/Powersoft newsgroups.

  To access the Technical Library Web site, go to support.sybase.com, click the Electronic Support Services tab, and select a link under the Technical Library heading.

Sybase Certifications on the Web

Technical documentation at the Sybase Web site is updated frequently.

For the latest information on product certifications and/or the EBF Rollups:

1. Point your Web browser to Technical Documents at the following Web site:

   at techinfo.sybase.com

2. In the Browse section, click on the What's Hot entry.
3. Explore your area of interest: Hot Docs covering various topics, or Hot Links to Technical News, Certification Reports, Partner Certifications, and so on.

If you are a registered Support***Plus*** user:

1. Point your Web browser to Technical Documents at the following Web site:

   at techinfo.sybase.com

2. In the Browse section, click on the What's Hot entry.
3. Click on the EBF Rollups entry.

   You can research EBFs using Technical Documents, and you can download EBFs using Electronic Software Distribution (ESD).

4. Follow the instructions associated with the Support*Plus*^{SM} Online Services entries.

If you are not a registered Support***Plus*** user, and you want to become one:

You can register by following the instructions on the Web.

To use Support*Plus*, you need:

- A Web browser that supports the Secure Sockets Layer (SSL), such as Netscape Navigator 1.2 or later
- An active support license
- A named technical support contact
- Your user ID and password

Whether or not you are a registered Support***Plus*** user:

You may use Sybase's Technical Documents. Certification Reports are among the features documented at this site.

1 Point your Web browser to Technical Documents at the following Web site:

   at techinfo.sybase.com

2 In the Browse section, click on the What's Hot entry.

3 Click on the topic that interests you.

Conventions

This manual uses the following font and syntax conventions:

- Classes, interfaces, methods, and packages are shown in **bold Helvetica** within paragraph text. For example:

  SybConnection class

  SybEventHandler interface

  setBinaryStream() method

  com.sybase.jdbcx package

- Objects, instances, and parameter names are shown in italics. For example:

  "In the following example, *ctx* is a **DirContext** object."

  "eventHdler is an instance of the **SybEventHandler** class that you implement."

  "The *classes* parameter is a string that lists specific classes you want to debug."

- Code fragments are shown in a monospace font. Variables in code fragments (that is, words that stand for values that you fill in) are italicized. For example:

```
Connection con = DriverManager.getConnection("jdbc:
    sybase:Tds:host:port", props);
```

If You Need Help

Each Sybase installation that has purchased a support contract has one or more designated people who are authorized to contact Sybase Technical Support. If you cannot resolve a problem using the manuals or online help, please have the designated person contact Sybase Technical Support or the Sybase subsidiary in your area.

CHAPTER 1 Introduction

This chapter introduces you to jConnect for JDBC, and describes its concepts and components.

This chapter contains the following sections:

What is JDBC?

JDBC (Java Database Connectivity) from the Java Software Division of Sun MicroSystems, Inc. is a specification for an application program interface (API) that allows Java applications to access multiple database management systems using Structured Query Language (SQL). The JDBC driver manager handles multiple drivers that connect to different databases.

A set of interfaces is included in the standard JDBC API so you can open connections to databases, execute SQL commands, and process results. The interfaces are described in Table 1-1.

Table 1-1: JDBC interfaces

Interface	Description
java.sql.Driver	Locates the driver for a database URL
java.sql.Connection	Connection to a specific database
java.sql.Statement	Executes SQL statements
java.sql.PreparedStatement	Handles parameterized SQL statements
java.sql.CallableStatement	Handles database stored procedure calls
java.sql.ResultSet	Gets the results of SQL statements
java.sql.DatabaseMetaData	Used to access a variety of information about a connection’s DBMS and database
java.sql.ResultSetMetaData	Used to access a variety of information describing a ResultSet’s attributes

Each relational database management system requires a driver to implement these interfaces. All JDBC calls are sent to the JDBC driver manager, which passes the call to the specified driver.

There are four types of JDBC drivers:

- *Type 1 JDBC-ODBC bridge* – Translates JDBC calls into ODBC calls and passes them to an ODBC driver. Some ODBC software must be resident on the client machine. Some client database code may also reside on the client machine.
- *Type 2 native-API partly-Java driver* – Converts JDBC calls into database-specific calls. This driver, which communicates directly with the database server, also requires some binary code on the client machine.
- *Type 3 net-protocol all-Java driver* – Communicates to a middle-tier server using a DBMS-independent net protocol. A middle-tier gateway then converts the request to a vendor-specific protocol.

- *Type 4 native-protocol all-Java driver* – Converts JDBC calls to the vendor-specific DBMS protocol, allowing client applications direct communication with the database server.

What is jConnect?

jConnect is Sybase's high-performance JDBC driver. jConnect is both a:

- Net-protocol/all-Java driver within a three-tier environment, and a
- Native-protocol/all-Java driver within a two-tier environment.

The protocol used by jConnect is TDS 5.0 (Tabular Data Stream™, version 5), the native protocol for Adaptive Server® and Open Server™ applications. jConnect implements the JDBC standard to provide optimal connectivity to the complete family of Sybase products, allowing access to over 25 enterprise and legacy systems, including:

- Adaptive Server Enterprise
- Adaptive Server Anywhere
- Adaptive Server IQ (formerly Sybase IQ™)
- Replication Server®
- OmniConnect™

Note Since changing the name of Sybase SQL Server™ to Adaptive Server Enterprise, Sybase may use the names Adaptive Server and Adaptive Server Enterprise to refer collectively to all supported versions of Sybase SQL Server and Adaptive Server Enterprise.

In addition, jConnect for JDBC can access Oracle, AS/400, and other data sources using Sybase DirectConnect™.

In some instances, jConnect's implementation of JDBC deviates from the JDBC 1.x or 2.x specifications. For more information see "Handling Restrictions, Limitations, and Deviations from JDBC Standards" on page 89.

CHAPTER 2 Programming Information

This chapter describes the basic components and programming requirements that comprise jConnect for JDBC. It explains how to invoke the jConnect driver, set connection properties, and connect to a database server. It also contains information about using jConnect features.

Note For information about JDBC programming, go to:

at http://java.sun.com/jdbc.

To access the *JDBC Guide: Getting Started* manual for JDBC 1.0, go to:

at http://java.sun.com/products/jdk/1.1/docs/guide/jdbc.

To access the *JDBC Guide: Getting Started* manual for JDBC 2.0, go to:

at http://java.sun.com/products/jdk/1.2/docs/guide/jdbc/.

The following topics are included in this chapter:

Name	Page
Setting Up jConnect	6
Establishing a Connection	12
Implementing Custom Socket Plug-Ins	26
Handling Internationalization and Localization	31
Working with Databases	36
Implementing Advanced Features	62
Handling Restrictions, Limitations, and Deviations from JDBC Standards	89

Setting Up jConnect

This section describes the tasks you need to perform before you use jConnect.

Setting the jConnect Version

There are several versions of jConnect; use a version setting to determine:

- The default value of the LANGUAGE connection property
- The version-specific features that are available
- The default character set, if no character set is specified through the CHARSET connection property
- The default value of the CHARSET_CONVERTER connection property
- The default value of the CANCEL_ALL connection property, which is used to set the behavior of **Statement.cancel( ),** which by default cancels the object on which it is invoked and any other **Statement** objects that have begun to execute and are waiting for results

Table 2-1 lists the version settings available and their features.

Table 2-1: jConnect version settings and their features

Version Constant	Features	Comments
VERSION_5	• The default value of the LANGUAGE connection property is null. • If the CHARSET connection property does not specify a character set, jConnect uses the database's default character set.The default value for CHARSET_CONVERTER is the PureConverter class. • By default, Statement.cancel( ) cancels only the Statement object it is invoked on. • JDBC 2.0 methods can be used to store and retrieve Java objects as column data.	For jConnect version 5.x, the default is VERSION_5. For additional information, the comments for VERSION_4.

Version Constant	Features	Comments
VERSION_4	• The default value of the LANGUAGE connection property is null. • If the CHARSET connection property does not specify a character set, jConnect uses the database's default character set.The default value for CHARSET_CONVERTER is the PureConverter class. • By default, Statement.cancel( ) cancels only the Statement object it is invoked on. • JDBC 2.0 methods can be used to store and retrieve Java objects as column data.	For jConnect version 4.x and earlier, the default is VERSION_2. Server messages are localized according to the language setting in your local environment. The languages supported are: Chinese, US English, French, German, Japanese, Korean, Portuguese, and Spanish. The default behavior of Statement.cancel( ) is JDBC-compliant. Use CANCEL_ALL to set the behavior of Statement.cancel( ). See "CANCEL_ALL Connection Property" on page 9. For information on Java objects as column data, see "Storing Java Objects as Column Data in a Table" on page 69.
VERSION_3	• The default value of the LANGUAGE connection property is us_english. • If the CHARSET connection property does not specify a character set, jConnect uses the database's default character set. • The default value for CHARSET_CONVERTER is the PureConverter class. • By default, Statement.cancel( ) cancels the object it is invoked on and any other Statement objects that have begun to execute and are waiting for results.	The default is VERSION_2. See the comments for VERSION_2.

Version Constant	Features	Comments
VERSION_2	• The default value of the LANGUAGE connection property is us_english. • If the CHARSET connection property does not specify a character set, the default character set is iso_1. • The default value for CHARSET_CONVERTER is the TruncationConverter class, unless the CHARSET connection property specifies a multibyte or 8-bit character set, in which case the default CHARSET_CONVERTER is the PureConverter class. • By default, Statement.cancel( ) cancels the object it is invoked on and any other Statement objects that have begun to execute and are waiting for results.	The default version setting for jConnect version 2.x is VERSION_2. **Note** VERSION_5 is the default version setting for jConnect version 5.x. The LANGUAGE connection property determines the language in which messages from jConnect and the server appear. For information on the CHARSET and CHARSET_CONVERTER connection classes, see "jConnect Character-Set Converters" on page 31. The VERSION_2 default behavior of Statement.cancel( ) is not JDBC-compliant. Use CANCEL_ALL to set the behavior of Statement.cancel( ). See "CANCEL_ALL Connection Property" on page 9.

The version values are constant values from the **SybDriver** class. When referring to the version constant, use this syntax:

```
com.sybase.jdbcx.SybDriver.VERSION_5
```

Use **SybDriver.setVersion( )** to set the jConnect version. The following code samples show how to load the jConnect driver and set the version.

For jConnect 4.x:

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
   Class.forName ("com.sybase.jdbc.SybDriver").newInstance();
sybDriver.setVersion
   (com.sybase.jdbcx.SybDriver.VERSION_4);
DriverManager.registerDriver(sybDriver);
```

For jConnect 5.x:

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
   Class.forName
   ("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
sybDriver.setVersion
   (com.sybase.jdbcx.SybDriver.VERSION_5);
DriverManager.registerDriver(sybDriver);
```

You can call **setVersion( )** multiple times to change the version setting. New connections inherit the behavior associated with the version setting at the time the connection was made. Changing the version setting during a session does not affect the current connection.

As described in the next section, you can use JCONNECT_VERSION to override the **SybDriver** version setting and specify a different version setting for a specific connection.

JCONNECT_VERSION Connection Property

Use JCONNECT_VERSION to specify the version setting for a specific session.You can set JCONNECT_VERSION to an integer value of “2,” “3,” “4,” or “5,” depending on the characteristics you want (see Table 2-1).

CANCEL_ALL Connection Property

CANCEL_ALL is a Boolean-valued connection property for specifying the behavior of the **Statement.cancel( )** method.

Note In jConnect version 4.0 and earlier, the default for CANCEL_ALL is “true.” In jConnect version 4.1 and later, to comply with the JDBC specification, if you set the connection property JCONNECT_VERSION to “4” or above, the default setting for CANCEL_ALL is “false.”

The settings for CANCEL_ALL have the following effect on **Statement.cancel( )**:

- If CANCEL_ALL is “false,” invoking **Statement.cancel( )** cancels only the **Statement** object it is invoked on. Thus, if **stmtA** is a **Statement** object, **stmtA.cancel( )** cancels the execution of the SQL statement contained in **stmtA** in the database, but no other statements are affected. **stmtA** is canceled whether it is in cache waiting to execute or has started to execute and is waiting for results.
- If CANCEL_ALL is “true,” invoking **Statement.cancel( )** cancels not only the object it is invoked on, but also any other **Statement** objects on the same connection that have executed and are waiting for results.

The following example sets CANCEL_ALL to “false.” In the example, *props* is a **Properties** object for specifying connection properties.

```
...
 props.put("CANCEL_ALL", "false");
```

Note To cancel the execution of all **Statement** objects on a connection, regardless of whether or not they have begun execution on the server, use the extension method **SybConnection.cancel( ).**

Invoking the jConnect Driver

To register and invoke the Sybase jConnect driver, use either of two suggested methods:

Method 1

For jConnect 4.x:

```
Class.forName("com.sybase.jdbc.SybDriver").newInstance();
```

For jConnect 5.x:

```
Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
```

Method 2

Add the jConnect driver to the **jdbc.drivers** system property. At initialization, the **DriverManager** class attempts to load the drivers listed in **jdbc.drivers**. This is less efficient than the previous approach. You can list multiple drivers in this property, separated with a colon (:). The following code samples show how to add a driver to **jdbc.drivers** within a program:

For jConnect 4.x:

```
Properties sysProps = System.getProperties();
String drivers = "com.sybase.jdbc.SybDriver";
String oldDrivers =
sysProps.getProperty("jdbc.drivers");
if (oldDrivers != null)
   drivers += ":" + oldDrivers;
sysProps.put("jdbc.drivers", drivers.toString());
```

For jConnect 5.x:

```
Properties sysProps = System.getProperties();
String drivers = "com.sybase.jdbc2.jdbc.SybDriver";
String oldDrivers =
sysProps.getProperty("jdbc.drivers");
if (oldDrivers != null)
   drivers += ":" + oldDrivers;
```

```
    sysProps.put("jdbc.drivers", drivers.toString());
```

Note **System.getProperties( )** is not allowed for Java applets. Use the **Class.forName( )** method, instead.

Establishing a Connection

This section describes how to establish a connection to an Adaptive Server Enterprise or Adaptive Server Anywhere database using jConnect.

Setting Connection Properties

Table 2-2 lists the connection properties for jConnect and indicates their default values. *You must set the connection properties before you make a connection.*

There are two ways to set the driver connection properties:

- Using the **DriverManager.getConnection( )** method in your application
- When you define the URL

Note Driver connection properties set in the URL do not override any corresponding connection properties set in the application using the **DriverManager.getConnection( )** method.

To obtain a current list of properties for any driver, use the **Driver.getDriverPropertyInfo(String url, Properties props),** which returns an array of **DriverPropertyInfo** objects. The array lists:

- The driver properties
- The current settings on which the driver properties are based
- The URL and **props** passed in

Driver connection property names are not case-sensitive (jConnect uses the **String.equalsIgnoreCase(String)** method to compare property names).

Table 2-2: Connection properties

Property	Description	Default Value
APPLICATIONNAME	A user-defined property. The server side can be programmed to interpret the value given to this property.	Null
CANCEL_ALL	Determines the behavior of the Statement.cancel( ) method. See “CANCEL_ALL Connection Property” on page 9.	Depends on version setting. (See “Setting the jConnect Version” on page 6.

Property	Description	Default Value
CHARSET	Specifies the character set for strings passed through TDS. If you specify a CHARSET, it must match a CHARSET listed in *syscharsets*. If null, jConnect uses the server's default CHARSET.	Null
CHARSET_CONVERTER_CLASS	Use this property to specify the character-set converter class you want jConnect to use. jConnect uses the version setting from SybDriver.setVersion( ) to determine the default character-set converter class to use. See "Selecting a Character-Set Converter" on page 31 for details.	Version dependent.
CONNECTION_FAILOVER	For use with the Java Naming and Directory Interface (JNDI). See "CONNECTION_FAILOVER Connection Property" on page 22.	true
DYNAMIC_PREPARE	Determines whether dynamic SQL prepared statements are precompiled in the database. See "DYNAMIC_PREPARE Connection Property" on page 115.	false
EXPIRESTRING	A read-only property that contains the license expiration date. Expiration is "never" except for evaluation copies of jConnect.	Never
HOSTNAME	The name of the current host.	None
HOSTPROC	Identifies the application's process on the host machine.	None
IGNORE_DONE_IN_PROC	When set to "true," intermediate update results (as in stored procedures) are not returned, only the final resultset.	false
JCONNECT_VERSION	Use this property to set version-specific characteristics. See "JCONNECT_VERSION Connection Property" on page 9.	5
LANGUAGE	Set this property for error messages returned from the server and for jConnect messages. It must match a language in *syslanguages*.	Version dependent. See "Setting the jConnect Version" on page 6.
LANGUAGE_CURSOR	Set this property to true if you want jConnect to use "language cursors" instead of "protocol cursors." See "Cursor Performance" on page 117.	false

Property	Description	Default Value
LITERAL_PARAMS	This property is for use only with Adaptive Server Anywhere, which requires you to send prepared statement parameters as literals. For all other Sybase databases, this property can be set to "false." When set to "true," any parameters set by the setXXX methods in the PreparedStatement interface are inserted literally into the SQL statement when it is executed. If set to "false," parameter markers are left in the SQL statement and the parameter values are sent to the server separately.	false
PACKETSIZE	Network packet size.	512
PASSWORD	Login password. Set automatically if using the getConnection(String, String, String) method, or explicitly if using getConnection(String, Props).	None
PROTOCOL_CAPTURE	The PROTOCOL_CAPTURE connection property is used to specify a file for capturing TDS communication between an application and an Adaptive Server.	Null
PROXY	Gateway address. For the HTTP protocol, the URL is: *http://host:port*. To use the HTTPS protocol that supports encryption, the URL is https://*host*:*port*/*servlet_alias*.	None
REMOTEPWD	Remote server passwords for access via server-to-server remote procedure calls. See "Performing Server-to-Server Remote Procedure Calls" on page 41.	None
REPEAT_READ	Determines whether the driver keeps copies of columns and output parameters so that columns can be read out of order or repeatedly. See "REPEAT_READ Connection Property" on page 109.	true

Property	Description	Default Value
REQUEST_HA_SESSION	This property indicates whether the connecting client wants to begin a HA Failover session with a version 12 or later Adaptive Server configured for HA Failover. Setting this property to "true" causes jConnect to attempt a HA Failover login. If you do not set this connection property, a HA Failover session will not start, even if the server is configured for HA Failover You cannot reset the property once a connection has been made. If you want more flexibility for requesting HA Failover sessions, code the client application to set REQUEST_HA_SESSION at runtime.	false
SELECT_OPENS_CURSOR	If set to "true," calls to Statement.executeQuery( ) will automatically generate a cursor when the query contains a "FOR UPDATE" clause. If you have previously called Statement.setFetchSize( ) or Statement.setCursorName( ) on the same statement, a setting of "true" for SELECT_OPENS_CURSOR has no effect. **Note** You may experience some performance degradation when SELECT_OPENS_CURSOR is set to "true." See "Using Cursors with Result Sets" on page 44 for more information on using cursors with jConnect.	false
SERIALIZE_REQUESTS	If set to "true," jConnect waits for responses from the server before sending additional requests.	false
SERVICENAME	The name of a back-end database server that a DirectConnect gateway serves. Also used to indicate the database to which Adaptive Server Anywhere wants to connect.	None

Property	Description	Default Value
SESSION_ID	When this property is set, jConnect assumes that an application is trying to resume communication on an existing TDS session held open by the TDS-tunnelling gateway. jConnect skips the login negotiations and forwards all requests from the application to the specified session ID.	Null
SESSION_TIMEOUT	Use this property to specify the amount of time (in seconds) that an http-tunnelled session (created using the jConnect TDS-tunnelling servlet) will remain alive while idle. After the specified time, the connection will be automatically closed. For more information about the TDS-tunnelling servlet, see page 136.	Null
SQLINITSTRING	Use this property to define a set of commands to be passed to the back-end database server. These must be SQL commands that can be executed using the Statement.executeUpdate( ) method.	Null
SYBSOCKET_FACTORY	Use this property to enable jConnect to use your custom socket implementation. Set SYBSOCKET_FACTORY either to: • The name of a class that implements com.sybase.jdbcx.SybSocketFactory; or • “DEFAULT,” which instantiates a new java.net.Socket( ) See “Implementing Custom Socket Plug-Ins” on page 26.	Null
STREAM_CACHE_SIZE	Maximum size used to cache statement response streams.	Null (unlimited cache size)

Property	Description	Default Value
USE_METADATA	When set to “true,” a DatabaseMetaData object will be created and initialized when you establish a connection. The DatabaseMetaData object is necessary to connect to a specified database. jConnect uses DatabaseMetaData for some features, including Distributed Transaction Management Support (JTA/JTS) and Dynamic Class Loading (DCL). If you receive error 010SJ, which indicates that your application requires metadata, install the stored procedures for returning metadata that come with jConnect (see “Installing Stored Procedures” in Chapter 3 of the *jConnect for JDBC Installation Guide*).	true
USER	Login ID. Set automatically if using the getConnection(String, String, String) method, or explicitly if using getConnection(String, Props).	None
VERSIONSTRING	Read-only version information for the JDBC driver.	jConnect driver version

The following code is an example of setting connection properties. The sample programs provided with jConnect also contain examples of setting these properties.

```
Properties props = new Properties();
 props.put("user", "userid");
 props.put("password", "user_password");
/*
 * If the program is an applet that wants to access
 * a server that is not on the same host as the
 * web server, then it uses a proxy gateway.
 */
 props.put("proxy", "localhost:port");
/*
 * Make sure you set connection properties before
 * attempting to make a connection. You can also
 * set the properties in the URL.
 */
Connection con = DriverManager.getConnection
("jdbc:sybase:Tds:host:port", props);
```

Connecting to Adaptive Server Enterprise

In your Java application, define a URL using the jConnect driver to connect to an Adaptive Server. The basic format of the URL is:

```
jdbc:sybase:Tds:host:port
```

where:

jdbc:sybase – Identifies the driver.

Tds – The Sybase communication protocol for Adaptive Server.

host:*port* – The Adaptive Server host name and listening port. See *$SYBASE/interfaces* (UNIX) or *%SYBASE%\ini\sql.ini* (Windows) for the entry that your database or Open Server application uses. Obtain the *host:port* from the “query” entry.

You can connect to a specific database using this format:

```
jdbc:sybase:Tds:host:port/database
```

Note To connect to a specific database using Adaptive Server Anywhere 6.x or DirectConnect, use the Connection properties connection property to specify the database name instead of “/database.”

Example

The following code creates a connection to an Adaptive Server on host “myserver” listening on port 3697:

```
SysProps.put("user","userid");
 SysProps.put("password","user_password");
 String url = "jdbc:sybase:Tds:myserver:3697";
 Connection_con =
   DriverManager.getConnection(url,SysProps);
```

URL Connection Property Parameters

You can specify the values for the jConnect driver connection properties when you define a URL.

Note Driver connection properties set in the URL do not override any corresponding connection properties set in the application using the **DriverManager.getConnection( )** method.

To set a connection property in the URL, append the property name and its value to the URL definition. Use this syntax:

```
jdbc:sybase:Tds:host:port/database?
 property_name=value
```

To set multiple connection properties, append each additional connection property and value, preceded by "&." For example:

```
jdbc:sybase:Tds:myserver:1234/mydatabase?
 LITERAL_PARAMS=true&PACKETSIZE=512&HOSTNAME=myhost
```

If the value for one of the connection properties contains "&," precede the "&" in the connection property value with a backslash (\). For example, if your host name is "a&bhost," use this syntax:

```
jdbc:sybase:Tds:myserver:1234/mydatabase?
 LITERAL_PARAMS=true&PACKETSIZE=512&HOSTNAME=
 a\&bhost
```

Do not use quotes for connection property values, even if they are strings. For example, use:

```
HOSTNAME=myhost
```

not:

```
HOSTNAME="myhost"
```

Connecting to Adaptive Server Anywhere

To use jConnect with Adaptive Server Anywhere, you should upgrade to Adaptive Server Anywhere version 6.x.

Connecting to Adaptive Server Anywhere 5.x.x

If you have to connect to Adaptive Server Anywhere version 5.x.x via jConnect, you must run the Adaptive Server Anywhere Open Server Gateway dbos50, which is distributed with Adaptive Server Anywhere.

Note The free download version of Adaptive Server Anywhere, available from the Powersoft Web site, does not include this Open Server Gateway. Call Powersoft at (800) 265-4555 to receive a CD that includes the Open Server Gateway and the required Open Server DLLs. You will be charged only for shipping and handling.

---

1 Install Open Server Gateway 5.5.x3 or later and the Open Server DLLs. Use Open Server DLLs, version 11.1.

2. Add an entry for the gateway to your *%SYBASE%\ini\sql.ini* file (using, for example, **sqledit**).
3. Start the gateway by entering:

   ```
   start dbos50 gateway-demo
   ```

   where *gateway-demo* is the gateway name defined in step 2.
4. When the Open Server Gateway is running, you can define a connection as follows:

   ```
   jdbc:sybase:Tds:host:port
   ```

   host is the host name where the Adaptive Server Anywhere and Open Server gateway is running, and *port* is the port number defined in *sql.ini*.

Note To support multiple Adaptive Server Anywhere databases, use **sqledit** to add an entry with a different port for each database, then run the Open Server Gateway for each database.

Connecting to a Server Using JNDI

In jConnect 4.0 and later, you can use the Java Naming and Directory Interface (JNDI) to provide connection information, which offers:

- A centralized location where you can specify host names and ports for connecting to a server. You do not need to hard code a specific host and port number in an application.
- A centralized location where you can specify connection properties and a default database for all applications to use.
- The jConnect CONNECTION_FAILOVER property for handling unsuccessful connection attempts. When CONNECTION_FAILOVER is set to “true,” jConnect attempts to connect to a sequence of host/port server addresses in the JNDI name space until one succeeds.

To use jConnect with JNDI, you need to make sure that certain information is available in any directory service that JNDI accesses and that required information is set in the **javax.naming.Context** class. This section covers the following topics:

- Connection URL for Using JNDI
- Required Directory Service Information

- CONNECTION_FAILOVER Connection Property
- Providing JNDI Context Information

Connection URL for Using JNDI

To specify that jConnect use JNDI to obtain connection information, place “jndi” as the URL’s subprotocol after “sybase”:

```
jdbc:sybase:jndi:protocol-information-for-use-with-JNDI
```

Anything that follows “jndi” in the URL is handled through JNDI. For example, to use JNDI with the Lightweight Directory Access Protocol (LDAP), you might enter:

```
jdbc:sybase:jndi:ldap://LDAP_hostname:port_number/servername=
   Sybase11,o=MyCompany,c=US
```

This URL tells JNDI to obtain information from an LDAP server, gives the host name and port number of the LDAP server to use, and provides the name of a database server in an LDAP-specific form.

Required Directory Service Information

When you use JNDI with jConnect, JNDI needs to return the following information for the target database server:

- A host name and port number to connect to
- The name of the database to use
- Any connection properties that individual applications are not allowed to set on their own

This information needs to be stored according to a fixed format in any directory service used for providing connection information. The required format consists of a numerical object identifier (OID), which identifies the type of information being provided (for example, the destination database), followed by the formatted information. Table 2-3 shows the required formatting.

Table 2-3: Directory service information required for JNDI

Type of Information	Object Identifier (OID)	Format	Comments
Host and port	1.3.6.1.4.1.897.4.2.5	TCP#1#*hostname* *portnumber*	You can specify multiple hosts and ports as separate entries, which lets you use CONNECTION_FAILOVER.

Type of Information	Object Identifier (OID)	Format	Comments
Connection property	1.3.6.1.4.1.897.4.2.10	*Prop1=value&Prop2= value&Prop3=value&...*	You can specify multiple connection properties by using a separate entry for each property or by putting multiple properties, separated by ampersands, in a single entry.
Database	1.3.6.1.4.1.897.4.2.11	*databasename*	The name of the database to which you want to connect. This property works like the "/database" in the JDBC URL.
Connection protocol	1.3.6.1.4.1.897.4.2.9	Tds	Optional, but if you use a connection protocol, it must always be "Tds."

The following example shows connection information entered for the database server SYBASE11 under an LDAP directory service:

```
dn: servername=SYBASE11,o=MyCompany,c=US
 servername: SYBASE11
 1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1266
 1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1337
 1.3.6.1.4.1.897.4.2.5:TCP#1#standby1 4444
 1.3.6.1.4.1.897.4.2.10:REPEAT_READ=false&PACKETSIZE=1024
 1.3.6.1.4.1.897.4.2.10:CONNECTION_FAILOVER=true
 1.3.6.1.4.1.897.4.2.11:pubs2
 1.3.6.1.4.1.897.4.2.9:Tds
```

In this example, SYBASE11 can be accessed through either port 1266 or port 1337 on host "giotto" and it can be accessed through port 4444 on host "standby1." Two connection properties, REPEAT_READ and PACKETSIZE, are set within one entry. The CONNECTION_FAILOVER connection property is set as a separate entry. Applications connecting to SYBASE11 are initially connected with the *pubs2* database. You do not need to specify a connection protocol, but if you do, you must enter the attribute as "`Tds`", not "`TDS`".

CONNECTION_FAILOVER Connection Property

CONNECTION_FAILOVER is a Boolean-valued connection property you can use when jConnect uses JNDI to get connection information.

If CONNECTION_FAILOVER is set to “true,” jConnect makes multiple attempts to connect to a server. If one attempt to connect to a host and port number associated with a server fails, jConnect uses JNDI to get the next host and port number associated with the server and attempts to connect through them. Connection attempts proceed sequentially through all the hosts and ports associated with a server.

For example, suppose CONNECTION_FAILOVER is set to “true,” and a database server is associated with the following hosts and port numbers, as in the earlier LDAP example:

```
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1266
1.3.6.1.4.1.897.4.2.5:TCP#1#giotto 1337
1.3.6.1.4.1.897.4.2.5:TCP#1#standby1 4444
```

To get a connection to the server, jConnect tries to connect to the host “giotto” at port 1266. If this fails, jConnect tries port 1337 on “giotto.” If this fails, jConnect tries to connect to host “standby1” through port 4444.

The default for CONNECTION_FAILOVER is “true.”

If CONNECTION_FAILOVER is set to “false,” jConnect attempts to connect to an initial host and port number. If the attempt fails, jConnect throws a SQL exception and does not try again.

Providing JNDI Context Information

To use jConnect with JNDI, a developer should be familiar with the JNDI specification from Sun Microsystems, available from the Web:

http://java.sun.com/products/jndi

In particular, the developer needs to make sure that required initialization properties are set in **javax.naming.directory.DirContext** when JNDI and jConnect are used together. These properties can be set either at the system level or at runtime.

Two key properties are:

- Context.INITIAL_CONTEXT_FACTORY

  This property takes the fully qualified class name of the initial context factory for JNDI to use. This determines the JNDI driver that is used with the URL specified in the Context.PROVIDER_URL property.

- Context.PROVIDER_URL

This property takes the URL of the directory service that the driver (for example, the LDAP driver) is to access. The URL should be a string, such as "ldap://ldaphost:427".

The following example shows how to set context properties at runtime and how to get a connection using JNDI and LDAP. In the example, the INITIAL_CONTEXT_FACTORY context property is set to invoke Sun Microsystem's implementation of an LDAP service provider. The PROVIDER_URL context property is set to the URL of an LDAP directory service located on the host "ldap_server1" at port 983.

```
Properties props = new Properties();

 /* We want to use LDAP, so INITIAL_CONTEXT_FACTORY is set to the
 * class name of an LDAP context factory. In this case, the
 * context factory is provided by Sun's implementation of a
 * driver for LDAP directory service.
 */
 props.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.ldap.LdapCtxFactory");

 /* Now, we set PROVIDER_URL to the URL of the LDAP server that
 * is to provide directory information for the connection.
 */
 props.put(Context.PROVIDER_URL, "ldap://ldap_server1:983");

 /* Set up additional context properties, as needed. */
 props.put("user", "xyz");
 props.put("password", "123");

 /* get the connection */
 Connection con = DriverManager.getConnection
   ("jdbc:sybase:jndi:ldap://ldap_server1:983" +
   "/servername=Sybase11,o=MyCompany,c=US",props);
```

Note that the connection string passed to **getConnection( )** contains LDAP-specific information, which the developer must provide.

When JNDI properties are set at runtime, as in the preceding example, jConnect passes them to JNDI to be used in initializing a server, as in the following jConnect code:

```
javax.naming.directory.DirContext ctx =
   new javax.naming.directory.InitialDirContext(props);
```

jConnect then obtains the connection information it needs from JNDI by invoking **DirContext.getAtributes( )**, as in the following example, where *ctx* is a **DirContext** object:

```
javax.naming.directory.Attributes attrs =
   ctx.getAttributes(ldap://ldap_server1:983/servername=
      Sybase11, SYBASE_SERVER_ATTRIBUTES);
```

In the example, SYBASE_SERVER_ATTRIBUTES is an array of strings defined within jConnect. The array values are the OIDs for the required directory information listed in Table 2-3.

Implementing Custom Socket Plug-Ins

This section discusses how to plug a custom socket implementation into an application to customize the communication between a client and server. **javax.net.ssl.SSLSocket** is an example of a socket that you could customize to enable encryption.

com.sybase.jdbcx.SybSocketFactory is a Sybase extension interface that contains the **createSocket(String, int, Properties)** method that returns a **java.net.Socket**. In order for a jConnect version 4.1 or later driver to load a custom socket, an application must:

- Implement this interface
- Define the **createSocket(..)** method

jConnect uses the new socket for its subsequent input/output operations. Classes that implement **SybSocketFactory** create sockets and provide a general framework for the addition of public socket-level functionality.

```
/**
 * Returns a socket connected to a ServerSocket on the named host,
 * at the given port.
 * @param host  the server host
 * @param port  the server port
 * @param props  Properties passed in through the connection
 * @returns Socket
 * @exception IOException, UnknownHostException
 */
 public java.net.Socket createSocket(String host, int port, Properties props)
throws IOException, UnknownHostException;
```

Passing in properties allows instances of **SybSocketFactory** to use connection properties to implement an intelligent socket.

When you implement **SybSocketFactory** to produce a socket, the same application code can use different kinds of sockets by passing the different kinds of factories or pseudo-factories that create sockets to the application. You can customize factories with parameters used in socket construction. For example, you could customize factories to return sockets with different networking time-outs or security parameters already configured. The sockets returned to the application can be subclasses of **java.net.Socket** to directly expose new APIs for features such as compression, security, record marking, statistics collection, or firewall tunnelling (**javax.net.SocketFactory**).

Note **SybSocketFactory** is intended to be an overly simplified **javax.net.SocketFactory**, enabling applications to bridge from **java.net.*** to **javax.net.*** if desired.

To use a custom socket with jConnect:

1. Provide a Java class that implements **com.sybase.jdbcx.SybSocketFactory**. See “Creating and Configuring a Custom Socket” on page 27.
2. Set the SYBSOCKET_FACTORY connection property so that jConnect can use your implementation to obtain a socket.

SYBSOCKET_FACTORY Connection Property

To use a custom socket with jConnect, set the SYBSOCKET_FACTORY connection property to a string that is either:

- The name of a class that implements **com.sybase.jdbcx.SybSocketFactory**

or

- DEFAULT, which instantiates a new **java.net.Socket( )**

See “Setting Connection Properties” on page 12 for instructions on how to set SYBSOCKET_FACTORY.

Creating and Configuring a Custom Socket

Once jConnect obtains a custom socket, it uses the socket to connect to a server. Any configuration of the socket must be completed before jConnect obtains it.

This section explains how to plug in an SSL socket implementation, such as **javax.net.ssl.SSLSocket**, with jConnect.

Note Currently, no Sybase servers support SSL.

The following example shows how an implementation of SSL can create an instance of **SSLSocket**, configure it, and then return it. In the example, the **MySSLSocketFactory** class implements **SybSocketFactory** and extends **javax.net.ssl.SSLSocketFactory** to implement SSL. It contains two **createSocket** methods—one for **SSLSocketFactory** and one for **SybSocketFactory**—that:

- Create an SSL socket
- Invoke **SSLSocket.setEnableCipherSuites( )** to specify the cipher suites available for encryption
- Return the socket to be used by jConnect

Example

```
public class MySSLSocketFactory extends SSLSocketFactory
   implements SybSocketFactory
 {
 /**
 * Create a socket, set the cipher suites it can use, return
 * the socket.
 * Demonstrates how cither suites could be hard-coded into the
 * implementation.
 *
 * See javax.net.SSLSocketFactory#createSocket
 */
public Socket createSocket(String host, int port)
   throws IOException, UnknownHostException
 {
   // Prepare an array containing the cipher suites that are to
   // be enabled.
   String enableThese[] =
   {
         "SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA",
         "SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5",
         "SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA"
   }
   ;
   Socket s =
          SSLSocketFactory.getDefault().createSocket(host, port);
   ((SSLSocket)s).setEnabledCipherSuites(enableThese);
   return s;
 }
/**
 * Return an SSLSocket.
 * Demonstrates how to set cipher suites based on connection
 * properties like:
 * Properties _props = new Properties();
 * Set other url, password, etc. properties.
 * _props.put(("CIPHER_SUITES_1",
 *     "SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA");
 * _props.put("CIPHER_SUITES_2",
 *     "SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5");
 * _props.put("CIPHER_SUITES_3",
 *     "SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA");
 * _conn = _driver.getConnection(url, _props);
 *
 * See com.sybase.jdbcx.SybSocketFactory#createSocket
 */
public Socket createSocket(String host, int port,
   Properties props)
   throws IOException, UnknownHostException
 {
   // check to see if cipher suites are set in the connection
   // properites
   Vector cipherSuites = new Vector();
   String cipherSuiteVal = null;
   int cipherIndex = 1;
   do
   {
         if((cipherSuiteVal = props.getProperty("CIPHER_SUITES_"
               + cipherIndex++)) == null)
         {
               if(cipherIndex <= 2)
               {
                     // No cipher suites available
                     // return what the object considers its default
                     // SSLSocket, with cipher suites enabled.
                     return createSocket(host, port);
               }
               else
               {
                     // we have at least one cipher suite to enable
                     // per request on the connection
                     break;
```

```
                 }
                 else
                 }
                       // add to the cipher suit Vector, so that
                       // we may enable them together
                       cipherSuites.addElement(cipherSuiteVal);
                 }
           }
           while(true);
           // lets you create a String[] out of the created vector
           String enableThese[] = new String[cipherSuites.size()];
           cipherSuites.copyInto(enableThese);
           // enable the cipher suites
           Socket s =
                 SSLSocketFactory.getDefault().createSocket
                    (host, port);
           ((SSLSocket)s).setEnabledCipherSuites(enableThese);
           // return the SSLSocket
           return s;
     }
     // other methods
  }
```

Since jConnect requires no information about the kind of socket it is, you must complete any configuration before you return a socket.

For additional information, see:

- *Encrypt.java* – Located in the *sample* (jConnect 4.x) and *sample2* (jConnect 5.x) subdirectories of your jConnect directory, this sample shows you how to use the **SybSocketFactory** interface with jConnect applications.
- *MySSLSocketFactory.java* – Also located in the *sample* (jConnect 4.x) and *sample2* (jConnect 5.x) subdirectories of your jConnect directory, this is a sample implementation of the **SybSocketFactory** interface that you can plug-in to your application and use.

Handling Internationalization and Localization

This section discusses internationalization and localization issues relevant to jConnect.

jConnect Character-Set Converters

jConnect uses special classes for all character-set conversions. By selecting a character-set converter class, you specify how jConnect should handle single-byte and multibyte character-set conversions, and the performance impact the conversions will have on your applications.

There are two character-set conversion classes. The conversion class that jConnect uses is based on the version setting (for example, VERSION_4), and the CHARSET and CHARSET_CONVERTER_CLASS connection properties.

- The **TruncationConverter** class works only with single-byte character sets that use ASCII characters such as iso_1 and cp850. It does not work with multibyte character sets or single-byte character sets that use non-ASCII characters.

  Using the **TruncationConverter** class, jConnect 5.x handles character sets in the same manner as jConnect version 2.2. The **TruncationConverter** class is the default converter when the version setting is VERSION_2.

- The **PureConverter** class is a pure Java, multibyte character-set converter. jConnect uses this class if the version setting is VERSION_4 or higher. jConnect also uses this converter with VERSION_2 if it detects a character set specified in the CHARSET connection property that is not compatible with the **TruncationConverter** class.

  Although it enables multibyte character set conversions, the **PureConverter** class may negatively impact jConnect driver performance. If driver performance is a concern, see “Improving Character Set Conversion Performance” on page 33.

Selecting a Character-Set Converter

jConnect uses the version setting from **SybDriver.setVersion( )** to determine the default character-set converter class to use. For VERSION_2, the default is **TruncationConverter**. For VERSION_4 and later, the default is **PureConverter**.

You can also set the CHARSET_CONVERTER_CLASS connection property to specify which character-set converter you want jConnect to use. This is useful if you want to use a character-set converter other than the default for your jConnect version.

For example, if you set jConnect to VERSION_4 or higher, but want to use the **TruncationConverter** class rather than the multibyte **PureConverter** class, you can set CHARSET_CONVERTER_CLASS:

For jConnect 4.1:

```
...
 props.put("CHARSET_CONVERTER_CLASS",
 "com.sybase.utils.TruncationConverter")
```

For jConnect 5.x:

```
...
 props.put("CHARSET_CONVERTER_CLASS",
 "com.sybase.jdbc2.utils.TruncationConverter")
```

Setting the CHARSET Connection Property

You can specify the character set to use in your application by setting the CHARSET driver property. If you do not set the CHARSET property:

- For VERSION_2, jConnect uses iso_1 as the default character set.
- For VERSION_3, VERSION_4, and VERSION_5, jConnect uses the database's default character set, and adjusts automatically to perform any necessary conversions on the client side.

You can also use the **-J** *charset* command line option for the **IsqlApp** application to specify a character set.

To determine which character sets are installed on your Adaptive Server, issue the following SQL query on your server:

```
select name from syscharsets
 go
```

For the **PureConverter** class, if the designated CHARSET does not work with the client's Java Virtual Machine (VM), the connection fails with a **SQLException**, indicating that you must set CHARSET to a character set that is supported by both Adaptive Server and the client.

When the **TruncationConverter** class is used, character truncation is applied regardless of whether the designated CHARSET is 7-bit ASCII or not.

Improving Character Set Conversion Performance

If you use multibyte character sets and need to improve driver performance, you can use the **SunIoConverter** class provided with the jConnect samples. See "Character-Set Conversion" on page 109 for details.

Supported Character Sets

Table 2-4 lists the Sybase character sets that are supported by this release of jConnect. The table also lists the corresponding JDK byte converter for each supported character set.

Although jConnect supports UCS-2, currently no Sybase databases or open servers support UCS-2.

The Sybase *sjis* character set does not include the IBM or Microsoft extensions to JIS, whereas the JDK SJIS byte converter includes these extensions. As a result, conversions from Java strings to a Sybase database using *sjis* may result in character values that are not supported by the Sybase database. However, conversions from *sjis* on a Sybase database to Java strings should not have this problem.

Table 2-4 lists the character sets currently supported by Sybase.

Table 2-4: Supported Sybase character sets

SybCharset Name	JDK Byte Converter
ascii_7	8859_1
big5	Big5
cp037	Cp037
cp437	Cp437
cp500	Cp500
cp850	Cp850
cp852	Cp852
cp855	Cp855
cp857	Cp857
cp860	Cp860
cp863	Cp863
cp864	Cp864
cp866	Cp866
cp869	Cp869
cp874	Cp874
cp932	Cp932

SybCharset Name	JDK Byte Converter
cp936	Cp936
cp950	Cp950
cp1250	Cp1250
cp1251	Cp1251
cp1252	Cp1252
cp1253	Cp1253
cp1254	Cp1254
cp1255	Cp1255
cp1256	Cp1256
cp1257	Cp1257
cp1258	Cp1258
deckanji	EUCJIS
eucgb	GB2312
eucjis	EUCJIS
eucksc	Cp949
ibm420	Cp420
ibm918	Cp918
iso_1	8859_1
iso88592	8859-2
is088595	8859_5
iso88596	8859_6
iso88597	8859_7
iso88598	8859_8
iso88599	8859_9
iso885915	8859_15
koi8	KOI8_R
mac	Macroman
mac_cyr	MacCyrillic
mac_ee	MacCentralEurope
macgreek	MacGreek
macturk	MacTurkish
sjis (see note)	SJIS
tis620	MS874
utf8	UTF8

European Currency Symbol Support

jConnect version 4.1 and later support the use of the new European currency symbol, or "*euro*" and its conversion to and from UCS-2 Unicode.

The *euro* has been added to the following Sybase character sets: cp1250, cp1251, cp1252, cp1253, cp1254, cp1255, cp1256, cp1257, cp1258, cp874, iso885915, and utf8.

Character sets cp1257, cp1258, and iso885915 are new.

To use the *euro* symbol:

- Use the **PureConverter** class, a pure Java, multibyte character-set converter. See "jConnect Character-Set Converters" on page 31 for more information.
- Verify that the new character sets are installed on the server.

  The *euro* symbol is currently supported only on Adaptive Server Enterprise version 11.9.2 and later; Adaptive Server Anywhere does not support the *euro* symbol.
- Select the appropriate character set on the client. See "Setting the CHARSET Connection Property" on page 32 for more information.
- Upgrade to JDK 1.1.7 or the Java™ 2 Platform.

Unsupported Character Sets

The following Sybase character sets are not supported in jConnect 5.x because no JDK byte converters are analogous to the Sybase character sets:

- cp1047
- euccns
- greek8
- roman8
- turkish8

You can use these character sets with the **TruncationConverter** class as long as the application uses only the 7-bit ASCII subsets of these characters.

Working with Databases

This section discusses database issues relevant to jConnect and includes these topics:

- Implementing High Availability Failover Support
- Performing Server-to-Server Remote Procedure Calls
- Accessing Database Metadata
- Using Cursors with Result Sets
- Support for Batch Updates
- Updating the Database from the Result Set of a Stored Procedure
- Working with Datatypes

Implementing High Availability Failover Support

jConnect versions 4.2 and 5.2 support the Sybase Failover feature available in Adaptive Server Enterprise version 12.0.

Note Sybase Failover in a high availability system is a different feature than "connection failover." Sybase strongly recommends that you read this section *very carefully* if you want to use both.

Overview

Sybase Failover allows you to configure two version 12.0 Adaptive Servers as companions. If the primary companion fails, that server's devices, databases, and connections can be taken over by the secondary companion.

You can configure a high availability system either asymmetrically or symmetrically.

An *asymmetric* configuration includes two Adaptive Servers, each physically located on a different machine, that are connected so that if one of the servers is brought down, the other assumes its workload. The secondary Adaptive Server acts as a "hot standby" and does not perform any work until failover occurs.

A *symmetric* configuration also includes two Adaptive Servers running on separate machines. However, if failover occurs, either Adaptive Server can act as a primary or secondary companion for the other Adaptive Server. In this configuration, each Adaptive Server is fully functional with its own system devices, system databases, user databases, and user logins.

In either setup, the two machines are configured for dual access, which makes the disks visible and accessible to both machines.

You can enable Sybase Failover in jConnect and connect a client application to an Adaptive Server configured for failover. If the primary server fails over to the secondary server, the client application also automatically switches to the second server and reestablishes network connections.

Note Refer to *Using Sybase Failover in A High Availability System* for more detailed information about Sybase Failover.

Requirements, Dependencies, and Restrictions

- You must have two version 12.0 Adaptive Servers configured for Failover.
- You must use jConnect 4.2 or jConnect 5.2. Previous driver versions do not support this feature.
- Only changes that were committed to the database before failover are retained when the client fails over.
- The client application connection must be made using JNDI. See “Connecting to a Server Using JNDI” on page 20.
- jConnect event notification does not work when failover occurs. See “Using Event Notification” on page 62.
- Close all statements when they are no longer used. jConnect stores information on statements to enable failover. If you do not close statements, you will experience memory leaks.

Implementing Failover in jConnect

To implement failover support in jConnect:

1 Configure the primary and secondary Adaptive Servers for failover.

2 Include an entry for the primary server and a separate entry for the secondary server in the directory service information file required by JNDI. The primary server entry will have an attribute (the HA OID) that refers to the entry for the secondary server.

Using LDAP as the service provider for JNDI, there are three possible forms that this HA attribute can have:

- *Relative Distinguished Name (RDN)* – This form assumes that the search base (typically provided by the **java.naming.provider.url** attribute) combined with this attribute's value is enough to identify the secondary server. For example, assume the primary server is at hostname:4200 and the secondary server is at hostname:4202:

```
dn: servername=haprimary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: servername=hasecondary
objectclass: sybaseServer


dn: servername=hasecondary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

- *Distinguished Name (DN)* – This form assumes that the HA attribute's value uniquely identifies the secondary server, and may or may not duplicate values found in the search base. For example:

```
dn: servername=haprimary, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: servername=hasecondary,
      o=Sybase, c=US ou=Accounting
objectclass: sybaseServer


dn: servername=hasecondary, o=Sybase, c=US, ou=Accounting
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

Notice that **hasecondary** is located in a different branch of the tree (see the additional **ou=Accounting** qualifier).

- *Full LDAP URL* – This form assumes nothing about the search base. The HA attribute is expected to be a fully-qualified LDAP URL that is used to identify the secondary (it may even point to a different LDAP server). For example:

```
dn: servername=hafailover, o=Sybase, c=US
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4200
1.3.6.1.4.1.897.4.2.15: ldap://ldapserver:386/servername=secondary,
      o=Sybase, c=US ou=Accounting
objectclass: sybaseServer

dn: servername=secondary, o=Sybase, c=US, ou=Accounting
1.3.6.1.4.1.897.4.2.5: TCP#1#hostname 4202
objectclass: sybaseServer
```

3 In the directory service information file required by JNDI, set the REQUEST_HA_SESSION connection property to enable a failover session every time you make a connection.

The new REQUEST_HA_SESSION connection property is used to indicate that the connecting client wants to begin a failover session with the version 12.0 Adaptive Server configured for failover. Setting this property to "true" causes jConnect to attempt a failover login. If you do not set this connection property, a failover session will not start, even if the server is configured correctly. The default value for REQUEST_HA_SESSION is "false."

Set the connection property like any other connection property. You cannot reset the property once a connection has been made.

If you want more flexibility for requesting failover sessions, code the client application to set REQUEST_HA_SESSION at runtime.

The following example shows connection information entered for the database server SYBASE11 under an LDAP directory service:

```
dn: servername=SYBASE11,o=MyCompany,c=US
1.3.6.1.4.1.897.4.2.5:TCP#1#tahiti 3456
1.3.6.1.4.1.897.4.2.10:REPEAT_READ=false&PACKETSIZE=1024
1.3.6.1.4.1.897.4.2.10:CONNECTION_FAILOVER=false
1.3.6.1.4.1.897.4.2.11:pubs2
1.3.6.1.4.1.897.4.2.9:Tds
1.3.6.1.4.1.897.4.2.15:servername=SECONDARY
1.3.6.1.4.1.897.4.2.10:REQUEST_HA_SESSION=true

dn:servername=SECONDARY, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1#moorea 6000
```

where "tahiti" is the primary server and "moorea" is the secondary companion server.

4 Request a connection using JNDI and LDAP.

- jConnect uses the LDAP server's directory server to determine the name and location of the primary and secondary servers:

```
/* get the connection */
Connection con = DriverManager.getConnection
      ("jdbc:sybase:jndi:ldap://ldap_server1:983" +
      "/servername=Sybase11,o=MyCompany,c=US",props);
```

or

- Specify a searchbase:

```
props.put(Context.PROVIDER_URL,
      "ldap://ldap_server1:983/ o=MyCompany, c=US");

Connection con=DriverManager.getConnection
      ("jdbc:sybase:jndi:servername=Sybase11", props);
```

Logging In to the Primary Server

If an Adaptive Server is not configured for failover, or for some reason cannot grant a failover session, the client cannot log in, and the following warning displays:

```
'The server denied your request to use the high-
availability feature.

Please reconfigure your database, or do not request a
high-availability session.'
```

Failing Over to the Secondary Server

When failover occurs, an exception is thrown and the client automatically reconnects to the secondary database using JNDI.

Note that:

- The identity of the database to which the client was connected and any committed transactions are retained.
- Partially read result sets, cursors, and stored procedure invocations are lost.
- When failover occurs, your application may need to restart a procedure or go back to the last completed transaction or activity.

Failing Back to the Primary Server

At some point, the client will fail back from the secondary server to the primary server. When failback occurs is determined by the System Administrator who issues **sp_failback** on the secondary server. Afterward, the client can expect the same behavior and results on the primary server as documented in "Failing Over to the Secondary Server" on page 40

Performing Server-to-Server Remote Procedure Calls

A Transact-SQL language command or stored procedure running on one server can execute a stored procedure located on another server. The server to which an application has connected logs in to the remote server, and executes a server-to-server remote procedure call.

An application can specify a "universal" password for server-to-server communication; that is, a password used in all server-to-server connections. Once the connection is open, the server uses this password to log in to any remote server.

By default, jConnect uses the current connection's password as the default password for server-to-server communications.

However, if the passwords are different on two servers for the same user and that user is performing server-to-server remote procedure calls, the application must explicitly define passwords for each server it plans to use.

jConnect version 4.1 and later include a property that lets you set a universal "remote" password or different passwords on several servers. jConnect lets you set and configure the property using the **setRemotePassword()** method in the **SybDriver** class:

```
Properties connectionProps = new Properties();

public final void setRemotePassword(String serverName,
       String password, Properties connectionProps)
```

To use this method, the application needs to import the **SybDriver** class, then call the method.

For jConnect 4.x:

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
       Class.forName("com.sybase.jdbc.SybDriver").newInstance();
sybDriver.setRemotePassword
       (serverName, password, connectionProps);
```

For jConnect 5.x:

```
import com.sybase.jdbcx.SybDriver;
SybDriver sybDriver = (SybDriver)
      Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
sybDriver.setRemotePassword
      (serverName, password, connectionProps);
```

Note To set different remote passwords for various servers, repeat the preceding call (appropriate for your version of jConnect) for each server.

This call adds the given server name-password pair to the given **Properties** object, which can be passed by the application to **DriverManager** in **DriverManager.getConnection** (*server_url, props*).

If **serverName** is NULL, the universal password will be set to **password** for subsequent connections to all servers except the ones specifically defined by previous calls to **setRemotePassword( )**.

When an application sets the REMOTEPWD property, jConnect no longer sets the default universal password.

Accessing Database Metadata

To support JDBC **DatabaseMetaData** methods, Sybase provides a set of stored procedures that jConnect can call for metadata about a database. These stored procedures must be installed on the server for the JDBC metadata methods to work.

If the stored procedures for providing metadata are not already installed in a Sybase server, you can install them using stored procedure scripts provided with jConnect:

- *sql_server.sql* installs stored procedures on pre-version 12.0 Adaptive Server databases.
- *sql_server12.sql* installs stored procedures on an Adaptive Server version 12.0 database.

- *sql_anywhere.sql* installs stored procedures on an Adaptive Server Anywhere database.

Note The most recent version of these scripts is compatible with all versions of jConnect.

See the *Sybase jConnect for JDBC Installation Guide and Release Bulletin* for complete instructions on installing stored procedures.

In addition, to use the metadata methods, you must set the USE_METADATA connection property to “true” (its default value) when you establish a connection.

You cannot get metadata about temporary tables in a database.

Note The **DatabaseMetaData.getPrimaryKeys( )** method finds primary keys declared in a table definition (CREATE TABLE) or with alter table (ALTER TABLE ADD CONSTRAINT). It does not find keys defined using **sp_primarykey.**

Server-Side Metadata Installation

Metadata support can be implemented in either the client (ODBC, JDBC) or in the data source (server stored procedures). jConnect provides metadata support on the server, which results in the following benefits:

- Maintains jConnect’s small size, which ensures the driver can be quickly downloaded from the Internet.
- Gains runtime efficiency from preloaded stored procedures on the data source.
- Provides flexibility—jConnect can connect to a variety of databases.

Using Cursors with Result Sets

jConnect 5.x implements many JDBC 2.0 cursor and update methods. These methods make it easier to use cursors and to update rows in a table based on values in a result set.

Note To have full JDBC 2.0 support, use jConnect version 5.x or later. jConnect version 4.x provides some JDBC 2.0 features via Sybase extensions and the **ScrollableResultSet.java** sample found in the *sample* subdirectory under your jConnect directory. See the **com.sybase.jdbcx** and the **sample** packages for the javadocs on these methods.

In JDBC 2.0, **ResultSets** are characterized by their type and their concurrency. The type and concurrency values are part of the **java.sql.ResultSet** interface and are described in its javadocs.

Table 2-5 identifies the characteristics of **java.sql.ResultSet** that are available in jConnect 5.x.

Table 2-5: java.sql.ResultSet options available in jConnect 5.x

	Type		
Concurrency	**TYPE_FORWARD_ ONLY**	**TYPE_SCROLL_ INSENSITIVE**	**TYPE_SCROLL_ SENSITIVE**
CONCUR_READ_ONLY	Supported in 5.x	Supported in 5.x	Not available in 5.x
CONCUR_UPDATABLE	Supported in 5.x	Not available in 5.x	Not available in 5.x

This section includes the following topics:

- Creating a Cursor
- Positioned Updates and Deletes Using JDBC 1.x Methods
- Using a Cursor with a PreparedStatement
- Support for SCROLL_INSENSITIVE Result Sets in jConnect

Creating a Cursor

To create a cursor using jConnect 4.x, use either **SybStatement.setCursorName( )** or **SybStatement.setFetchSize( )**. When you use **SybStatement.setCursorName( )**, you explicitly assign the cursor a name. The signature for **SybStatement.setCursorName( )** is:

```
void setCursorName(String name) throws SQLException;
```

You use **SybStatement.setFetchSize( )** to create a cursor and specify the number of rows returned from the database in each fetch. The signature for **SybStatement.setFetchSize( )** is:

```
void setFetchSize(int rows) throws SQLException;
```

When you use **setFetchSize( )** to create a cursor, the jConnect driver names the cursor. To get the cursor's name, use **ResultSet.getCursorName( )**.

You create cursors in jConnect version 5.x the same way as in version 4.x, but because version 5.x supports JDBC 2.0, there is another way to create cursors. You can specify which kind of **ResultSet** you want returned by the statement, using the following JDBC 2.0 method on the connection:

```
Statement createStatement(int resultSetType, int
resultSetConcurrency)throws SQL Exception
```

The type and concurrency correspond to the types and concurrences found on the **ResultSet** interface listed in Table 2-5. If you request an unsupported **ResultSet**, a SQL warning is chained to the connection. When the returned **Statement** is executed, you will receive the kind of **ResultSet** that is most like the one you requested. See the JDBC 2.0 specification for more details on this method's behavior.

If you do not use **createStatement( ),** or you are using jConnect version 4.x, the default types of **ResultSet** are:

- If you call only **Statement.executeQuery( )**, then the **ResultSet** returned is a **SybResultSet** that is TYPE_FORWARD_ONLY and CONCUR_READ_ONLY.
- If you call **setFetchSize( )** or **setCursorName( )**, then the **ResultSet** returned from **executeQuery(**) is a **SybCursorResultSet** that is TYPE_FORWARD_ONLY and CONCUR_UPDATABLE.

To verify that the kind of **ResultSet** object is what you intended, the JDBC 2.0 API for **ResultSet** has added two methods:

```
int getConcurrency() throws SQLException;
int getType() throws SQLException;
```

The basic steps for creating and using a cursor are:

1. Create the cursor using **Statement.setCursorName( )** or **SybStatement.setFetchSize( )**.
2. Invoke **Statement.executeQuery( )** to open the cursor for a statement and return a cursor result set.

3 Invoke **ResultSet.next( )** to fetch rows and position the cursor in the result set.

The following example uses each of the two methods for creating cursors and returning a result set. It also uses **ResultSet.getCursorName( )** to get the name of the cursor created by **SybStatement.setFetchSize( )**.

```
// With conn as a Connection object, create a
// Statement object and assign it a cursor using
// Statement.setCursorName().
Statement stmt = conn.createStatement();
stmt.setCursorName("author_cursor");

// Use the statement to execute a query and return
// a cursor result set.
ResultSet rs = stmt.executeQuery("SELECT au_id,
   au_lname, au_fname FROM authors
   WHERE city = ’Oakland’");
while(rs.next())
{
...
}

// Create a second statement object and use
// SybStatement.setFetchSize()to create a cursor
// that returns 10 rows at a time.
SybStatement syb_stmt = conn.createStatement();
syb_stmt.setFetchSize(10);

// Use the syb_stmt to execute a query and return
// a cursor result set.
SybCursorResultSet rs2 =
   (SybCursorResultSet)syb_stmt.executeQuery
   ("SELECT au_id, au_lname, au_fname FROM authors
   WHERE city = ’Pinole’");
while(rs2.next())
{
...
}

// Get the name of the cursor created through the
// setFetchSize() method.
String cursor_name = rs2.getCursorName();
 ...
// For jConnect 5.x, create a third statement
// object using the new method on Connection,
// and obtain a SCROLL_INSENSITIVE ResultSet.
```

```
   // Note: you no longer have to downcast the
   // Statement or the ResultSet.
   Statement stmt = conn.createStatement(

   ResultSet.TYPE_SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_READ_ONLY);
   ResultSet rs3 = stmt.executeQuery
    ("SELECT ... [whatever]");
   // Execute any of the JDBC 2.0 methods that
   // are valid for read only ResultSets.
   rs3.next();
   rs3.previous();
   rs3.relative(3);
   rs3.afterLast();
   ...
```

Positioned Updates and Deletes Using JDBC 1.x Methods

The following example shows how to use methods in JDBC 1.x to do a positioned update. The example creates two **Statement** objects, one for selecting rows into a cursor result set, and the other for updating the database from rows in the result set.

```
// Create two statement objects and create a cursor
// for the result set returned by the first
// statement, stmt1. Use stmt1 to execute a query
// and return a cursor result set.
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
stmt1.setCursorName("author_cursor");
ResultSet rs = stmt1.executeQuery("SELECT
   au_id,au_lname, au_fname
   FROM authors WHERE city = ’Oakland’
   FOR UPDATE OF au_lname");

// Get the name of the cursor created for stmt1 so
// that it can be used with stmt2.
String cursor = rs.getCursorName();

// Use stmt2 to update the database from the
// result set returned by stmt1.
String last_name = new String("Smith");
while(rs.next())
{
   if (rs.getString(1).equals("274-80-9391"))
     {
```

```
                 stmt2.executeUpdate("UPDATE authors "+
                 "SET au_lname = "+last_name +
                 "WHERE CURRENT OF " + cursor);
             }
         }
```

Deletions in a Result Set

The following example uses **Statement** object *stmt2*, from the preceding code, to perform a positioned deletion:

```
   stmt2.executeUpdate("DELETE FROM authors
            WHERE CURRENT OF " + cursor);
```

Positioned Updates and Deletes Using JDBC 2.0 Methods

This section discusses JDBC 2.0 methods for updating columns in the current cursor row and updating the database from the current cursor row in a result set. They are followed by an example.

Updating Columns in a Result Set

JDBC 2.0 specifies a number of methods for updating column values from a result set in memory, on the client. The updated values can then be used to perform an update, insert, or delete operation on the underlying database. All of these methods are implemented in the **SybCursorResultSet** class.

Examples of some of the JDBC 2.0 update methods available in jConnect are:

```
void updateAsciiStream(String columnName, java.io.InputStream x,
   int length) throws SQLException;
void updateBoolean(int columnIndex, boolean x) throws
   SQLException;
void updateFloat(int columnIndex, float x) throws SQLException;
void updateInt(String columnName, int x) throws SQLException;
void updateInt(int columnIndex, int x) throws SQLException;
void updateObject(String columnName, Object x) throws
   SQLException;
```

Methods for Updating the Database from a Result Set

JDBC 2.0 specifies two new methods for updating or deleting rows in the database, based on the current values in a result set. These methods are simpler in form than **Statement.executeUpdate( )** in JDBC 1.x and do not require a cursor name. They are implemented in **SybCursorResultSet**:

```
void updateRow() throws SQLException;
void deleteRow() throws SQLException;
```

Note The concurrency of the result set must be CONCUR_UPDATABLE, otherwise the above methods will raise an exception. For **insertRow( )**, all table columns that require non-null entries must be specified.

Methods provided on **DatabaseMetaData** dictate when these changes are visible.

Example

The following example creates a single **Statement** object that is used to return a cursor result set. For each row in the result set, column values are updated in memory and then the database is updated with the row's new column values.

```
// Create a Statement object and set fetch size to
// 25. This creates a cursor for the Statement
// object Use the statement to return a cursor
// result set.
SybStatement syb_stmt =
(SybStatement)conn.createStatement();
syb_stmt.setFetchSize(25);
SybCursorResultSet syb_rs =
(SybCursorResultSet)syb_stmt.executeQuery(
   "SELECT * from T1 WHERE ...")

// Update each row in the result set according to
// code in the following while loop. jConnect
// fetches 25 rows at a time, until fewer than 25
// rows are left. Its last fetch takes any
// remaining rows.
while(syb_rs.next())
{
// Update columns 2 and 3 of each row, where
// column 2 is a varchar in the database and
// column 3 is an integer.
syb_rs.updateString(2, "xyz");
syb_rs.updateInt(3,100);
//Now, update the row in the database.
syb_rs.updateRow();
}
// Create a Statement object using the
// JDBC 2.0 method implemented in jConnect 5.x
Statement stmt = conn.createStatement
(ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_UPDATABLE);
// Use the Statement to return an updatable ResultSet
```

```
ResultSet rs = stmt.executeQuery(“SELECT * FROM T1 WHERE...”);
// In jConnect 5.x, downcasting to SybCursorResultSet is not
// necessary. Update each row in the ResultSet in the same
// manner as above
while (rs.next())
{
rs.updateString(2, “xyz”);
rs.updateInt(3,100);
rs.updateRow();
}
```

Deleting a Row from a Result Set

To delete a row from a cursor result set, you can use **SybCursorResultSet.deleteRow( )** as follows:

```
while(syb_rs.next())
{
  int col3 = getInt(3);
  if (col3 >100)
  {
  syb_rs.deleteRow();
  }
}
```

Inserting a Row Into a Result Set

The following example illustrates how to do inserts using the JDBC 2.0 API, which is only available in jConnect 5.x. There is no need to downcast to a **SybCursorResultSet**.

```
// prepare to insert
rs.moveToInsertRow();
// populate new row with column values
rs.updateString(1, "New entry for col 1");
rs.updateInt(2, 42);
// insert new row into db
rs.insertRow();
// return to current row in result set
rs.moveToCurrentRow();
```

Using a Cursor with a PreparedStatement

Once you create a **PreparedStatement** object, you can use it multiple times with the same or different values for its input parameters. If you use a cursor with a **PreparedStatement** object, you need to close the cursor after each use and then reopen the cursor to use it again. A cursor is closed when you close its result set (**ResultSet.close( )**). It is opened when you execute its prepared statement (**PreparedStatement.executeQuery( )**).

The following example shows how to create a **PreparedStatement** object, assign it a cursor, and execute the **PreparedStatement** object twice, closing and then reopening the cursor.

```
// Create a prepared statement object with a
// parameterized query.
PreparedStatement prep_stmt =
conn.prepareStatement(
"SELECT au_id, au_lname, au_fname "+
"FROM authors WHERE city = ? "+
"FOR UPDATE OF au_lname");

//Create a cursor for the statement.
prep_stmt.setCursorName("author_cursor");

// Assign the parameter in the query a value.
// Execute the prepared statement to return a
// result set.
prep_stmt.setString(1, "Oakland");
ResultSet rs = prep_stmt.executeQuery();

//Do some processing on the result set.
while(rs.next())
{
    ...
}

// Close the result, which also closes the cursor.
rs.close();

// Execute the prepared statement again with a new
// parameter value.
prep_stmt.setString(1,"San Francisco");
rs = prep_stmt.executeQuery();
// reopens cursor
```

Support for SCROLL_INSENSITIVE Result Sets in jConnect

jConnect version 5.x supports only TYPE_SCROLL_INSENSITIVE result sets.

jConnect uses the Tabular Data Stream (TDS)—Sybase's proprietary protocol—to communicate with Sybase database servers. As of jConnect 5.x, TDS does not support scrollable cursors. To support scrollable cursors, jConnect caches the row data on demand, on the client, on each call to **ResultSet.next( )**. However, when the end of the result set is reached, the entire result set is stored in the client's memory. Because this may cause a performance strain, we recommend that you use TYPE_SCROLL_INSENSITIVE result sets only when the result set is reasonably small.

Note When you use TYPE_SCROLL_INSENSITIVE **ResultSets** in jConnect 5.x, you can only call the **isLast( )** method after the last row of the **ResultSet** has been read. Calling **isLast( )** before the last row is reached will cause an **UnimplementedOperationException** to be thrown.

A sample has been added to jConnect version 4.x that provides a limited TYPE_SCROLL_INSENSITIVE **ResultSet** using JDBC 1.0 interfaces.

This implementation uses standard JDBC 1.0 methods to produce a scroll-insensitive, read-only result set; that is, a static view of the underlying data that is not sensitive to changes made while the result set is open. **ExtendedResultSet** caches all of the **ResultSet** rows on the client. Be cautious when you use this class with large result sets.

The **sample.ScrollableResultSet** interface:

- Is an extension of JDBC 1.0 **java.sql.ResultSet**.
- Defines additional methods that have the same signatures as the JDBC 2.0 **java.sql.ResultSet**.
- Does *not* contain all of the JDBC 2.0 methods. The missing methods deal with modifying the **ResultSet**.

The methods from the JDBC 2.0 API that *are*

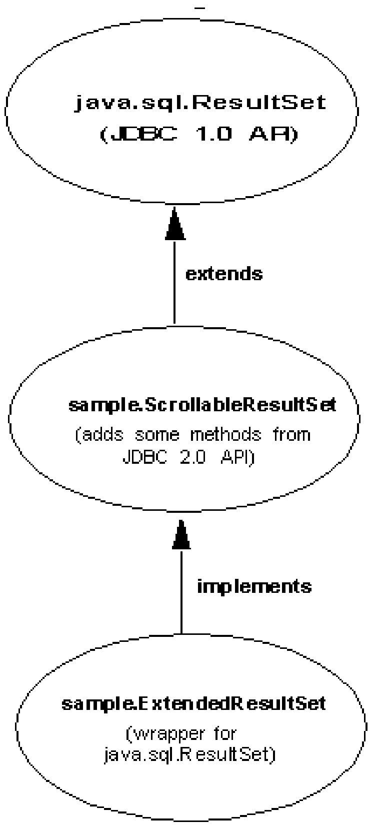
```
boolean previous() throws SQLException;
boolean absolute(int row) throws SQLException;
boolean relative(int rows) throws SQLException;
boolean first() throws SQLException;
boolean last() throws SQLException;
void beforeFirst() throws SQLException;
void afterLast() throws SQLException;
boolean isFirst() throws SQLException;
boolean isLast() throws SQLException;
boolean isBeforeFirst() throws SQLException;
boolean isAfterLast() throws SQLException;
int getFetchSize() throws SQLException;
void setFetchSize(int rows) throws SQLException;
int getFetchDirection() throws SQLException;
void setFetchDirection(int direction) throws SQLException;
int getType() throws SQLException;
int getConcurrency() throws SQLException;
int getRow() throws SQLException;
```

To use the new sample classes, create an **ExtendedResultSet** using any JDBC 1.0 **java.sql.ResultSet**. Below are the relevant pieces of code (assume a Java 1.1 environment):

```
// import the sample files
import sample.*;
//import the JDBC 1.0 classes
import java.sql.*;
// connect to some db using some driver;
// create a statement and a query;
// Get a reference to a JDBC 1.0 ResultSet
ResultSet rs = stmt.executeQuery(_query);
// Create a ScrollableResultSet with it
ScrollableResultSet srs = new ExtendedResultSet(rs);
// invoke methods from the JDBC 2.0 API
srs.beforeFirst();
// or invoke methods from the JDBC 1.0 API
if (srs.next())
   String column1 = srs.getString(1);
```

Figure 2-1 is a class diagram that shows the relationships between the new sample classes and the JDBC API.

Figure 2-1: Class diagram



See the JDBC 2.0 API at *http://java.sun.com/products/jdbc/jdbcse2.html* for more details.

Support for Batch Updates

Batch updates allow a **Statement** object to submit multiple update commands as one unit (batch) to an underlying database for processing together.

Note To use batch updates, you must refresh the SQL scripts in the *sp* directory under your jConnect installation directory.

See *BatchUpdates.java* in the *sample* (jConnect 4.x) and *sample2* (jConnect 5.x) subdirectories for an example of using batch updates with **Statement**, **PreparedStatement**, and **CallableStatement**.

jConnect also supports dynamic **PreparedStatements** in batch.

Implementation Notes

jConnect implements batch updates as specified in the JDBC 2.0 API, except as described below.

- If the JDBC 2.0 standard for implementing **BatchUpdateException.getUpdateCounts( )** is modified or relaxed in the future, jConnect will continue to implement the original standard by having **BatchUpdateException.getUpdateCounts( )** return an **int[ ]** length of M < N, indicating that the first M statements in the batch succeeded, that the M+1 statement failed, and M+2..N statements were not executed; where "N" equals the total statements in the batch.
- Batch updates of stored procedures

  To call stored procedures in batch (unchained) mode, you must create the stored procedure in unchained mode. For more information, see "Stored Procedure Executed in Unchained Transaction Mode" on page 103.
- Adaptive Server Enterprise version 11.5.x and later

  BatchUpdateException.getUpdateCounts( ) will return only an **int[ ]** length of zero. The entire transaction is rolled back if an error is encountered, resulting in zero successful rows.
- Adaptive Server Enterprise version 11.0.1

  Returns 0 (zero) rows affected for stored procedures.
- SQL Anywhere version 5.5.x
  - SQL Anywhere version 5.5.x does not allow you to obtain inserted row counts from stored procedures that contain inserts. For example:

    ```
    create proc sp_A as insert tableA values (1,
    ‘hello A’)
    create proc sp_B
    as
    insert tableA values (1, ‘hello A’)
    update tableA set col1=2
    create proc sp_C
    as
    ```

```
update tableA set col1=2
delete tableA
```

Running **executeBatch** on the preceding stored procedures would result in, respectively:

```
0 Rows Affected
1 Rows Affected
2 Rows Affected
```

- There is no support for dynamic **PreparedStatements** in batch.
- Because SQL Anywhere 5.5.x does not natively support batch updates according to the JDBC 2.0 specification, batch updates are carried out in an **executeUpdate** loop.

- Batch updates in databases that do not support batch updates

  jConnect carries out batch updates in an **executeUpdate** loop even if your database does not support batch updates. This allows you to use the same batch code, regardless of the database to which you are pointing.

See *Sun Microsystems, Inc. JDBC™ 2.0 API* for more details on batch updates.

Updating the Database from the Result Set of a Stored Procedure

jConnect includes update and delete methods that allow you to get a cursor on the result set returned by a stored procedure. You can then use the cursor's position to update or delete rows in the underlying table that provided the result set. The methods are in **SybCursorResultSet**:

void updateRow(String *tableName*) throws SQLException;

void deleteRow(String *tableName*) throws SQLException;

The *tableName* parameter identifies the database table that provided the result set.

To get a cursor on the result set returned by a stored procedure, you need to use either **SybCallableStatement.setCursorName( )** or **SybCallableStatement.setFetchSize( )** before you execute the callable statement that contains the procedure. The following example shows how to create a cursor on the result set of a stored procedure, update values in the result set, and then update the underlying table using the **SybCursorResultSet**.**update( )** method:

```
// Create a CallableStatement object for executing the stored
// procedure.
```

```
CallableStatement sproc_stmt =
   conn.prepareCall("{call update_titles}");

// Set the number of rows to be returned from the database with
// each fetch. This creates a cursor on the result set.
(SybCallableStatement)sproc_stmt.setFetchSize(10);

//Execute the stored procedure and get a result set from it.
SybCursorResultSet sproc_result = (SybCursorResultSet)
   sproc_stmt.executeQuery();

// Move through the result set row by row, updating values in the
// cursor’s current row and updating the underlying titles table
// with the modified row values.
while(sproc_result.next())
{
   sproc_result.updateString(...);
   sproc_result.updateInt(...);
   ...
   sproc_result.updateRow(titles);
}
```

Working with Datatypes

Sending *Image* Data

jConnect has a **TextPointer** class with **sendData( )** methods for updating an *image* column in an Adaptive Server Enterprise or Adaptive Server Anywhere database. In earlier versions of jConnect, you had to send image data using the **setBinaryStream( )** method in **java.sql.PreparedStatement**. The **TextPointer.sendData( )** methods use **java.io.InputStream** and greatly improve performance when you send image data to an Adaptive Server database.

To obtain instances of the **TextPointer** class, you can use either of two **getTextPtr( )** methods in **SybResultSet**:

```
public TextPointer getTextPtr(String columnName)
public TextPointer getTextPtr(int columnIndex)
```

Public Methods in the TextPointer Class

The **com.sybase.jdbc** package contains the **TextPointer** class. Its public method interface is:

```
public void sendData(InputStream is, boolean log)
     throws SQLException

public void sendData(InputStream is, int length,
     boolean log) throws SQLException

public void sendData(InputStream is, int offset,
     int length, boolean log) throws SQLException

public void sendData(byte[] byteInput, int offset,
     int length, boolean log) throws SQLEXception
```

sendData(InputStream *is*, **boolean** *log)* – Updates an image column with data in the specified input stream.

sendData(InputStream *is*, **int** *length*, **boolean** *log)* – Updates an image column with data in the specified input stream. *length* is the number of bytes being sent.

sendData(InputStream *is*, **int** *offset*, **int** *length*, **boolean** *log)* – Updates an image column with data in the specified input stream, starting at the byte offset given in the *offset* parameter and continuing for the number of bytes specified in the *length* parameter.

sendData(byte[ ] *byteInput*, **int** *offset*, **int** *length*, **boolean** *log)* – Updates a column with image data contained in the byte array specified in the *byteInput* parameter. The update starts at the byte offset given in the *offset* parameter and continues for the number of bytes specified in the *length* parameter.

Each method has a *log* parameter. The *log* parameter specifies whether image data is to be fully logged in the database transaction log. If the *log* parameter is set to "true," the entire binary image is written into the transaction log. If the *log* parameter is set to "false," the update is logged, but the image itself is not included in the log.

v Updating an *Image* Column with *TextPointer.sendData( )*

To update a column with image data:

1 Get a **TextPointer** object for the row and column that you want to update.

2 Use **TextPointer.sendData( )** to execute the update.

The next two sections illustrate these steps with an example. In the example, image data from the file *Anne_Ringer.gif* is sent to update the *pic* column of the *au_pix* table in the *pubs2* database. The update is for the row with author ID 899-46-2035.

Getting a *TextPointer* Object

text and image columns contain timestamp and page-location information that is separate from their text and image data. When data is selected from a text or image column, this extra information is “hidden” as part of the result set.

A **TextPointer** object for updating an image column requires this hidden information, but does not need the image portion of the column data. To get this information, you need to select the column into a **ResultSet** object and then use **SybResultSet.getTextPtr( )** (see the example that follows the next paragraph). **SybResultSet.getTextPtr( )** extracts text-pointer information, ignores image data, and creates a **TextPointer** object.

In cases where a column contains a significant amount of image data, selecting the column for one or more rows and waiting to get all the data is likely to be inefficient, since the data is not used. You can shortcut this process by using the **set textsize** command to minimize the amount of data returned in a packet. The following code example for getting a **TextPointer** object includes the use of **set textsize** for this purpose.

```
/*
 * Define a string for selecting pic column data for author ID
 * 899-46-2035.
 */
 String getColumnData = "select pic from au_pix where au_id = ’899-46-2035’";

 /*
 * Use set textsize to return only a single byte of column data
 * to a Statement object. The packet with the column data will
 * contain the "hidden" information necessary for creating a
 * TextPointer object.
 */
 Statement stmt= connection.createStatement();
 stmt.executeUpdate("set textsize 1");

 /*
 * Select the column data into a ResultSet object--cast the
 * ResultSet to SybResultSet because the getTextPtr method is
 * in SybResultSet, which extends ResultSet.
 */
 SybResultSet rs = (SybResultSet)stmt.executeQuery(getColumnData);

 /*
```

```
* Position the result set cursor on the returned column data
* and create the desired TextPointer object.
*/
rs.next();
TextPointer tp = rs.getTextPtr("pic");

/*
* Now, assuming we are only updating one row, and won’t need
* the minimum textsize set for the next return from the server,
* we reset textsize to its default value.
*/
stmt.executeUpdate("set textsize 0");
```

Executing the Update with *TextPointer.sendData*

The following code uses the **TextPointer** object from the preceding section to update the *pic* column with image data in the file *Anne_Ringer.gif*.

```
/*
 * First, define an input stream for the file.
 */
 FileInputStream in = new FileInputStream("Anne_Ringer.gif");

 /*
 * Prepare to send the input stream without logging the image data
 * in the transaction log.
 */
 boolean log = false;

 /*
 * Send the image data in Anne_Ringer.gif to update the pic
 * column for author ID 899-46-2035.
 */
 tp.sendData(in, log);
```

See the *TextPointers.java* sample in the *sample* (jConnect 4.x) and *sample2* (jConnect 5.x) subdirectories under your jConnect installation directory for more information.

Using *Date* and *Time* Datatypes

JDBC uses three temporal datatypes: Time, Date, and Timestamp. Adaptive Server uses only one temporal datatype, datetime, which is equivalent to the JDBC Timestamp datatype. The Adaptive Server datetime datatype supports second resolution to 1/300th of a second.

All three JDBC datatypes are treated as datetime datatypes on the server side. A JDBC Timestamp is essentially the same as a server datetime; therefore, no conversion is necessary. However, translating a JDBC Time or Date datatype to or from a server datetime datatype requires a conversion.

- To convert Time to datetime, the date 1 Jan 1970 is added.
- To convert Date to datetime, "00:00:00" is appended.
- To convert a datetime to a Date variable or a Time variable, the unused information is stripped out.

Implementation Notes

- JDBC's Timestamp datatype is not the same as Adaptive Server's timestamp datatype. The Adaptive Server timestamp datatype is a unique varbinary value used when updates are made with an "optimistic concurrency" strategy.
- When a value is inserted as a Time datatype, the date portion is essentially meaningless, so the value should be fetched back using only a Time datatype, never a Date or Timestamp datatype.
- If you use **getObject( )** with an Adaptive Server Anywhere *date* or time *column*, the value will be returned as a JDBC *Timestamp* datatype.

Char/*Varchar*/*Text* Datatypes and *getByte( )*

Do not use**rs.getByte( )** on a char, varchar, or text field unless the data is hex, octal, or decimal.

Implementing Advanced Features

This section describes how to use advanced jConnect features and contains the following topics:

- Using Event Notification
- Handling Error Messages
- Storing Java Objects as Column Data in a Table
- Dynamic Class Loading
- JDBC 2.0 Optional Package Extensions Support

Using Event Notification

You can use the jConnect event notification feature to have your application notified when an Open Server procedure is executed.

To use this feature, you must use the **SybConnection** class, which extends the **Connection** interface. **SybConnection** contains a **regWatch( )** method for turning event notification on and a **regNoWatch( )** method for turning event notification off.

Your application must also implement the **SybEventHandler** interface. This interface contains one public method, **void event(String proc_name, ResultSet params)**, which is called when the specified event occurs. The parameters of the event are passed to **event( )** and it tells the application how to respond.

To use event notification in your application, call **SybConnection.regWatch( )** to register your application in the notification list of a registered procedure. Use this syntax:

SybConnection.regWatch(*proc_name*,*eventHdlr*,*option*)

- *proc_name* is a String that is the name of the registered procedure that generates the notification.
- *eventHdler* is an instance of the **SybEventHandler** class that you implement.
- *option* is either NOTIFY_ONCE or NOTIFY_ALWAYS. Use NOTIFY_ONCE if you want the application to be notified only the first time a procedure executes. Use NOTIFY_ALWAYS if you want the application to be notified every time the procedure executes.

Whenever an event with the designated *proc_name* occurs on the Open Server, jConnect calls **eventHdlr.event( )** from a separate thread. The event parameters are passed to **eventHdlr.event( )** when it is executed. Because it is a separate thread, event notification does not block execution of the application.

If *proc_name* is not a registered procedure, or if Open Server is unable to add the client to the notification list, the call to **regWatch( )** throws a SQL exception.

To turn off event notification, use this call:

```
SybConnection.regNoWatch(proc_name)
```

Note When you use Sybase event notification extensions, the application needs to call the **close( )** method on the connection to remove a child thread created by the first call to **regWatch( )**. Failing to do so may cause the Virtual Machine to hang when exiting the application.

Event Notification Example

The following example shows how to implement an event handler and then register an event with an instance of your event handler, once you have a connection:

```
public class MyEventHandler implements SybEventHandler
{
   // Declare fields and constructors, as needed.
   ...
   public MyEventHandler(String eventname)
   {
      ...
   }

   // Implement SybEventHandler.event.
   public void event(String eventName, ResultSet params)
   {
      try
      {
         // Check for error messages received prior to event
         // notification.
         SQLWarning sqlw = params.getWarnings();
         if sqlw != null
         {
            // process errors, if any
            ...
         }
         // process params as you would any result set with
         // one row.
         ResultSetMetaData rsmd = params.getMetaData();
         int numColumns = rsmd.getColumnCount();
         while (params.next())          // optional
         {
            for (int i = 1; i <= numColumns; i++)
            {
               System.out.println(rsmd.getColumnName(i) + " =
                  " + params.getString(i));
            }
            // Take appropriate action on the event. For example,
            // perhaps notify application thread.
            ...
         }
      }
      catch (SQLException sqe)
      {
         // process errors, if any
         ...
      }
   }
}

public class MyProgram
{
   ...
   // Get a connection and register an event with an instance
   // of MyEventHandler.
   Connection conn = DriverManager.getConnection(...);
   MyEventHandler myHdlr = new  MyEventHandler("MY_EVENT");

   // Register your event handler.
   ((SybConnection)conn).regWatch("MY_EVENT", myHdlr,
      SybEventHandler.NOTIFY_ALWAYS);
   ...
   conn.regNoWatch("MY_EVENT");
   conn.close();
}
```

Handling Error Messages

jConnect provides two classes for returning Sybase-specific error information, **SybSQLException** and **SybSQLWarning**, as well as a **SybMessageHandler** interface that allows you to customize the way jConnect handles error messages received from the server.

Retrieving Sybase-Specific Error Information

jConnect provides an **EedInfo** interface that specifies methods for obtaining Sybase-specific error information. The **EedInfo** interface is implemented in **SybSQLException** and **SybSQLWarning**, which extend the **SQLException** and **SQLWarning** classes.

SybSQLException and **SybSQLWarning** contain the following methods:

- **public ResultSet getEedParams( );**

  Returns a one-row result set containing any parameter values that accompany the error message.

- **public int getStatus( );**

  Returns a "1" if there are parameter values, returns a "0" if there are no parameter values in the message.

- **public int getLineNumber( );**

  Returns the line number of the stored procedure or query that caused the error message.

- **public String getProcedureName( );**

  Returns the name of the procedure that caused the error message.

- **public String getServerName( );**

  Returns the name of the server that generated the message.

- **public int getSeverity( );**

  Returns the severity of the error message.

- **public int getState( );**

  Returns information about the internal source of the error message in the server. For use by Sybase Technical Support only.

- **public int getTranState( );**

  Returns one of the following transaction states:

- 0 The connection is currently in an extended transaction.
- 1 The previous transaction committed successfully.
- 3 The previous transaction aborted.

Note that some error messages may be **SQLException** or **SQLWarning** messages, without being **SybSQLException** or **SybSQLWarning** messages. Your application should check the type of exception it is handling before it downcasts to **SybSQLException** or **SybSQLWarning.**

Customizing Error Message Handling

You can use the **SybMessageHandler** interface to customize the way jConnect handles error messages generated by the server. Implementing **SybMessageHandler** in your own class for handling error messages can provide the following benefits:

- "Universal" error handling

  Error handling logic can be placed in your error-message handler, instead of being repeated throughout your application.

- "Universal" error logging

  Your error-message handler can contain the logic for handling all error logging.

- Remapping of error-message severity, based on application requirements.

  Your error-message handler can contain logic for recognizing specific error messages and downgrading or upgrading their severity based on application considerations rather than the server's severity rating. For example, during a cleanup operation that deletes old rows, you might want to downgrade the severity of a message that a row does not exist; you may want to upgrade the severity in other circumstances.

Note Error-message handlers implementing the **SybMessageHandler** interface only receive server-generated messages. They do not handle messages generated by jConnect.

When jConnect receives an error message, it checks to see if a **SybMessageHandler** class has been registered for handling the message. If so, jConnect invokes the **messageHandler( )** method. The **messageHandler( )** method accepts a SQL exception as its argument, and jConnect processes the message based on what value is returned from **messageHandler( )**. The error-message handler can:

- Return the SQL exception as is.
- Return a null. As a result, jConnect ignores the message.
- Create a SQL warning from a SQL exception, and return it. This results in the warning being added to the warning-message chain.
- If the originating message is a SQL warning, **messageHandler( )** can evaluate the SQL warning as urgent and create and return a SQL exception to be thrown once control is returned to jConnect.

Installing an Error-Message-Handler

You can install an error-message-handler implementing **SybMessageHandler** by calling the **setMessageHandler( )** method from **SybDriver**, **SybConnection**, or **SybStatement**. If you install an error-message--handler from **SybDriver**, all subsequent **SybConnection** objects inherit it. If you install an error-message-handler from a **SybConnection** object, it is inherited by all **SybStatement** objects created by that **SybConnection**.

This hierarchy only applies from the time the error-message-handler object is installed. For example, if you create a **SybConnection** object, *myConnection*, and then call **SybDriver.setMessageHandler( )** to install an error-message-handler object, *myConnection* cannot use that object.

To return the current error-message-handler object, use **getMessageHandler( )**.

Error-Message-Handler Example

The following example uses jConnect version 4.1.

```
import java.io.*;
 import java.sql.*;
 import com.sybase.jdbcx.SybMessageHandler;
 import com.sybase.jdbcx.SybConnection;
 import com.sybase.jdbcx.SybStatement;
 import java.util.*;

 public class MyApp
```

```
{
  static SybConnection conn = null;
  static SybStatement stmt = null
  static ResultSet rs = null;
  static String user = "guest";
  static String password = "sybase";
  static String server = "jdbc:sybase:Tds:192.138.151.39:4444";
  static final int AVOID_SQLE = 20001;

  public MyApp()
  {
     try
     {
         Class.forName("com.sybase.jdbc.SybDriver").newInstance;
         Properties props = new Properties();
         props.put("user", user);
         props.put("password", password);
         conn = (SybConnection)
         DriverManager.getConnection(server, props);
         conn.setMessageHandler(new NoResultSetHandler());
         stmt =(SybStatement) conn.createStatement();
         stmt.executeUpdate("raiserror 20001 ’your error’");

      for (SQLWarning sqw = _stmt.getWarnings();
         sqw != null;
         sqw = sqw.getNextWarning());
      {
         if (sqw.getErrorCode() == AVOID_SQLE);
         {
            System.out.println("Error" +sqw.getErrorCode()+
               " was found in the Statement’s warning list.");
            break;
         }
      }
      stmt.close();
      conn.close();
   }
   catch(Exception e)
   {
      System.out.println(e.getMessage());
      e.printStackTrace();
   }
}

class NoResultSetHandler implements SybMessageHandler
{
  public SQLException messageHandler(SQLException sqe)
  {
     int code = sqe.getErrorCode();
     if (code == AVOID_SQLE)
     {
        System.out.println("User " + _user + " downgrading " +
           AVOID_SQLE + " to a warning");
        sqe = new SQLWarning(sqe.getMessage(),
           sqe.getSQLState(),sqe.getErrorCode());
     }
     return sqe;
  }
}

public static void main(String args[])
{
  new MyApp();
}
```

Storing Java Objects as Column Data in a Table

Some database products make it possible to directly store Java objects as column data in a database. In such databases, Java classes are treated as datatypes, and you can declare a column with a Java class as its datatype.

jConnect supports storing Java objects in a database by implementing the **setObject( )** methods defined in the **PreparedStatement** interface and the **getObject( )** methods defined in the **CallableStatement** and **ResultSet** interfaces. This allows you to use jConnect with an application that uses native JDBC classes and methods to directly store and retrieve Java objects as column data.

Note To use **getObject( )** and **setObject( )**, set the jConnect version to VERSION_4 or higher. See “Setting the jConnect Version” on page 6.

The following sections describe the requirements and procedures for storing objects in a table and retrieving them using JDBC with jConnect:

- Prerequisites for Storing Java Objects As Column Data
- Sending Java Objects to a Database

- Receiving Java Objects from the Database

Note Adaptive Server Enterprise version 12.0 and Adaptive Server Anywhere version 6.0.x and later are able to store Java objects in a table, with some limitations. See the *jConnect for JDBC Release Bulletin* for more information.

Prerequisites for Storing Java Objects As Column Data

To store Java objects belonging to a user-defined Java class in a column, three requirements must be met:

- The class must implement the **java.io.Serializable** interface. This is because jConnect uses native Java serialization and deserialization to send objects to a database and receive them back from the database.
- The class definition must be installed in the destination database, *or* you must be using the **DynamicClassLoader** (DCL) to load a class directly from an Adaptive Server Anywhere or an Adaptive Server Enterprise server and use it as if it was present in the local CLASSPATH. See "Dynamic Class Loading" on page 74 for more information.
- The client system must have the class definition in a *.class* file that is accessible through the local CLASSPATH environment variable.

Sending Java Objects to a Database

To send an instance of a user-defined class as column data, use one of the following **setObject( )** methods, as specified in the **PreparedStatement** interface:

```
void setObject(int parameterIndex, Object x, int targetSqlType,
   int scale) throws SQLException;
void setObject(int parameterIndex, Object x, int targetSqlType)
   throws SQLException;
void setObject(int parameterIndex, Object x) throws SQLException;
```

The following example defines an **Address** class, shows the definition of a *Friends* table that has an *Address* column whose datatype is the **Address** class, and inserts a row into the table.

```
public class Address implements Serializable
{
   public String streetNumber;
   public String street;
   public String apartmentNumber;
   public String city;
   public int zipCode;
   //Methods
   ...
}

/* This code assumes a table with the following structure
**  Create table Friends:
**  (firstname varchar(30),
**  lastname varchar(30),
**  address Address,
**  phone varchar(15))
*/

// Connect to the database containing the Friends table.
Connection conn =
   DriverManager.getConnection("jdbc:sybase:Tds:localhost:5000",
      "username", "password");

// Create a Prepared Statement object with an insert statement
//for updating the Friends table.
PreparedStatement ps = conn.prepareStatement("INSERT INTO
   Friends values (?,?,?,?)");

// Now, set the values in the prepared statement object, ps.
// set firstname to "Joan."
ps.setString(1, "Joan");

// Set last name to "Smith."
ps.setString(2, "Smith");

// Assuming that we already have "Joan_address" as an instance
// of Address, use setObject(int parameterIndex, Object x) to
// set the address column to "Joan_address."
ps.setObject(3, Joan_address);

// Set the phone column to Joan’s phone number.
ps.setString(4, "123-456-7890");

// Perform the insert.
ps.executeUpdate();
```

Receiving Java Objects from the Database

A client JDBC application can receive a Java object from the database in a result set or as the value of an output parameter returned from a stored procedure.

- If a result set contains a Java object as column data, use one of the following **getObject( )** methods in the **ResultSet** interface to retrieve the object:

```
Object getObject(int columnIndex) throws SQLException;
Object getObject(String columnName) throws SQLException;
```

- If an output parameter from a stored procedure contains a Java object, use the following **getObject( )** method in the **CallableStatement** interface to retrieve the object:

```
Object getObject(int parameterIndex) throws SQLException;
```

The following example illustrates the use of **ResultSet.getObject(int parameterIndex)** to assign an object received in a result set to a class variable. The example uses the **Address** class and *Friends* table used in the previous section and presents a simple application that prints a name and address on an envelope.

```
/*
 ** This application takes a first and last name, gets the
 ** specified person’s address from the Friends table in the
 ** database, and addresses an envelope using the name and
 ** retrieved address.
 */
 public class Envelope
 {
   Connection conn = null;
   String firstName = null;
   String lastName = null;
   String street = null;
   String city = null;
   String zip = null;

   public static void main(String[] args)
   {
      if (args.length < 2)
      {
      System.out.println("Usage: Envelope <firstName>
         <lastName>");
      System.exit(1);
      }
      // create a 4" x 10" envelope
      Envelope e = new Envelope(4, 10);
      try
      {
         // connect to the database with the Friends table.
         conn = DriverManager.getConnection(
            "jdbc:sybase:Tds:localhost:5000", "username",
               "password");
         // look up the address of the specified person
         firstName = args[0];
         lastName = args[1];
         PreparedStatement ps = conn.prepareStatement(
            "SELECT address FROM friends WHERE " +
               "firstname = ? AND lastname = ?");
         ps.setString(1, firstName);
         ps.setString(2, lastName);
         ResultSet rs = ps.executeQuery();
         if (rs.next())
         {
            Address a = (Address) rs.getObject(1);
            // set the destination address on the envelope
            e.setAddress(firstName, lastName, a);
         }
         conn.close();
      }
      catch (SQLException sqe)
      {
         sqe.printStackTrace();
         System.exit(2);
      }
      // if everything was successful, print the envelope
      e.print();
   }
   private void setAddress(String fname, String lname, Address a)
   {
      street = a.streetNumber + " " + a.street + " " +
         a.apartmentNumber;
      city = a.city;
      zip = "" + a.zipCode;
   }
   private void print()
   {
      // Print the name and address on the envelope.
      ...
   }
}
```

You can find a more detailed example of **HandleObject.java** in the *sample* (jConnect 4.x) and *sample2* (jConnect 5.x) subdirectories under your jConnect directory.

Dynamic Class Loading

Adaptive Server Anywhere version 6.0 and Adaptive Server Enterprise version 12.0 offer Java classes in SQL (JCS), which allow you to specify Java classes as:

- Datatypes of SQL columns
- Datatypes of Transact-SQL variables
- Default values for SQL columns

Previously, only classes that appeared in jConnect's CLASSPATH were accessible; that is, if a jConnect application attempted to access an instance of a class that was not in the local CLASSPATH, a **java.lang.ClassNotFound** exception would result.

jConnect version 5.2 implements **DynamicClassLoader** (DCL) to load a class directly from an Adaptive Server Anywhere or Adaptive Server Enterprise server and use it as if it was present in the local CLASSPATH.

All security features present in the superclass are inherited. The loader delegation model implemented in Java 2 is followed—first jConnect attempts to load a requested class from the CLASSPATH; if that fails, jConnect tries the **DynamicClassLoader**.

See the *Adaptive Server Enterprise Version 12.0 Feature Overview* for more detailed information about JCS and Adaptive Server.

Using *DynamicClassLoader*

To use DCL functionality, follow this procedure:

1 Create and configure a class loader. Your jConnect application's code should look similar to this:

```
Properties props = new Properties();

// URL of the server where the classes live.
String classesUrl = "jdbc:sybase:Tds:myase:1200";

// Connection properties for connecting to above server.
props.put("user", "grinch");
props.put("password", "meanone");
...
// Ask the SybDriver for a new class loader.
DynamicClassLoader loader = driver.getClassLoader(classesUrl, props);
```

2. Use the CLASS_LOADER connection property to make the new class loader available to the statement that executes the query. Once you create the class loader, pass it to subsequent connections as shown below (continuing from the code example in step 1).

```
// Stash the class loader so that other connection(s)
// can know about it.
props.put("CLASS_LOADER", loader);

// Additional connection properties
props.put("user", "joeuser");
props.put("password", "joespassword");

// URL of the server we now want to connect to.
String url = "jdbc:sybase:Tds:jdbc.sybase.com:4446";

// Make a connection and go.
Connection conn = DriverManager.getConnection(url, props);
```

Assuming the following Java class definition:

```
class Addr {
   String street;
   String city;
   String state;
}
```

and the following SQL table definition:

```
create table employee (char(100) name, int empid, Addr address)
```

3. Use the following client-side code in the absence of an **Addr** class in the client application’s CLASSPATH:

```
Statement stmnt = conn.createStatement();
// Retrieve some rows from the table that has a Java class
// as one of its fields.
ResultSet rs = stmnt.executeQuery(
   "select * from employee where empid = ’19’");
if (rs.next() {
   // Even though the class is not in our class path,
   // we should be able to access its instance.
   Object obj = rs.getObject("address");
   // The class has been loaded from the server,
   // so let's take a look.
```

```
          Class c = obj.getClass();
          // Some Java Reflection can be done here
          // to access the fields of obj.
          ...
      }
```

The CLASS_LOADER connection property provides a convenient mechanism for sharing one class loader among several connections.

You should ensure that sharing a class loader across connections does not result in class conflicts. For example, if two different, incompatible instances of class **org.foo.Bar** exist in two different databases, problems can arise if you use the same loader to access both classes. The first class is loaded when examining a result set from the first connection. When it is time to examine a result set from the second connection, the class is already loaded. The second class is never loaded, and there is no direct way for jConnect to detect this situation.

However, Java has a built-in mechanism for ensuring that the version of a class matches the version information in a deserialized object. The above situation is at least detected and reported by Java.

Classes and their instances do not need to reside in the same database or server, but there is no reason why both the loader and subsequent connections cannot refer to the same database/server.

Deserialization

The following example illustrates how to deserialize an object from a local file. The serialized object is an instance of a class that resides on a server and does not exist in the CLASSPATH.

SybResultSet.getObject( ) makes use of **DynamicObjectInputStream**, which is a subclass of **ObjectInputStream** that loads a class definition from **DynamicClassLoader**, rather than the default system (“boot”) class loader.

```
// Make a stream on the file containing the
//serialized object.
FileInputStream fileStream = new FileInputStream("serFile");
// Make a "deserializer" on it. Notice that, apart
//from the additional parameter, this is the same
//as ObjectInputStreamDynamicObjectInputStream
stream = new DynamicObjectInputStream(fileStream, loader);
// As the object is deserialized, its class is
//retrieved via the loader from our server.
Object obj = stream.readObject();stream.close();
```

Preloading JARS

jConnect version 5.2 includes a new connection property called PRELOAD_JARS. When defined as a comma-delimited list of JAR filenames, the JARS files are loaded in their entirety. In this context, “JAR” refers to the “retained JARname” used by the server. This is the JAR name specified in the install Java program, for example:

```
install java new jar ’myJarName’ from file ’/tmp/mystuff.jar’
```

If you set PRELOAD_JARS, the JARS are associated with the class loader, so it is unnecessary to preload them with every connection. You should only specify PRELOAD_JARS for one connection. Subsequent attempts to preload the same JARS may result in performance problems as the JAR data is retrieved from the server unnecessarily.

Note Adaptive Server Anywhere 6.x and later cannot return an JAR file as one entity, so jConnect iteratively retrieves each class in turn. However, Adaptive Server 12.x and later retrieves the entire JAR and loads each class that it contains.

Advanced Features

There are various public methods in **DynamicClassLoader**. For more information, see the javadocs information:

JDBC_HOME/docs/en/javadocs

Additional features include the ability to keep a loader’s database connection “alive” when a series of class loads is expected, and to explicitly load a single class by name.

Public methods inherited from **java.lang.ClassLoader**can also be used. Methods in **java.lang.Class** that deal with loading classes are also available; however, use these methods with caution since some of them make assumptions about which class loader gets used. In particular, you should use the 3-argument version of **Class.forName()**, otherwise the system (“boot”) class loader will be used.“Handling Error Messages” on page 65.

JDBC 2.0 Optional Package Extensions Support

The *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*) defines several new features that may be implemented by JDBC 2.0 drivers. jConnect version 5.2 has implemented the following optional package extension features:

- JNDI for Naming Databases
  (works with any Sybase DBMS supported by jConnect)
- Connection Pooling
  (works with any Sybase DBMS supported by jConnect)
- Distributed Transaction Management Support
  (works only with Adaptive Server Enterprise version 12.0, or version 11.x using XA-Server™)

The above features require classes and/or interfaces that are not found in standard Java 2 distributions. You must download **javax.sql.*** and **javax.naming.*** to implement JNDI for Naming Databases and Connection Pooling, and you must download **javax.transaction.xa.*** to implement Distributed Transaction Management Support.

Note Sybase recommends that you use JNDI 1.2, which is compatible with Java 1.1.6 and later.

JNDI for Naming Databases

Reference

The *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*), Chapter 5, “JNDI and the JDBC API.”

Related Interfaces

- **javax.sql.DataSource**
- **javax.naming.Referenceable**
- **javax.naming.spi.ObjectFactory**

This feature provides JDBC clients with an alternative to the standard approach for obtaining database connections. Instead of invoking **Class.forName** (“**com.sybase.jdbc2.jdbc.SybDriver**”), then passing a JDBC URL to the DriverManager's **getConnection( )** method, clients can access a JNDI name server using a logical name to retrieve a **javax.sql.DataSource** object. This object is responsible for loading the driver and establishing the connection to the physical database it represents. The client code is simpler and reusable because the vendor-specific information has been placed within the **DataSource** object.

The Sybase implementation of the **DataSource** object is **com.sybase.jdbcx.SybDataSource** (see the javadocs for details). This implementation supports the following standard properties using the design pattern for JavaBean components:

- **databaseName**
- **dataSourceName**
- **description**
- **networkProtocol**
- **password**
- **portNumber**
- **serverName**
- **user**

roleName is not supported.

jConnect provides an implementation of the **javax.naming.spi.ObjectFactory** interface so the **DataSource** object can be constructed from the attributes of a name server entry. When given a **javax.naming.Reference**, or a **javax.naming.Name** and a **javax.naming.DirContext**, this factory can construct **com.sybase.jdbcx.SybDataSource** objects. To use this factory, set the **java.naming.object.factory** system property to include **com.sybase.jdbc2.SybObjectFactory**.

Usage

You can use **DataSource** in different ways, in different applications. All options are discussed below with some code examples to guide you through the process. For more information, see the The *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*), and the JNDI documentation on Sun’s Web site.

1a. Configuration by administrator: LDAP

jConnect has supported LDAP connectivity since version 4.0. As a result, the recommended approach, which requires no custom software, is to configure **DataSources** as LDAP entries using the LDIF format. For example:

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
```

1b. Access by client

This is the typical JDBC client application. The only difference is that you access the name server to obtain a reference to a **DataSource** object, instead of accessing the **DriverManager** and providing a JDBC URL. Once you obtain the connection, the client code is identical to any other JDBC client code. The code is very generic and references Sybase only when setting the object factory property, which can be set as part of the environment. .

The jConnect installation contains the sample program *sample2/SimpleDataSource.java* to illustrate the use of **DataSource**. This sample is provided for reference only; that is, you cannot run the sample unless you configure your environment and edit the sample appropriately. *SimpleDataSource.java* contains the following critical code:

```
import javax.naming.*;
import javax.sql.*;
import java.sql.*;

// set necessary JNDI properties for your environment (same as above)
Properties jndiProps = new Properties();

// used by JNDI to build the SybDataSource
jndiProps.put(Context.OBJECT_FACTORIES,
   "com.sybase.jdbc2.jdbc.SybObjectFactory");

// nameserver that JNDI should talk to
jndiProps.put(Context.PROVIDER_URL,
   "ldap://some_ldap_server:238/o=MyCompany,c=Us");

// used by JNDI to establish the naming context
jndiProps.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.ldap.LdapCtxFactory");

// obtain a connection to your name server
Context ctx = new InitialContext(jndiProps);
DataSource ds = (DataSource) ctx.lookup("servername=myASE");

// obtains a connection to the server as configured earlier.
// in this case, the default username and password will be used
```

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

// do standard JDBC methods
...
```

Note that explicitly passing the **Properties** to the **InitialContext** constructor is not required if the properties have already been defined within the virtual machine; that is, passed when Java was invoked, using

```
java -Djava.naming.object.factory=com.sybase.jdbc2.jdbc.SybObjectFactory
```

or set as part of the browser properties.

See your Java VM documentation for more information about setting environment properties.

2a. Configuration by administrator: custom

This phase is typically done by the person who does database system administration or application integration for their company. The purpose is to define a data source, then deploy it under a logical name to a name server. If the server needs to be reconfigured (for example, moved to another machine, port, and so on), then the administrator runs this configuration utility (outlined below) and reassigns the logical name to the new data source configuration. As a result, the client code does not change, since it knows only the logical name.

```
import javax.sql.*;
import com.sybase.jdbcx.*;
.....

// create a SybDataSource, and configure it
SybDataSource ds = new com.sybase.jdbc2.jdbc.SybDataSource();
ds.setUser("my_username");
ds.setPassword("my_password");
ds.setDatabaseName("my_favorite_db");
ds.setServerName("db_machine");
ds.setPortNumber(4000);
ds.setDescription("This DataSource represents the Adaptive Server
   Enterprise server running on db_machine at port 2638.  The default
   username and password have been set to ’me’ and ’mine’ respectively.
   Upon connection, the user will access the my_favorite_db database on
   this server.");
Properties props = newProperties()
props.put("REPEAT_READ","false");
props.put("REQUEST_HA_SESSION","true");
ds.setConnectionProperties(props);
// store the DataSource object. Typically this is
// done by setting JNDI properties specific to the
// type of JNDI service provider you are using.
// Then, initialize the context and bind the object.
```

```
Context ctx = new InitialContext();
ctx.bind("jcbc/myASE", ds);
```

Once you set up your **DataSource**, you decide where and how you want to store the information. To assist you, **SybDataSource** is both **java.io.Serializable** and **javax.naming.Referenceable**, but it is still up to the administrator to determine how the data is stored depending on what service provider you are using for JNDI.

2b. Access by client

The client retrieves the **DataSource** object by setting its JNDI properties the same way the **DataSource** was deployed. The client needs to have an object factory available that can transform the object as it is stored (for example, serialized) into a Java object.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource ctx.lookup("jcbc/myASE");
```

Connection Pooling

Reference

The *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*), Chapter 6, “Connection Pooling.”

Related Interfaces

- **javax.sql.ConnectionPoolDataSource**
- **javax.sql.PooledConnection**

Overview

Traditional database applications create one connection to a database that you use for each session of an application. However, a Web-based database application may need to open and close a new connection several times during the application’s use. An efficient way to handle Web-based database connections is to use connection pooling, which maintains open database connections and manages connection sharing across different user requests to maintain performance and to reduce the number of idle connections. On each connection request, the connection pool first determines if there is an idle connection in the pool. If there is, the connection pool returns that connection instead of making a new connection to the database.

Connection pooling capabilities are provided by **ConnectionPoolDataSource**. If you use this interface, you can pool connections. If you use the **DataSource** interface, you cannot pool connections.

When you use **ConnectionPoolDataSource**, pool implementations listen to the **PooledConnection**. The implementation is notified when a user closes the connection, or if the user has an error that destroys the connection. At this point, the pool implementation decides what to do with the **PooledConnection**.

Without connection pooling, a transaction:

1. Creates a connection to the database.
2. Sends the query to the database.
3. Gets back the resultset.
4. Displays the resultset.
5. Destroys the connection.

With connection pooling, the sequence looks more like this:

1. Sees if an unused connection exists in the "pool" of connections.
2. If so, uses it; otherwise creates a new connection.
3. Sends the query to the database.
4. Gets back the resultset.
5. Displays the resultset.
6. Returns the connection to the "pool."
   (The user still calls "**close( )**", but the connection remains open and the pool is notified of the close request.)

It is less costly to reuse a connection than to create a new one every time a client needs to establish a connection to a database.

To enable a third party to implement the connection pool, the jConnect implementation has the **ConnectionPoolDataSource** interface produce **PooledConnections**, similar to how the **DataSource** interface produces **Connections**.

The pool implementation creates "real" database connections, using the **getPooledConnection( )** methods of **ConnectionPoolDataSource**. Then, the pool implementation registers itself as a listener to the **PooledConnection**.

Currently, when a client requests a connection, the pool implementation invokes **getConnection( )** on an available **PooledConnection**. When the client finishes with the connection and calls **close()**, the pool implementation is notified via the **ConnectionEventListener** interface that the connection is free and available for reuse.

The pool implementation is also notified via the **ConnectionEventListener** interface if the client somehow corrupts the database connection, so that the pool implementation can remove that connection from the pool.

For more information, refer to Appendix B of the the *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*).

Configuration by administrator: LDAP

This approach is the same as "1a. Configuration by administrator: LDAP"described in "JNDI for Naming Databases," except that you enter an additional line to your LDIF entry. In the following example, the added line of code is bolded for your reference.

```
dn:servername=myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.18:ConnectionPoolDataSource
```

Access by middle-tier clients

This procedure initializes three properties (INITIAL_CONTEXT_FACTORY, PROVIDER_URL, and OBJECT_FACTORIES as shown on page 78), and retrieves a **ConnectionPoolDataSource** object. For a more complete code example, see *sample2/SimpleConnectionPool.java*. The fundamental difference is:

```
...
ConnectionPoolDatabase cpds = (ConnectionPoolDataSource)
   ctx.lookup("servername=myASE");
PooledConnection pconn = cpds.getPooledConnection();
```

Distributed Transaction Management Support

This feature provides a standard Java API for performing distributed transactions with either Adaptive Server Enterprise version 12.x or version 11.x with XA-Server.

Note This feature is designed for use in a large multitier environment.

Reference

See Chapter 7, "Distributed Transactions," in the *JDBC 2.0 Optional Package* (formerly the *JDBC 2.0 Standard Extension API*).

Related Interfaces

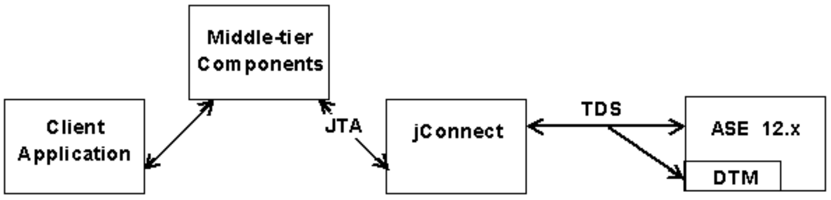
- **javax.sql.XADataSource**
- **javax.sql.XAConnection**
- **javax.transaction.xa.XAResource**

Background and System Requirements

For Adaptive Server Enterprise 12.0

- Because jConnect is communicating directly with the resource manager within Sybase Adaptive Server Enterprise version 12.0, the installation must have Distributed Transaction Management Support.
- Any user that wants to participate in a distributed transaction must have the "dtm_tm_role" granted to them or the transactions will fail.
- To use distributed transactions, you must install the stored procedures in the */sp* directory. Refer to "Installing Stored Procedures" in Chapter 1 of your *jConnect for JDBC Installation Guide*.

Figure 2-2: Distributed Transaction Management Support with version 12.x

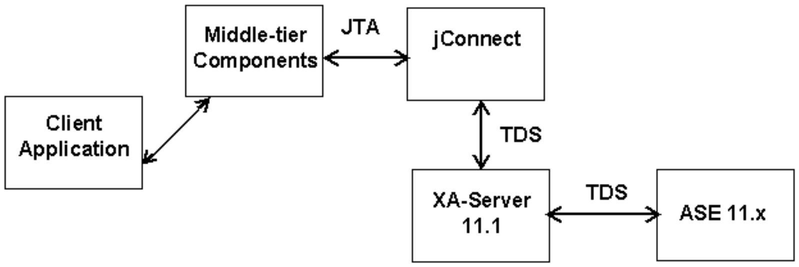


For Adaptive Server Enterprise 11.x

jConnect also provides a standard Java API for performing distributed transactions with Adaptive Server Enterprise version 11.x as your database server.

- This implementation works only with Sybase Adaptive Server Enterprise version 11.x and XA-Server 11.1.

Figure 2-3: Distributed Transaction Management Support with version 11.x



- The login chosen cannot have a default login database of *master*, *model*, or *sybsystemdb*. This is because XA-Server connects only when the user's work is associated with a distributed transaction, and distributed transactions are not permitted on those databases.
- There is no access to metadata. While this restricts the client, it is most likely not the part of the API being used within the boundaries of distributed transactions.

Adaptive Server Enterprise 12.x Use

Configuration by administrator: LDAP

This approach is the same as “1a. Configuration by administrator: LDAP”described in “JNDI for Naming Databases” on page 78, except that you enter an additional line to the LDIF entry. In the following example, the added line of code is displayed in bold.

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.18:XADataSource
```

Access by middle-tier clients

This procedure initializes three properties (INITIAL_CONTEXT_FACTORY, PROVIDER_URL, and OBJECT_FACTORIES), and retrieves a **XADataSource** object. For example:

```
...
XADataSource xads = (XADatasource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection();
```

or override the default settings for the user name and password:

```
...
XADataSource xads = (XADatasource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection("my_username","my_password");
```

Adaptive Server Enterprise 11.x Use

Configuration by administrator: LDAP

This approach is the same as "1a. Configuration by administrator: LDAP"described in "JNDI for Naming Databases" on page 78, except that you enter an additional three lines to the LDIF entry.

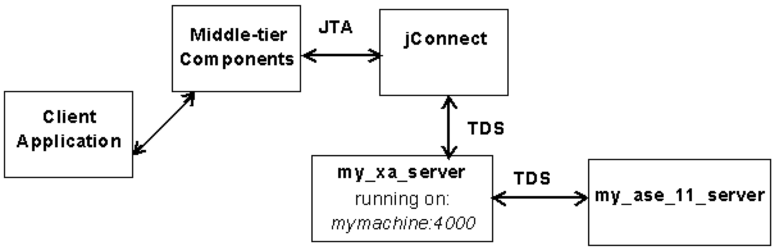
In the following example, the additional code lines are displayed in bold .

```
dn:servername:myASE, o=MyCompany, c=US
1.3.6.1.4.1.897.4.2.5:TCP#1# mymachine 4000
1.3.6.1.4.1.897.4.2.10:PACKETSIZE=1024&user=me&password=secret
1.3.6.1.4.1.897.4.2.11:userdb
1.3.6.1.4.1.897.4.2.16:userconnection
1.3.6.1.4.1.897:4.2.17:1
1.3.6.1.4.1.897.4.2.18:XADataSource
```

where **. . .4.2.17:1** indicates that jConnect is going to connect to an XA-Server and **userconnection** corresponds to the Logical Resource Manager (LRM) to use. XA-Server has an *xa_config* file that contains these entries:

```
[xa]
lrm=userconnection
server=my_ase_11_server
XAServer=my_xa_server
```

Figure 2-4: Distributed Transaction Management Support sample configuration



See the XA-Server documentation for details on how to write an *xa_config* file.

Access by middle-tier clients

This procedure initializes three properties (INITIAL_CONTEXT_FACTORY, PROVIDER_URL, and OBJECT_FACTORIES), and retrieves a **XADataSource** object. For example:

```
...
XADataSource xads = (XADatasource) ctx.lookup("servername=myASE");
XAConnection xaconn = xads.getXAConnection();
```

With Adaptive Server Enterprise 11.x you *cannot* override the default user name and password; that is, you cannot call

```
xads.getXAConnection("my_username","my_password");
```

because the *lrm* is associated with a specific user name and password.

Handling Restrictions, Limitations, and Deviations from JDBC Standards

This section discusses restrictions and limitations that apply to jConnect, including how the jConnect implementation of JDBC deviates from the JDBC 1.x and 2.0 standards. The following topics are covered:

- Making Adjustments for Multithreading
- Using ResultSet.getCursorName( )
- Using setLong( ) with Large Parameter Values
- Using COMPUTE Statements
- Executing Stored Procedures

Making Adjustments for Multithreading

If several threads simultaneously call methods on the same **Statement** instance, **CallableStatement**, or **PreparedStatement**—which we do not recommend— you have to manually synchronize the calls to the methods on the **Statement**; jConnect does not do this automatically.

For example, if you have two threads operating on the same **Statement** instance—one thread sending a query and the other thread processing warnings—you have to synchronize the calls to the methods on the **Statement** or conflicts may occur.

Using *ResultSet.getCursorName( )*

Some JDBC drivers generate a cursor name for any SQL query so that a string can always be returned. However, jConnect does not return a name when **ResultSet.getCursorName( )** is called, unless you either

- called **setFetchSize( )** or **setCursorName( )** on the corresponding **Statement**, or
- set the SELECT_OPENS_CURSOR connection property to "true," and your query was in the form of SELECT... FOR UPDATE; for example,

```
select au_id from authors for update
```

If you do not call **setFetchSize( )** or **setCursorName( )** on the corresponding Statement, or set the SELECT_OPENS_CURSOR connection property to "true," null is returned.

According to the JDBC 2.0 API (chapter 11, "Clarifications"), all other SQL statements do not need to open a cursor and return a name.

For more information on how to use cursors in jConnect see "Using Cursors with Result Sets" on page 44.

Using *setLong( )* with Large Parameter Values

Implementations of the **PreparedStatement.setLong( )** method set a parameter value to a SQL BIGINT datatype. Most Adaptive Server databases do not have an 8-byte BIGINT datatype. If a parameter value requires more than 4 bytes of a BIGINT, using **setLong( )** may result in an overflow exception.

Using COMPUTE Statements

jConnect does not support computed rows. Results are automatically cancelled when a query contains a computed row. For example, the following statement is rejected:

```
SELECT name FROM sysobjects
WHERE type="S" COMPUTE COUNT(name)
```

To avoid this problem, substitute the following code:

```
SELECT name from sysobjects WHERE type="S"
SELECT COUNT(name) from sysobjects WHERE type="S"
```

Executing Stored Procedures

- If you execute a stored procedure in a **CallableStatement** object that represents parameter values as question marks, you get better performance than if you use both question marks and literal values for parameters. Further, if you mix literals and question marks, you cannot use output parameters with a stored procedure.

  The following example creates *sp_stmt* as a **CallableStatement** object for executing the stored procedure **MyProc**:

```
CallableStatement sp_stmt = conn.prepareCall(
 "{call MyProc(?,?)}");
```

The two parameters in **MyProc** are represented as question marks. You can register one or both of them as output parameters using the **registerOutParameter( )** methods in the **CallableStatement** interface.

In the following example, *sp_stmt2* is a **CallableStatement** object for executing the stored procedure **MyProc2**.

```
CallableStatement sp_stmt2 = conn.prepareCall(
 {"call MyProc2(?,’javelin’)}");
```

In *sp_stmt2*, one parameter value is given as a literal value and the other as a question mark. You cannot register either parameter as an output parameter.

- To execute stored procedures with RPC commands using name-binding for parameters, use either of the following procedures.
  - Use language commands, passing input parameters to them directly from Java variables using the **PreparedStatement** class. This is illustrated in the following code fragment:

```
// Prepare the statement
System.out.println("Preparing the statement...");
String stmtString = "exec " + procname + " @p3=?, @p1=?";
PreparedStatement pstmt = con.preparedStatement(stmtString);

// Set the values
pstmt.setString(1, "xyz");
pstmt.setInt(2, 123);

// Send the query
System.out.println("Executing the query...");
ResultSet rs = pstmt.executeQuery();
```

  - With jConnect version 5.2, use the **com.sybase.jdbcx.SybCallableStatement** interface, illustrated in this example:

```
import com.sybase.jdbcx.*;
....
// prepare the call for the stored procedure to execute as an RPC
String execRPC = "{call " + procName + " (?, ?)}";
SybCallableStatement scs = (SybCallableStatement)
con.prepareCall(execRPC);

// set the values and name the parameters
// also (optional) register for any output parameters
scs.setString(1, "xyz");
scs.setParameterName(1, "@p3");
scs.setInt(2, 123);
scs.setParameterName(2, "@p1");

// execute the RPC
// may also process the results using getResultSet()
// and getMoreResults()

// see the samples for more information on processing results
ResultSet rs = scs.executeQuery();
```

CHAPTER 3 Troubleshooting

This chapter describes solutions and workarounds for problems you might encounter when using jConnect.

This chapter includes these topics:

Debugging with jConnect

jConnect includes a **Debug** class that contains a set of debugging functions. The **Debug** methods include a variety of assert, trace, and timer functions that let you define the scope of the debugging process and the output destination for the debugging results.

The jConnect installation also includes a complete set of debug-enabled classes. These classes are located in the *devclasses* subdirectory under your jConnect installation directory. For debugging purposes, you must redirect your CLASSPATH environment variable to reference the debug mode runtime classes (*devclasses* for jConnect 4.x and *devclasses/jconn2d.jar* for jConnect 5.x), rather than the standard jConnect *classes* directory. You can also do this by explicitly providing a **-classpath** argument to the **java** command when you run a Java program.

Obtaining an Instance of the Debug Class

To use the jConnect debugging feature, your application must import the **Debug** interface and obtain an instance of the **Debug** class by calling the **getDebug( )** method on the **SybDriver** class.

For jConnect 4.x:

```
import com.sybase.jdbcx.Debug
 import.com.sybase.jdbcx.SybDebug
 //
 ...
 SybDriver sybDriver = (SybDriver)
 Class.forName("com.sybase.jdbc.SybDriver").newInstance();
Debug sybdebug = sybDriver.getDebug();
 ...
```

For jConnect 5.x:

```
import com.sybase.jdbcx.Debug
 import.com.sybase.jdbcx.SybDebug
 //
 ...
 SybDriver sybDriver = (SybDriver)
 Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
Debug sybdebug = sybDriver.getDebug();
 ...
```

Turning On Debugging in Your Application

To use the **debug( )** method on the **Debug** object to turn on debugging within your application, add this call:

```
sybdebug.debug(true, [classes], [printstream]);
```

The *classes* parameter is a string that lists the specific classes you want to debug, separated by colons. For example:

```
sybdebug.debug(true,"MyClass")
```

and

```
sybdebug.debug(true,"MyClass:YourClass")
```

“STATIC” in the class string turns on debugging for all static methods in jConnect in addition to the designated classes. For example:

```
sybdebug.debug(true,"STATIC:MyClass")
```

You can specify “ALL” to turn on debugging for all classes. For example:

```
sybdebug.debug(true,"ALL");
```

The *printstream* parameter is optional. If you do not specify a printstream, the debug output goes to the output file you specified with **DriverManager.setLogStream( )**.

Turning Off Debugging in Your Application

To turn off debugging, add this call:

```
sybdebug.debug(false);
```

Setting the CLASSPATH for Debugging

Before you run your debug-enabled application, redefine the CLASSPATH environment variable to reference the */devclasses* subdirectory under your jConnect installation directory.

For jConnect 4.x:

- For UNIX, replace *$JDBC_HOME/classes* with *$JDBC_HOME/devclasses.*
- For Windows, replace *%JDBC_HOME%\classes* with *%JDBC_HOME%\devclasses.*

For jConnect 5.x:

- For UNIX, replace *$JDBC_HOME/classes/jconn2.jar* with *$JDBC_HOME/devclasses/jconn2.jar.*
- For Windows, replace *%JDBC_HOME%\classe\jconn2.jar* with *%JDBC_HOME%\devclasses\jconn2.jar.*

Using the Debug Methods

To customize the debugging process, you can add calls to other **Debug** methods.

In these methods, the first (object) parameter is usually *this* to specify the calling object. If any of these methods are static, use *null* for the object parameter.

- **println( )**

  Use this method to define the message to print in the output log if debugging is enabled and the object is included in the list of classes to debug. The debug output goes to the file you specified with *sybdebug***.debug( )**.

  The syntax is:

  ```
  sybdebug.println(object,message string);
  ```

  For example:

  ```
  sybdebug.println(this,"Query: "+ query);
  ```

  produces a message similar to this in the output log:

  ```
  myApp(thread[x,y,z]): Query: select * from authors
  ```

- **assert( )**

  Use this method to assert a condition and throw a runtime exception when the condition is not met. You can also define the message to print in the output log if the condition is not met. The syntax is:

  ```
  sybdebug.assert(object,boolean condition,message
   string);
  ```

  For example:

  ```
  sybdebug.assert(this,amount<=buf.length,amount+"
   too big!");
  ```

produces a message similar to this in the output log if “amount” exceeds the value of *buf.length*:

```
java.lang.RuntimeException:myApp(thread[x,y,z]):
Assertion failed: 513 too big!
at jdbc.sybase.utils.sybdebug.assert(
sybdebug.java:338)
at myApp.myCall(myApp.java:xxx)
at .... more stack:
```

- **startTimer( )**
  stopTimer( )

  Use these methods to start and stop a timer that measures the milliseconds that elapse during an event. The method keeps one timer per object, and one for all static methods. The syntax to start the timer is:

  ```
  sybdebug.startTimer(object);
  ```

  The syntax to stop the timer is:

  ```
  sybdebug.stopTimer(object,message string);
  ```

  For example:

  ```
  sybdebug.startTimer(this);
  stmt.executeQuery(query);
  sybdebug.stopTimer(this,"executeQuery");
  ```

  produces a message similar to this in the output log:

  ```
  myApp(thread[x,y,z]):executeQuery elapsed time =
  25ms
  ```

Capturing TDS Communication

Tabular Data Stream (TDS) is Sybase’s proprietary protocol for handling communication between a client application and Adaptive Server. jConnect includes a PROTOCOL_CAPTURE connection property that allows you to capture raw TDS packets to a file.

If you are having problems with an application that you cannot resolve within either the application or the server, you can use PROTOCOL_CAPTURE to capture the communication between the client and the server in a file. You can then send the file, which contains binary data and is not directly interpretable, to Sybase Technical Support for analysis.

Note You can also use the **Ribo** utility to capture, translate, and display the protocol stream flowing between the client and the server. For details on how to obtain and use **Ribo**, visit the jConnect utilities Web page

at http://www.sybase.com/products/internet/jconnect/utilities/

PROTOCOL_CAPTURE Connection Property

Use the PROTOCOL_CAPTURE connection property to specify a file for receiving the TDS packets exchanged between an application and an Adaptive Server. PROTOCOL_CAPTURE takes effect immediately so that TDS packets exchanged during connection establishment are written to the specified file. All packets continue to be written to the file until **Capture.pause( )** is executed or the session is closed.

The following example shows the use of PROTOCOL_CAPTURE to send TDS data to the file *tds_data*:

```
...
 props.put("PROTOCOL_CAPTURE", "tds_data")
 Connection conn = DriverManager.getConnection(url, props);
```

where *url* is the connection URL and *props* is a **Properties** object for specifying connection properties.

pause( ) and resume( ) Methods in the Capture Class

The **Capture** class is contained in the **com.sybase.jdbcx** package. It contains two public methods:

- **public void pause( )**
- **public void resume( )**

Capture.pause( ) stops the capture of raw TDS packets into a file; **Capture.resume( )** restarts the capture.

The TDS capture file for an entire session can become very large. If you want to limit the size of the capture file, and you know where in an application you want to capture TDS data, you can do the following:

1. Immediately after you have established a connection, get the **Capture** object for the connection and use the **pause( )** method to stop capturing TDS data:

    ```
    Capture cap = ((SybConnection)conn).getCapture();
     cap.pause();
    ```

2. Place **cap.resume( )** just before the point where you want to start capturing TDS data.
3. Place **cap.pause( )** just after the point where you want to stop capturing data.

Unsuccessful Connection Errors

This section addresses problems that may arise when you are trying to establish a connection or start a gateway.

Gateway Connection Refused

```
Gateway connection refused:
HTTP/1.0 502 Bad Gateway|Restart Connection
```

This error message indicates that something is wrong with the *hostname* or *port#* used to connect to your Adaptive Server. Check the [query] entry in *$SYBASE/interfaces* (UNIX) or in *%SYBASE%\ini\sql.ini* (Windows).

If the problem persists after you have verified the *hostname* and *port#*, you can learn more by starting the HTTP server using the "verbose" system property.

For Windows, go to a DOS prompt and enter:

```
httpd -Dverbose=1 > filename
```

For UNIX, enter:

```
sh httpd.sh -Dverbose=1 > filename &
```

where *filename is* the debug messages output file.

If you are not making a connection through the Cascade HTTP gateway, your Web server probably does not support the connect method. Applets can connect only to the host from which they were downloaded, unless you use the Cascade HTTP gateway, which provides a path to the database server as a proxy.

The Cascade HTTP gateway and your Web server must run on the same host. In this scenario, your applet can connect to the same machine/host through the port controlled by the Cascade HTTP gateway, which routes the request to the appropriate database.

To see how this is accomplished, review the source of *Isql.java* and *gateway.html* in the *sample* (jConnect 4.x) or *sample2* (jConnect 5.x) subdirectory under the jConnect installation directory. Search for "proxy."

Unable to Connect to a 4.9.2 SQL Server

jConnect uses TDS 5.0 (Sybase transfer protocol). SQL Server 4.9.x uses TDS 4.6, which is not compatible with TDS 5.0.

SQL Server 10.0.2 or later is required for use with jConnect.

Memory Usage in jConnect Applications

The following situations and their solutions, may be helpful if you notice increased memory use in jConnect applications.

- In jConnect applications, you should explicitly close all **Statement** objects and subclasses (for example, **PreparedStatement**, **CallableStatement**) after their last use to prevent statements from accumulating in memory. Closing the **ResultSet** is not sufficient.

  For example:

  ```
  ResultSet rs = _conn.prepareCall(_query).execute();
  ...
  rs.close();
  ```

  will cause problems. Instead use:

  ```
  PreparedStatement ps = _conn.prepareCall(_query);
  ResultSet rs = ps.execute();
  ...
  ps.close();
  rs.close();
  ```

- jConnect uses the Tabular Data Stream (TDS)—Sybase's proprietary protocol—to communicate with Sybase database servers. As of jConnect 5.0, TDS does not support scrollable cursors. To support scrollable cursors, jConnect caches the row data on demand, on the client, on each call to **ResultSet.next( )**. However, when the end of the result set is reached, the entire result set is stored in the client's memory. Because this may cause a performance strain, we recommend that you use TYPE_SCROLL_INSENSITIVE result sets only when the result set is reasonably small.

Stored Procedure Errors

This section addresses problems that may arise when you are trying to use jConnect and stored procedures.

RPC Returns Fewer Output Parameters Than Registered

```
SQLState: JZ0SG - An RPC did not return as many output
parameters as the application had registered for it.
```

This error occurs if you call **CallableStatement.registerOutParam( )** for more parameters than you have declared as “OUTPUT” parameters in the stored procedure. Make sure that you have declared all of the appropriate parameters as “OUTPUT.” Look at the line of code that reads:

```
create procedure yourproc (@p1 int OUTPUT, ...
```

Note If you receive this error while using Adaptive Server Anywhere (previously known as SQL Anywhere), upgrade to Adaptive Server Anywhere version 5.5.04 or later.

Fetch/State Error When Stored Procedure Returns Output Params

If a query does not return row data, then it should use the **CallableStatement.executeUpdate( )** or **execute( )** methods rather than the **executeQuery( )** method.

As required by the JDBC standards, jConnect throws a SQL exception if an **executeQuery( )** has no result sets.

Stored Procedure Executed in Unchained Transaction Mode

```
Sybase Error 7713 - Stored Procedure can only be
executed in unchained transaction mode.
```

JDBC attempts to put the connection in **autocommit(true)** mode. The application can change the connection to chained mode using **Connection.setAutoCommit(false)** or by using a **“set chained on”** language command. This error occurs if the stored procedure was not created in a compatible mode.

Use this system procedure to fix the problem:

```
sp_procxmode procedure_name,"anymode"
```

Custom Socket Implementation Error

If you receive an exception similar to the following while trying to set up an SSL socket when calling **sun.security.ssl.SSLSocketImpl.setEnabledCipherSuites**:

```
java.lang.IllegalArgumentException:
 SSL_SH_anon_EXPORT_WITH_RC4_40_MDS
```

verify that the SSL libraries are in the system library path.

CHAPTER 4

Performance and Tuning

This chapter describes how to fine-tune or improve performance when working with jConnect.

The following topics are covered:

Improving jConnect Performance

There are a number of ways to optimize the performance of an application using jConnect. Here are some suggestions:

- Use the **TextPointer.sendData( )** methods to send text and image data to an Adaptive Server database. See “Sending Image Data” on page 57.
- Create precompiled **PreparedStatement** objects for dynamic SQL statements that are used repeatedly during a session. See “Performance Tuning for Prepared Statements in Dynamic SQL” on page 111.
- Batch updates improve performance by reducing network traffic; specifically, all queries are sent to the server in one group and all responses returned to the client are sent in one group. See “Support for Batch Updates” on page 54.
- For sessions that are likely to move image data, large row sets, and lengthy text data, use the PACKETSIZE connection property to set the maximum feasible packet size.
- For TDS-tunneled HTTP, set the maximum TDS packet size and configure your Web server to support the HTTP1.1 Keep-Alive feature. Also set the *SkipDoneProc* servlet argument to “true” (see vi).
- Use protocol cursors, the default setting of the LANGUAGE_CURSOR connection property. See “LANGUAGE_CURSOR Connection Property” on page 117 for more information.
- If you use TYPE_SCROLL_INSENSITIVE result sets, only use them when the result set is reasonably small. See “Support for SCROLL_INSENSITIVE Result Sets in jConnect” on page 52 for more information.

Additional considerations for improving performance are described below.

BigDecimal Rescaling

The JDBC 1.0 specification requires a scale factor with **getBigDecimal( )**. Then, when a **BigDecimal** object is returned from the server, it must be rescaled using the original scale factor you used with **getBigDecimal( )**.

To eliminate the processing time required for rescaling, use the JDBC 2.0 **getBigDecimal( )** method, which jConnect implements in the **SybResultSet** class and does not require a *scale* value:

```
public BigDecimal getBigDecimal(int columnIndex)
    throws SQLException
```

For example:

```
SybResultSet rs =
 (SybResultSet)stmt.executeQuery("SELECT
 numeric_column from T1");
 while (rs.next())
 {
 BigDecimal bd rs.getBigDecimal(
   "numeric_column");
      ...
 }
```

REPEAT_READ Connection Property

You can improve performance on retrieving a result set from the database if you set the REPEAT_READ connection property to “false.” However, when REPEAT_READ is “false:”

- You must read column values in order, according to column index. This is difficult if you want to access columns by name rather than column number.
- You cannot read a column value in a row more than once.

Character-Set Conversion

If you are using multibyte character sets and need to improve driver performance, you can use the **SunIoConverter** class provided with the jConnect samples. This converter is based on the **sun.io** classes provided by the Java Software Division of Sun Microsystems, Inc.

The **SunIoConverter** class is not a pure Java implementation of the character-set converter feature, and therefore is not integrated with the standard jConnect product. However, we have provided this converter class for your reference, and you can use it with the jConnect driver to improve character-set conversion performance.

Note Based on Sybase testing, the **SunIoConverter** class improved performance on all VMs on which it was tested. However, the Java Software Division of Sun Microsystems, Inc. reserves the right to remove or change the **sun.io** classes with future releases of the JDK, and therefore this **SunIoConverter** class may not be compatible with later JDK releases.

To use the **SunIoConverter** class, you must install the jConnect sample applications. See the *Sybase jConnect for JDBC Installation Guide* for complete instructions on installing jConnect and its components, including the sample applications. Once the samples are installed, set the CHARSET_CONVERTER_CLASS connection property to reference the **SunIoConverter** class in the *sample* (jConnect 4.x) or *sample2* (jConnect 5.x) subdirectory under your jConnect installation directory.

Performance Tuning for Prepared Statements in Dynamic SQL

In Embedded SQL, dynamic statements are SQL statements that need to be compiled at runtime, rather than statically. Typically, dynamic statements contain input parameters, although this is not a requirement. In SQL, the **prepare** command is used to precompile a dynamic statement and save it so that it can be executed repeatedly without being recompiled during a session.

If a statement is used multiple times in a session, precompiling it provides better performance than sending it to the database and compiling it for each use. The more complex the statement, the greater the performance benefit.

If a statement is likely to be used only a few times, precompiling it may be inefficient because of the overhead involved in precompiling, saving, and later deallocating it in the database.

Precompiling a dynamic SQL statement for execution and saving it in memory uses time and resources. If a statement is not likely to be used multiple times during a session, the costs of doing a database **prepare** may outweigh its benefits. Another consideration is that once a dynamic SQL statement is prepared in the database, it is very similar to a stored procedure. In some cases, it may be preferable to create stored procedures and have them reside on the server, rather than defining prepared statements in the application. This is discussed under “Choosing Between Prepared Statements and Stored Procedures” on page 112.

You can use jConnect to optimize the performance of dynamic SQL statements on a Sybase database as follows:

- Create **PreparedStatement** objects that contain precompiled statements in cases where a statement is likely to be executed several times in a session.
- Create **PreparedStatement** objects that contain uncompiled SQL statements in cases where a statement will be used very few times in a session.

As described in the following sections, the optimal way to set the DYNAMIC_PREPARE connection property and create **PreparedStatement** objects is likely to depend on whether your application needs to be portable across JDBC drivers or whether you are writing an application that allows jConnect-specific extensions to JDBC.

jConnect 4.1 and later provide performance tuning features for dynamic SQL statements.

Choosing Between Prepared Statements and Stored Procedures

If you create a **PreparedStatement** object containing a precompiled dynamic SQL statement, once the statement is compiled in the database, it effectively becomes a stored procedure that is retained in memory and attached to the data structure associated with your session. In deciding whether to maintain stored procedures in the database or to create **PreparedStatement** objects containing compiled SQL statements in your application, resource demands and database and application maintenance are important considerations:

- Once a stored procedure is compiled, it is globally available across all connections. In contrast, a dynamic SQL statement in a **PreparedStatement** object needs to be compiled and deallocated in every session that uses it.
- If your application accesses multiple databases, using stored procedures means that the same stored procedures need to be available on all target databases. This can create a database maintenance problem. If you use **PreparedStatement** objects for dynamic SQL statements, you avoid this problem.
- If your application creates **CallableStatement** objects for invoking stored procedures, you can encapsulate SQL code and table references in the stored procedures. You can then modify the underlying database or SQL code without have to change the application.

Prepared Statements in Portable Applications

If your application is to run on databases from different vendors and you want some **PreparedStatement** objects to contain precompiled statements and others to contain uncompiled statements, proceed as follows:

- When you access a Sybase database, make sure that the DYNAMIC_PREPARE connection property is set to “true.”
- To return **PreparedStatement** objects containing precompiled statements, use **Connection.prepareStatement( )** in the standard way:

  ```
  PreparedStatement ps_precomp =
   Connection.prepareStatement(sql_string);
  ```

- To return **PreparedStatement** objects containing uncompiled statements, use **Connection.prepareCall( ).**

Connection.prepareCall( ) returns a **CallableStatement** object, but since **CallableStatement** is a subclass of **PreparedStatement**, you can upcast a **CallableStatement** object to a **PreparedStatement** object, as in the following example:

```
PreparedStatement ps_uncomp =
 Connection.prepareCall(sql_string);
```

The **PreparedStatement** object *ps_uncomp is* guaranteed to contain an uncompiled statement, since only **Connection.prepareStatement( )** is implemented to return **PreparedStatement** objects containing precompiled statements.

Prepared Statements in Applications with jConnect Extensions

If you are not concerned about portability across drivers, you can write code that uses **SybConnection.prepareStatement( )** to specify whether a **PreparedStatement** object contains precompiled or uncompiled statements. In this case, how you code prepared statements is likely to depend on whether most of the dynamic statements in an application are likely to be executed many times or only a few times during a session.

If Most Dynamic Statements Are Executed Very Few Times

For an application in which most dynamic SQL statements are likely to be executed only once or twice in a session:

- Set the connection property DYNAMIC_PREPARE to “false.”
- To return **PreparedStatement** objects containing uncompiled statements, use **Connection.prepareStatement( )** in the standard way:

  ```
  PreparedStatement ps_uncomp =
   Connection.prepareStatement(sql_string);
  ```

- To return **PreparedStatement** objects containing precompiled statements, use **SybConnection.prepareStatement( )** with *dynamic* set to “true:”

```
PreparedStatement ps_precomp =
 (SybConnection)conn.prepareStatement(sql_string, true);
```

If Most Dynamic Statements Are Executed Many Times in a Session

If most of the dynamic statements in an application are likely to be executed many times in the course of a session, proceed as follows:

- Set the connection property DYNAMIC_PREPARE to "true."
- To return **PreparedStatement** objects containing precompiled statements, use **Connection.prepareStatement( )** in the standard way:

```
    PreparedStatement ps_precomp =
     Connection.prepareStatement(sql_string);
```

- To return **PreparedStatement** objects containing uncompiled statements, you can use either **Connection.prepareCall( )** (see the third bullet under Prepared Statements in Portable Applications) or **SybConnection.prepareStatement( )**, with *dynamic* set to "false:"

```
PreparedStatement ps_uncomp =
 (SybConnection)conn.prepareStatement(sql_string, false);

PreparedStatement ps_uncomp =
 Connection.prepareCall(sql_string);
```

Connection.prepareStatement( )

jConnect implements **Connection.prepareStatement( )** so you can set it to return either precompiled SQL statements or uncompiled SQL statements in **PreparedStatement** objects. If you set **Connection.prepareStatement( )** to return precompiled SQL statements in **PreparedStatement** objects, it sends dynamic SQL statements to the database to be precompiled and saved exactly as they would be under direct execution of the **prepare** command. If you set **Connection.prepareStatement( )** to return uncompiled SQL statements, it returns them in **PreparedStatement** objects without sending them to the database.

The type of SQL statement that **Connection.prepareStatement( )** returns is determined by the connection property DYNAMIC_PREPARE, and applies throughout a session.

For Sybase-specific applications, jConnect 5.0 provides a **prepareStatement( )** method under the jConnect **SybConnection** class. **SybConnection.prepareStatement( )** allows you to specify whether an individual dynamic SQL statement is to be precompiled, independent of the session-level setting of the DYNAMIC_PREPARE connection property.

DYNAMIC_PREPARE Connection Property

DYNAMIC_PREPARE is a Boolean-valued connection property for enabling dynamic SQL prepared statements:

- If DYNAMIC_PREPARE is set to "true," every invocation of **Connection.prepareStatement( )** during a session attempts to return a precompiled statement in a **PreparedStatement** object.

  In this case, when a **PreparedStatement** is executed, the statement it contains is already precompiled in the database, with place holders for dynamically assigned values, and the statement needs only to be executed.

- If DYNAMIC_PREPARE is set to "false" for a connection, the **PreparedStatement** object returned by **Connection.prepareStatement( )** does not contain a precompiled statement.

  In this case, each time a **PreparedStatement** is executed, the dynamic SQL statement it contains must be sent to the database to be both compiled and executed.

The default value for DYNAMIC_PREPARE is "false."

In the following example, DYNAMIC_PREPARE is set to "true" to enable precompilation of dynamic SQL statements. In the example, **props** is a **Properties** object for specifying connection properties.

```
...
 props.put("DYNAMIC_PREPARE", "true")
 Connection conn = DriverManager.getConnection(url, props);
```

When DYNAMIC_PREPARE is set to "true," note that:

- Not all dynamic statements can be precompiled under the **prepare** command. The SQL-92 standard places some restrictions on the statements that can be used with the **prepare** command, and individual database vendors may have their own constraints.
- If the database generates an error because it is unable to precompile and save a statement sent to it through **Connection.prepareStatement( )**, jConnect traps the error and returns a **PreparedStatement** object containing an uncompiled dynamic SQL statement. Each time the **PreparedStatement** object is executed, the statement is re-sent to the database to be compiled and executed.
- A precompiled statement resides in memory in the database and persists either to the end of a session or until its **PreparedStatement** object is explicitly closed. Garbage collection on a **PreparedStatement** object does not remove the prepared statement from the database.

As a general rule, you should explicitly close every **PreparedStatement** object after its last use to prevent prepared statements from accumulating in server memory during a session and slowing performance.

SybConnection.prepareStatement( )

If your application allows jConnect-specific extensions to JDBC, you can use the **SybConnection.prepareStatement( )** extension method to return dynamic SQL statements in **PreparedStatement** objects:

```
PreparedStatement SybConnection.prepareStatement(String sql_stmt,
   boolean dynamic) throws SQLException
```

SybConnection.prepareStatement( ) can return **PreparedStatement** objects containing either precompiled or uncompiled SQL statements, depending on the setting of the *dynamic* parameter. If *dynamic* is “true,” **SybConnection.prepareStatement( )** returns a **PreparedStatement** object with a precompiled SQL statement. If *dynamic* is “false,” it returns a **PreparedStatement** object with an uncompiled SQL statement.

The following example shows the use of **SybConnection.prepareStatement( )** to return a **PreparedStatement** object containing a precompiled statement:

```
PreparedStatement precomp_stmt =
   ((SybConnection) conn).prepareStatement( "SELECT * FROM
   authors WHERE au_fname LIKE ?", true);
```

In the example, the connection object *conn* is downcast to a **SybConnection** object to allow the use of **SybConnection.prepareStatement( )**. Note that the SQL string passed to **SybConnection.prepareStatement( )** will be precompiled in the database, even if the connection property DYNAMIC_PREPARE is “false.”

If the database generates an error because it is unable to precompile a statement sent to it through **SybConnection.prepareStatement( )**, jConnect throws a **SQLException** and the call fails to return a **PreparedStatement** object. This is unlike **Connection.prepareStatement( )**, which traps SQL errors and, in the event of an error, returns a **PreparedStatement** object containing an uncompiled statement.

Cursor Performance

When you use the **Statement.setCursorName( )** method or the **setFetchSize( )** method in the **SybCursorResultSet** class, jConnect creates a cursor in the database. Other methods cause jConnect to open, fetch, and update a cursor.

Versions of jConnect earlier than version 4.0 can create and manipulate cursors only by sending SQL statements with explicit cursor commands to the database for parsing and compilation.

jConnect version 4.0 and later create and manipulate cursors either by sending SQL statements to the database or by encoding cursor commands as tokens within the Tabular Data Stream (TDS) communication protocol. Cursors of the first type are "language cursors;" cursors of the second type are "protocol cursors."

Protocol cursors provide better performance than language cursors. In addition, not all databases support language cursors. For example, Adaptive Server Anywhere databases do not support language cursors.

In jConnect, the default condition is for all cursors to be protocol cursors. However, the LANGUAGE_CURSOR connection property gives you the option of having cursors created and manipulated through language commands in the database.

LANGUAGE_CURSOR Connection Property

LANGUAGE_CURSOR is a Boolean-valued connection property in jConnect that allows you to determine whether cursors are created as protocol cursors or language cursors:

- If LANGUAGE_CURSOR is set to "false," all cursors created during a session are protocol cursors, which provide better performance. jConnect creates and manipulates the cursors by sending cursor commands as tokens in the TDS protocol.

  By default, LANGUAGE_CURSOR is set to "false."

- If LANGUAGE_CURSOR is set to "true," all cursors created during a session are language cursors. jConnect creates and manipulates the cursors by sending SQL statements to the database for parsing and compilation.

There is no known advantage to setting LANGUAGE_CURSOR to “true,” but the option is provided in case an application displays unexpected behavior when LANGUAGE_CURSOR is “false.”

CHAPTER 5

Migrating jConnect Applications

This chapter explains how to migrate applications that use Sybase extensions from jConnect version 4.0 or earlier to use jConnect versions 4.1 and later.

The following topics are covered:

Migrating jConnect Applications

Migrating Applications to jConnect 4.1

jConnect 4.1 is backward compatible with previous versions of jConnect. All existing applications should continue to work without recompiling.

When you develop new applications that use Sybase extensions, use the interfaces located in **com.sybase.jdbcx**.

This common interface allows you to upgrade applications to jConnect version 4.1 and later with minimal changes. See “Sybase Extension Changes” on page 123 for changes made in the Sybase extensions.

Migrating Applications to jConnect 5.x

The package and class for the Sybase jConnect 5.x driver is *com.sybase.jdbc2.jdbc.SybDriver*. This is different from previous versions of jConnect. Applications need to change their source code where they load SybDriver and recompile with the Java™ 2 Platform.

Here is an example of loading the driver in jConnect 5.0:

```
Driver d =
(Driver)Class.forName("com.sybase.jdbc2.jdbc.SybDriver").newInstance();
DriverManager.registerDriver(d);
```

If your applications use Sybase extensions to the JDBC API, change your imports from using **com.sybase.jdbc** and/or **com.sybase.utils** to **com.sybase.jdbcx**. See “Sybase Extension Changes” on page 123 for changes made to the Sybase extensions.

See the samples provided with jConnect for more examples of how to use Sybase extensions.

Migrating Applications to jConnect 4.2 and 5.2

If you upgrade to jConnect 4.2 or 5.2 from previous versions, the following table shows which upgrade paths require you to make changes and recompile the source code.

Legend:

A Recommend changing to the **com.sybase.jdbcx** package

B Change CLASSPATH for new installation structure

C Recompile to use new jConnect 5.x driver

See below for additional details.

Upgrading From	**To jConnect Version**			
jConnect Version	*4.1*	*4.2*	*5.0*	*5.2*
4.0 & earlier	A	AB	AC	ABC
4.1	-	AB	AC	ABC
4.2	-	-	*not a supported path*	AC
5.0	-	-	-	B

❖ **A. Use the new Sybase extensions.**

1 Change package imports from

```
import com.sybase.jdbc.*
```

to

```
import com.sybase.jdbcx.*;
```

2 Use new Sybase extentions APIs. See “Sybase Extension Changes” on page 123.

❖ **B. Change CLASSPATH for newJDBC_HOME installation structure.**

Set JDBC_HOME to the top directory of the jConnect driver you installed. For example:

- For jConnect 4.2:

  ```
  JDBC_HOME=jConnect-4_2
  ```

- For jConnect 5.0:

  ```
  JDBC_HOME=<jConnect installation directory>
  ```

For more information on setting JDBC_HOME, see “Setting Environment Variables” in Chapter 1 of the *jConnect for JDBC Installation Guide*.

Version Change		**CLASSPATH Includes**
From	4.1	*JDBC_HOME/classes*
To	5.2	*JDBC_HOME/jconn2.jar*
From	4.1	*JDBC_HOME/classes*

Version Change		CLASSPATH Includes
To	4.2	*JDBC_HOME/classes*
From	5.0	*JDBC_HOME/classes/jconn2.jar*
To	5.2	*JDBC_HOME/classes/jconn2.jar*

❖ **C. Recompile to use new jConnect 5.x driver**

- Change the source code where the driver is loaded from:

  ```
  Class.forName("com.sybase.jdbc.SybDriver");
  ```

  to

  ```
  Class.forName("com.sybase.jdbc2.jdbc.SybDriver");
  ```

Sybase Extension Changes

A new package, **com.sybase.jdbcx**, has been added to jConnect versions 4.1, 4.2, and 5.x that contains all of the Sybase extensions to JDBC. In previous versions of jConnect, these extensions were available in the **com.sybase.jdbc** and **com.sybase.utils** packages.

com.sybase.jdbcx provides a consistent interface across different versions of jConnect. All of the Sybase extensions are defined as Java interfaces, which allows the underlying implementations to change without affecting applications built using these interfaces.

When you develop new applications that use Sybase extensions, use **com.sybase.jdbcx**. The interfaces in this package allow you to upgrade applications to versions of jConnect that follow version 4.0 with minimal changes.

Note Applications previously built using the Sybase extensions to the JDBC API, which were available in **com.sybase.jdbc** and **com.sybase.utils,** will continue to work under jConnect 4.x; however, all Sybase extensions in **com.sybase.jdbc** and **com.sybase.utils** have been marked deprecated.

Some of the Sybase extensions have been changed to accommodate the new **com.sybase.jdbcx** interface.

Change Example

If an application uses the **SybMessageHandler**, the code differences would be:

- **jConnect 4.0** code:

```
import com.sybase.jdbc.SybConnection;
import com.sybase.jdbc.SybMessageHandler;
.
.
Connection con = DriverManager.getConnection(url, props);
SybConnection sybCon = (SybConnection) con;
sybCon.setMessageHandler(new ConnectionMsgHandler());
```

- **jConnect 4.1** and later code:

```
import com.sybase.jdbcx.SybConnection;
import com.sybase.jdbcx.SybMessageHandler;
.
```

```
.
Connection con = DriverManager.getConnection(url, props);
SybConnection sybCon = (SybConnection) con;
sybCon.setSybMessageHandler(new ConnectionMsgHandler());
```

See the samples provided with jConnect for more examples of how to use Sybase extensions.

Changed Method Names

The following table lists how methods have been renamed in the new interface.

Class	Old Name	New Name
SybConnection	getCapture( )	createCapture**( )**
SybConnection	setMessageHandler( )	setSybMessageHandler**( )**
SybConnection	getMessageHandler**( )**	getSybMessageHandler**( )**
SybStatement	setMessageHandler( )	setSybMessageHandler**( )**
SybStatement	getMessageHandler**( )**	getSybMessageHandler**( )**

Debug Class

Direct static references to the **Debug** class are no longer supported, but exist in deprecated form in the **com.sybase.utils** package. To use jConnect debugging facilities, use the **getDebug( )** method of the **SybDriver** class to obtain a reference to the **Debug** class. For example:

```
import com.sybase.jdbcx.SybDriver;
import com.sybase.jdbcx.Debug;
.
.
.
SybDriver sybDriver =
      SybDriver)Class.forName
      ("com.sybase.jdbc2.jdbc.SybDriver") newInstance();
Debug sybDebug = sybDriver.getDebug();
sybDebug.debug(true, "ALL", System.out);
```

A complete list of Sybase extensions is in the jConnect javadoc documentation located in the *docs/* directory of your jConnect installation directory.

CHAPTER 6 Web Server Gateways

This chapter describes Web server gateways and explains how to use them with jConnect.

The chapter is divided into these sections:

About Web Server Gateways

If your database server runs on a different host than your Web server, or if you are developing Internet applications that must connect to a secure database server through a firewall, you need a gateway to act as a proxy, providing a path to the database server.

To connect to servers using the Secure Sockets Layer (SSL) protocol, jConnect provides a Java servlet that you can install on any Web server that supports the **javax.servlet** interfaces. This servlet enables jConnect to support encryption using the Web server as the gateway.

Note jConnect includes support for SSL on the client system. For information on jConnect's client-side support of SSL, see "Implementing Custom Socket Plug-Ins" on page 26.

jConnect includes a Cascade gateway, a slightly modified version of the Cascade Web server. The Cascade gateway supports the CONNECT HTTP method, which allows HTTP-tunnelling of Tabular Data Stream (TDS) data. You can use the Cascade gateway to connect to a database server that runs on a different host than the Web server, but that does not need to pass through a firewall. See "Using the Cascade Gateway" on page 131.

TDS Tunnelling

jConnect uses TDS to communicate with database servers. HTTP-tunnelled TDS is useful for forwarding requests. Requests from a client to a back-end server that go through the gateway contain TDS in the body of the request. The request header indicates the length of the TDS included in the request packet.

TDS is a connection-oriented protocol, whereas HTTP is not. To support security features such as encryption for Internet applications, jConnect uses a TDS-tunnelling servlet to maintain a logical connection across HTTP requests. The servlet generates a session ID during the initial login request, and the session ID is included in the header of every subsequent request. Using session IDs lets you identify active sessions, and even resume a session as long as the servlet has an open connection using that specific session ID.

The logical connection provided by the TDS-tunnelling servlet enables jConnect to support encrypted communication between two systems—for example, a jConnect client with the CONNECT_PROTOCOL connection property set to "https" connecting to a Web server running the TDS-tunnelling servlet.

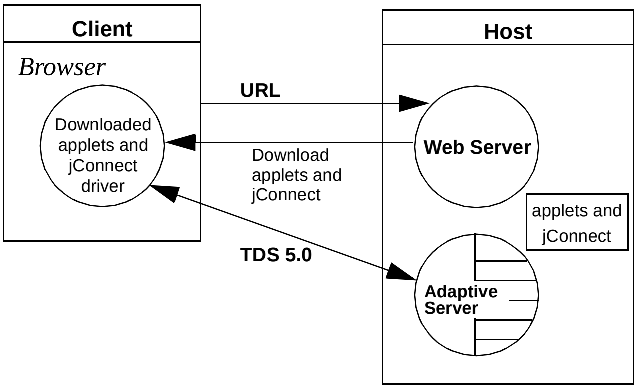
jConnect and Gateway Configuration

There are several options for setting up your Web servers and Adaptive Servers. Four common configurations are described below. These examples show where to install the jConnect driver and when to use the Cascade gateway or a gateway with the TDS-tunnelling servlet.

Web Server and Adaptive Server on One Host

In this two-tier configuration, the Web server and Adaptive Server are both installed on the same host.

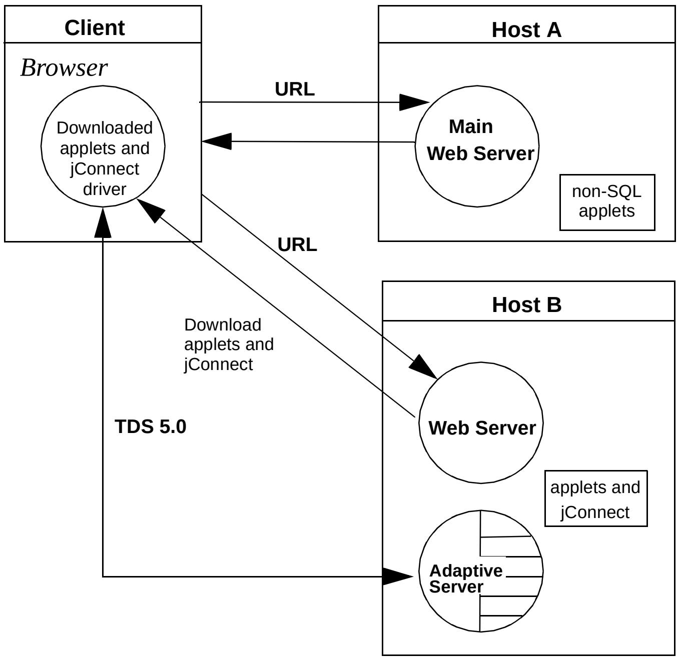
- Install jConnect on the Web server host.
- No gateway required.



Dedicated JDBC Web Server and Adaptive Server on One Host

With this configuration, you have a separate host for your main Web server. A second host is shared by a Web server specifically for Adaptive Server access and the Adaptive Server. Links from the main server direct requests requiring SQL access to the dedicated Web server.

- Install jConnect on the second (Adaptive Server) host.
- No gateway required.

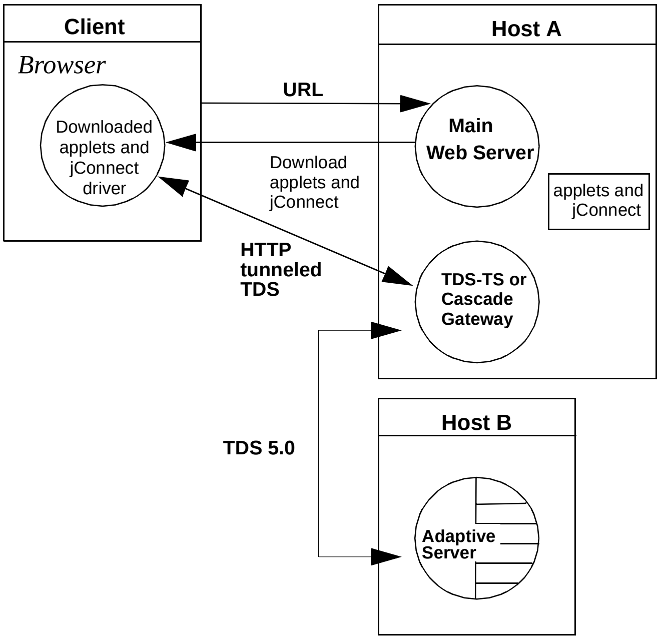


Web Server and Adaptive Server on Separate Hosts

In this three-tier configuration, the Adaptive Server is on a separate host from the Web server. jConnect requires a gateway to act as a proxy to the Adaptive Server.

- Install jConnect on the Web server host.

- Requires either a TDS-tunnelling servlet or the Cascade gateway.

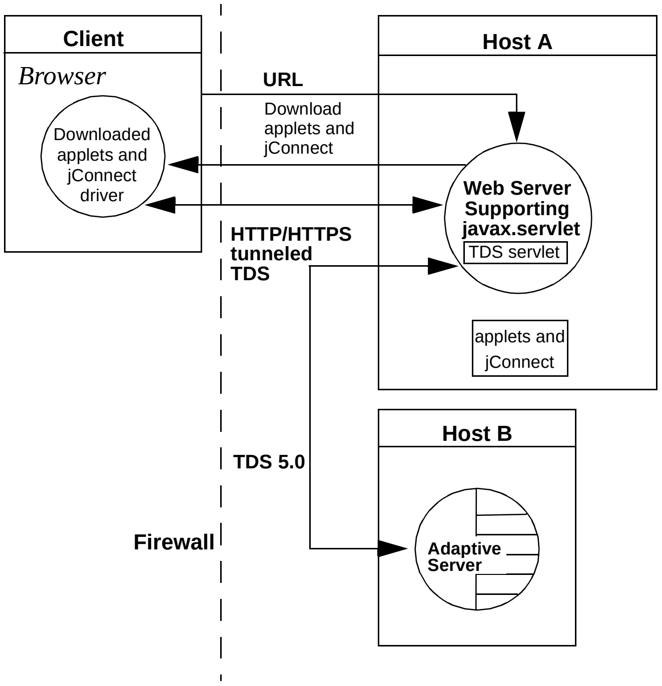


Connecting to a Server Through a Firewall

To connect to a server protected by a firewall, you must use a Web server with the TDS-tunnelling servlet to support transmission of database request responses over the Internet.

- Install jConnect on the Web server host.

- Requires a Web server that supports the **javax.servlet** interfaces.



Using the Cascade Gateway

If your database server runs on a different host than the Web server, even if there is no firewall, you need a gateway to act as a proxy, providing a path to the database server.

The jConnect installation includes a Cascade gateway, a slightly modified version of the Cascade Web server written in Java by David Wilkerson (email: davidw@cascade.org.uk; Web site: http://www.cascade.org.uk/).

The Cascade gateway receives and forwards packets, switching from HTTP to TDS when sending a request to the Adaptive Server, and from TDS to HTTP when returning results.

Note The Cascade gateway does not support encryption, and therefore is not suitable for Internet applications that connect to a back-end server through a firewall.

Usage Requirements

- If you did not install jConnect in the default installation directory, you must edit *www.dos* (DOS) or *www.template* (UNIX) and change all default installation directory references to point to the literal path where you installed jConnect.
- The Cascade gateway and your Web server must run on the same host machine. This way, applets connect to the same host as the Web server, but to the port controlled by the Cascade gateway. The gateway routes the request to the appropriate database. To see how this is accomplished, review the code in *Isql.java* and *gateway.html* in the *sample* (jConnect 4.x) or *sample2* (jConnect 5.x) subdirectory of the jConnect installation directory. Search for "proxy."

If your database server runs on the same host as your Web server, you do not need the Cascade gateway.

Installing the Cascade Gateway

The Cascade gateway is installed when you perform a full jConnect installation, either using the jConnect installer, or using the *install.bat* or *install.sh* files. You can also use the jConnect installer to install only the Cascade gateway files, if necessary. See the *Sybase JConnect for JDBC Installation Guide and Release Bulletin*.

Starting the Cascade Gateway

Follow the platform-specific instructions below to test the Cascade gateway.

Windows NT and Windows 95

1. At a DOS prompt, change to the jConnect installation directory.

   Your JDBC_HOME environment variable should be set to this directory.

2. Start the Cascade gateway by entering:

   ```
   httpd
   ```

   If the command executes successfully, you see output from *httpd.bat*, ending with:

   ```
   HTTPDServer www.dos
   ```

UNIX

Change to the directory where you installed jConnect (the JDBC_HOME directory). Type the following command:

```
sh httpd.sh &
```

Troubleshooting

- If you do not see a message after you enter the **httpd** command, the server is not running. Retry the command in verbose mode.

  For Windows, go to a DOS prompt and enter:

  ```
  httpd -Dverbose=1 > filename
  ```

  For UNIX, enter:

  ```
  sh httpd.sh -Dverbose=1 > filename &
  ```

  In these commands, *filename* is the output file for debug messages.

- If you receive this error message:

  ```
  HTTPServer: IOException: getRequest() Address
  already in use
  ```

  it means another process is running on the port specified in the *www.dos* (DOS) or *www.template* (UNIX) file, located in JDBC_HOME. This error occurs when you start the gateway.

You can:

- Stop the process that is currently running on the specified port. After verifying that the process has terminated, try starting the gateway;

  or

- Change the port number in the *www.dos* or *www.template* file, then modify the *gateway.html* file, located in the *sample* (jConnect 4.x) or *sample2* (jConnect 5.x) subdirectory of JDBC_HOME, by changing the *proxy* parameter to "localhost:*new_port."*

  If your host is not "localhost" (that is, your Cascade HTTP server and browser are on different hosts), make sure that the *proxy* parameter uses the remote host name rather than "localhost."

Testing the Cascade Gateway

To check your setup and test the Cascade gateway, you can run a validation program that connects to the Sybase demonstration database.

Note The Validate program uses "localhost:8000" to test your gateway.

From the DOS prompt for Windows NT or Windows 95, or at a UNIX prompt, change to the JDBC_HOME directory.

For jConnect 4.x, type

```
java sample.SybSample Validate
```

For jConnect 5.x, type

```
java sample2.SybSample Validate
```

If the validation is successful, you see the jConnect version number and the message "Connected successfully" in the Sample output window.

Troubleshooting

If you receive a "Bad command or file name" error (Windows 95) or a "Name specified is not recognized as an internal or external command" error (Windows NT), make sure that your path includes the *\bin* subdirectory of the JDK home directory.

Reading the *index.html* File

Use your Web browser to view the *index.html* file in your jConnect installation directory. *index.html* provides links to the jConnect documentation and sample code.

Note If you use Netscape on the same machine where you have installed jConnect, make sure that your browser does not have access to your CLASSPATH environment variable. See “Restrictions on Setting CLASSPATH When You Use Netscape” in Chapter 3 of the *Sybase jConnect for JDBC Installation Guide and Release Bulletin*.

---

1 Open your Web browser.

2 Enter the URL that matches your setup. For example, if your browser and the Cascade gateway are running on the same host, enter:

```
http://localhost:8000/index.html
```

If the browser and the Cascade gateway are running on different hosts, enter:

```
http://host:port/index.html
```

where *host* is the name of the host on which the Cascade gateway is running, and *port* is the listening port.

Troubleshooting

If you entered the correct host, port, and file information, but the browser cannot open this link, the Cascade gateway is not running. See “Starting the Cascade Gateway” on page 132.

Running the Sample Isql Applet

After loading the *index.html* file in your browser:

1 Click “Run Sample JDBC Applets.”

This takes you to the jConnect Sample Programs page.

2 Move down the Sample Programs page to find the table under “Executable Samples.”

3 Locate “Isql.java” in the table and click on “Run” at the end of the row.

The sample **Isql.java** applet prompts for a simple query on a sample database and displays the results. The applet displays a default Adaptive Server host name, port number, user name (*guest*), password (*sybase*), database, and query. Using the default values, the applet connects to the Sybase demonstration database. It returns results after you click Go.

Troubleshooting

Under UNIX, if the applet does not appear as expected, you can modify the applet screen dimensions:

1. Use a text editor to edit the following:

   For jConnect 4.x

   $JDBC_HOME/sample/gateway.html

   For jConnect 5.x

   $JDBC_HOME/sample2/gateway.html

2. Change the height parameter in line 7 to 650. You can experiment with different height settings.

3. Reload the Web page on your browser.

Defining a Connection to the Cascade Gateway

To define a connection in your application that uses the Cascade gateway, put the name of the host on which the Cascade gateway is running in the URL:

host:*port*

where *host* is the name of the host on which the Cascade gateway is running, and *port* is the listening port.

Using the TDS-Tunnelling Servlet

In order to use the TDS-tunnelling servlet, you need a Web server that supports the **javax.servlet** interfaces, such as Sun Microsystems, Inc.'s Java™ Web server. When you install the Web server, include the jConnect TDS-tunnelling servlet in the list of active servlets. You can also set servlet parameters to define connection timeouts and maximum packet size.

With the TDS-tunnelling servlet, requests from a client to the back-end server that go through the gateway include a GET or POST command, the TDS session ID (after the initial request), back-end address, and status of the request.

TDS is in the body of the request. Two header fields indicate the length of the TDS stream and the session ID assigned by the gateway.

When the client sends a request, the Content-Length header field indicates the size of the TDS content, and the request command is POST. If there is no TDS data in the request because the client is either retrieving the next portion of the response data from the server, or closing the connection, the request command is GET.

The following example illustrates how information is passed between the client and an HTTPS gateway using the TDS-tunneled HTTPS protocol. This example shows a connection to a back-end server named DBSERVER with a port number "1234."

Table 6-1: Client to gateway login request. No session ID.

Query	POST/tds?ServerHost=dbserver&ServerPort=1234& Operation=more HTTP/1.0
Header	Content-Length: 605
Content (TDS)	Login request

Table 6-2: Gateway to client. Header contains session ID assigned by the TDS servlet.

Query	200 SUCCESS HTTP/1.0
Header	Content-Length: 210 TDS-Session: TDS00245817298274292
Content (TDS)	Login acknowledgment EED

Table 6-3: Client to gateway. Headers for all subsequent requests contain the session ID.

Query	POST/tds?TDS-Session=TDS00245817298274292&Operation=more HTTP/1.0
Header	Content-Length: 32
Content (TDS)	Query “SELECT * from authors”

Table 6-4: Gateway to client. Headers for all subsequent responses contain the session ID.

Query	200 SUCCESS HTTP/1.0
Header	Content-Length: 2048 TDS-Session: TDS00245817298274292
Content (TDS)	Row format and some rows from query response

TDS-tunnelling Servlet System Requirements

To use the jConnect servlet for TDS-tunneled HTTP, you need:

- A Web server that supports **javax.servlet** interfaces. To install the server, follow the instructions that are provided with it.
- A Web browser that supports JDK 1.1, such as Netscape 4.0, Internet Explorer 4.0, or HotJava.

Installing the Servlet

Your jConnect installation includes a *gateway* subdirectory (jConnect 4.x) or *gateway2* subdirectory (jConnect 5.x) under the *classes* directory. The subdirectory contains files required for the TDS-tunnelling servlet.

Copy the jConnect **gateway** package to a *gateway* subdirectory (jConnect 4.x) or *gateway2* subdirectory (jConnect 5.x) under your Web server’s servlets directory. Once you have copied the servlets, activate the servlets by following the instructions for your Web server.

Setting Servlet Arguments

When you add the servlet to your Web server, you can enter optional arguments to customize performance:

- *SkipDoneProc [true|false]* – Sybase databases often return row count information as intermediate processing steps are performed during the execution of a query. Usually client applications ignore this data. By setting *SkipDoneProc* to "true," the servlet will remove this extra information from responses "on the fly," which reduces network usage and processing requirements on the client. This is particularly useful when using HTTPS/SSL because the unwanted data does not get encrypted/decrypted before it is ignored.
- *TdsResponseSize* – Set the maximum TDS packet size for the tunneled HTTPS. A larger *TdsResponseSize* is more efficient if you have only a few users with a large volume of data. Use a smaller *TdsResponseSize* if you have many users making smaller transactions.
- *TdsSessionIdleTimeout* – Define the amount of time (in milliseconds) that the server connection can remain idle before the connection is automatically closed. The default *TdsSessionIdleTimeout* is 600,000 (10 minutes).

  If you have interactive client programs that may be idle for long periods of time and you don't want the connections broken, increase the *TdsSessionIdleTimeout*.

  You can also set the connection timeout value from the jConnect client using the SESSION_TIMEOUT connection property. This is useful if you have specific applications that may be idle for longer periods. In this case, set a longer timeout for those connections with the SESSION_TIMEOUT connection property, rather than setting it for the servlet.
- *Debug* – Turn on debugging. See "Debugging with jConnect" on page 94.

Enter the servlet arguments in a comma-delimited string. For example:

```
TdsResponseSize=[size],TdsSessionIdleTimeout=[timeout],Debug=true
```

Refer to your Web server documentation for complete instructions on entering servlet arguments.

Invoking the Servlet

jConnect determines when to use the gateway where the TDS-tunnelling servlet is installed based on the path extension of the *proxy* connection property. jConnect recognizes the servlet path extension to the *proxy* and invokes the servlet on the designated gateway.

Define the connection URL using this format:

```
http://host:port/TDS-servlet-path
```

jConnect invokes the TDS-tunnelling servlet on the Web server to tunnel TDS through HTTP. The servlet path must be the path you defined in your Web server's servlet alias list.

Tracking Active TDS Sessions

You can view information about active TDS sessions, including the server connections for each session. Use your Web browser to open the administrative URL:

```
http://host:port/TDS-servlet-path?Operation=list
```

For example, if your server is MYSERVER and the TDS servlet path is */tds*, you would enter:

```
http://myserver:8080/tds?Operation=list
```

This shows you a list of active TDS sessions. You can click on a session to see more information, including the server connection.

Terminating TDS Sessions

You can use the URL described above to terminate any active TDS session. Click on an active session from the list of sessions on the first page, then click Terminate This Session.

Resuming a TDS Session

You can set the SESSION_ID connection property so that, if necessary, you can resume an existing open connection. When you specify a SESSION_ID, jConnect skips the login phase of the protocol and resumes the connection with the gateway using the designated session ID. If the session ID you specified does not exist on the servlet, jConnect throws a SQL exception the first time you attempt to use the connection.

TDS Tunnelling and Netscape Enterprise Server 3.5.1 on Solaris

Netscape Enterprise Server 3.5.1 does not support the **javax.servlet.ServletConfig.getInitParameters( )** or **javax.servlet.ServletConfig.getInitParameterNames( )** methods. To provide the necessary parameter values, you need to replace calls to **getInitParameter( )** and **getInitParameterNames( )** with hard-coded parameter values in *TDSTunnelServlet.java*.

To enter the required parameter values in *TDSTunnelServlet.java* and use TDS tunnelling with Netscape Enterprise Server 3.5.1 on Solaris:

1. Hard code parameter values in *TDSTunnelServlet.java*.
2. Create *.class* files from the class declarations in *TDSTunnelServlet.java*.

   This should result in the following files:

   - *TDSTunnelServet.class*
   - *TdsSession.class*
   - *TdsSessionManager.class*
3. Create a directory for the *.class* files under your Netscape Enterprise Server 3.5.1 (NSE_3.5.1) installation directory, as follows:

```
mkdir NSE_3.5.1_install_dir/plugins/java/servlets/gateway
```

4. Copy the *.class* files derived from *TDSTunnelServlet.java* to the directory you just created.
5. Copy the classes under *$JDBC_HOME/classes/com/sybase* to *NSE_3.5.1_install_dir/docs/com/sybase*.

   An easy way to do this is to recursively copy everything under *$JDBC_HOME/classes* to *NSE_3.5.1_install_dir/docs*, as:

   ```
   cp -r $JDBC_HOME/classes NSE_3.5.1_install_dir/docs
   ```

   This copies a number of files and directories that are not under *$JDBC_HOME/classes/com/sybase*. The extra files and directories are harmless, but take up disk space. You can delete them to reclaim the disk space.
6. Set the *proxy* URL to the TDS-tunnelling servlet.

   For example, in *$JDBC_HOME/sample/gateway.html*, you would edit the *proxy* parameter to appear as follows:

```
<param name=proxy value=”http://hostname/servlet/
 gateway_name.TDSTunnel_Servlet_name”>
```

APPENDIX A

SQL Exception and Warning Messages

The following table lists the SQL exception and warning messages that you may encounter when using jConnect.

SQLState	Message/Description/Action
010DP	`Duplicate connection property _____ ignored.` *Description:* A connection property is defined twice. It may be defined twice in the driver connection properties list with different capitalization, for example “password” and “PASSWORD.” Connection property names are not case-sensitive, and therefore jConnect does not distinguish between property names with the same name but different capitalization. The connection property may also be defined both in the connection properties list, and in the URL. In this case, the property value in the connection property list takes precedence. *Action:* Make sure your application defines the connection property only once. However, you may want you application to take advantage of the precedence of connection properties defined in the property list over those defined in the URL. In this case, you can safely ignore this warning.
010HA	`The server denied your request to use the high-availability feature. Please reconfigure your database, or do not request a high-availability session.` *Description:* The REQUEST_HA_SESSION connection property was not set to “true,” and the server to which jConnect attempted a connection did not allow the connection. *Action:* Reconfigure the server to support high-availability Failover or do not set REQUEST_HA_SESSION to “true.”
010HD	`Sybase high-availability failover is not supported by this type of database server.` *Description:* The database to which jConnect attempted a connection does not support high-availability Failover. *Action:* Connect only to database servers that support high-availability Failover.

SQLState	Message/Description/Action
010MX	`Metadata accessor information was not found on this database. Please install the required tables as mentioned in the jConnect documentation. Error encountered while attempting to retrieve metadata information: _____` *Description:* The server may not have the necessary stored procedures for returning metadata information. *Action:* Make sure that stored procedures for providing metadata are installed on the server. See "Installing Stored Procedures" in Chapter 3 of the *jConnect for JDBC Installation Guide*.
010P4	`An output parameter was received and ignored.` *Description:* The query you executed returned an output parameter but the application result-processing code did not fetch it so it was ignored. *Action:* If your application needs the output parameter data, you must rewrite the application so it can get it. This may require using a CallableStatement to execute the query, and adding calls to registerOutputParameter() and getXXX().
010PF	`One or more jars specified in the PRELOAD_JARS connection property could not be loaded.` *Description:* This happens when using the DynamicClassLoader with the PRELOAD_JARS connection property set to a comma-delimited list of JAR names. When the DynamicClassLoader opens its connection to the server from which the classes are to be loaded, it attempts to "preload" all the JARs mentioned in this connection property. If one or more of the JAR names specified doesn't exist on the server, the above error message results. *Action:* Verify that every JAR file mentioned in your application's PRELOAD_JARS connection property exists and is accessible on the server.
010RC	`The requested ResultSet type and concurrency is not supported. They have been converted.` *Description:* You requested a type and concurrency combination for the ResultSet that is not supported. The requested values had to be converted. *Action:* Request a type and concurrency combination for the ResultSet that is supported.

SQLState	Message/Description/Action
010SJ	`Metadata accessor information was not found on this database. Please install the required tables as mentioned in the jConnect documentation.` *Description:* The metadata information is not configured on the server. *Action:* If your application requires metadata, install the stored procedures for returning metadata that come with jConnect (see "Installing Stored Procedures" in Chapter 3 of the *jConnect for JDBC Installation Guide*). If you do not need metadata, set the USE_METADATA property to "false."
010SK	`Database cannot set connection option _____.` *Description:* Your application attempted an operation that the database you are connected to does not support. *Action:* You may need to upgrade your database or make sure that the latest version of metadata information is installed on it.
010SN	`Permission to write to file was denied. File: _____. Error message: _____` *Description:* Permission to write to a file specified in the PROTOCOL_CAPTURE connection property is denied because of a security violation in the VM. This can occur if an applet attempts to write to the specified file. *Action:* If you are attempting to write to the file from an applet, make sure that the applet has access to the target file system.
010SP	`File could not be opened for writing. File: _____. Error message: _____` *Action:* Make sure that the file name is correct and that the file is writable.
010TP	`The connection's initial character set,_____, could not be converted by the server. The server's proposed character set,_____, will be used, with conversions performed by jConnect.` *Description:* The server cannot use the character set initially requested by jConnect, and has responded with a different character set. jConnect accepts the change, and will perform the necessary character-set conversions. The message is strictly informational and has no further consequences. *Action:* To avoid this message, set the CHARSET connection property to a character set that the server supports.

SQLState	Message/Description/Action
010UF	`Attempt to execute use database command failed. Error message:_____` *Description:* jConnect was unable to connect to the database specified in the connection URL. Two possibilities are: • The name was entered incorrectly in the URL. • USE_METADATA is "true" (the default condition), but the stored procedures for returning metadata have not been installed. As a result, jConnect tried to execute the use *database* command with the database in the URL, but the command failed. This may be because you attempted to access a SQL Anywhere database. SQL Anywhere databases do not support the use *database* command. *Action:* Make sure the database name in the URL is correct. Make sure that the stored procedures for returning metadata are installed on the server (see "Installing Stored Procedures" in Chapter 3 of the *jConnect for JDBC Installation Guide and Release Bulletin*). If you are attempting to access a SQL Anywhere database, either do not specify a database name in the URL or set USE_METADATA to "false."
010UP	`Unrecognized connection property _____ ignored.` *Description:* You attempted to set a connection property in the URL that jConnect does not currently recognize. jConnect ignores the unrecognized property. *Action:* Check the URL definition in your application to make sure it references only valid jConnect driver connection properties.
0100V	`The version of TDS protocol being used is too old. Version: _____` *Description: The server does not support the required version of the TDS protocol. jConnect requires version 5.0 or later.* *Action:* Use a server that supports the required version of TDS. See the jConnect installation guide's system requirements section for details.
JW0I0	`I/O layer: thread operation failed.` *Description:* An internal error occurred with a timed I/O stream. *Action:* Close and reopen the connection.
JZ001	`User name property '_____' too long. Maximum length is 30.` *Action:* Do not exceed the 30 byte maximum.
JZ002	`Password property '_____' too long. Maximum length is 30.` *Action:* Do not exceed the 30-byte maximum.

SQLState	Message/Description/Action
JZ003	`Incorrect URL format. URL: _____` *Action:* Verify the URL format. See "URL Connection Property Parameters" on page 18. If you are using the PROXY connection property, you may get a JZ003 exception while trying to connect if the format for the PROXY property is incorrect. The PROXY format for the Cascade proxy is: *ip_address*:*port_number* The PROXY format for the TDS Tunneling Servlet is: http[s]://*host*:*port*/*tunneling_servlet_alias*
JZ004	`User name property missing in DriverManager.getConnection(..., Properties)` *Action:* Provide the required user property.
JZ006	`Caught IOException: _____` *Description:* An unexpected I/O error was detected from a lower layer. When such I/O exceptions are caught, they are re-thrown as SQL exceptions using the ERR_IO_EXCEPTION JZ006 sqlstate. These errors are often the result of network communication problems. *Action:* Try increasing the statement cache size.
JZ008	`Invalid column index value _____.` *Description:* You have requested a column index value of less than 1 or greater than the highest available value. *Action:* Check call to the getXXX() method and the text of the original query, or be sure to call rs.next().
JZ009	`Error encountered in conversion. Error message: _____` *Description:* Some of the possibilities are: • A conversion between two incompatible datatypes was attempted, such as date to int. • There was an attempt to convert a string containing a non-numeric character to a numeric type. • There was a formatting error, such as an incorrectly formatted time/date string. *Action:* Make sure that the JDBC specification supports the attempted type conversion. Make sure that strings are correctly formatted. If a string contains non-numeric characters, do not attempt to convert it to a numeric type.

SQLState	Message/Description/Action
JZ00B	`Numeric overflow.` *Description:* You tried to send a BigInteger as a TDS numeric, and the value was too large, or you tried to send a Java long as an int and the value was too large. *Action:* These values cannot be stored in Sybase. For long, consider using a Sybase numeric. There is no solution for Bignum.
JZ00E	`Attempt to call execute() or executeUpdate() for a statement where setCursorName() has been called.` *Action:* Don't try to call execute or executeUpdate on a statement that has a cursor name set. Use a separate statement to delete or update a cursor. See "Using Cursors with Result Sets" on page 44 for more information
JZ00F	`Cursor name has already been set by setCursorName().` *Action:* Do not set the cursor name twice for a statement. Close the result set of the current cursor statement.
JZ00G	`No column values were set for this row update.` *Description:* You attempted to update a row in which no column values were changed. *Action:* To change column values in a row, call updateXX() methods before calling updateRow().
JZ00H	`The result set is not updatable. Use Statement.setResultSetConcurrencyType().` *Action:* To change a result set from read-only to updatable, use the Statement.setResultSetConcurrencyType() method or add a for update clause to your SQL select statement.
JZ00L	`Login failed. Examine the SQLWarnings chained to this exception for the reason(s).` *Action:* See message text; proceed according to the reason(s) given for the login failure.
JZ010	`Unable to deserialize an Object value. Error text: _____` *Action:* Make sure that the Java object from the database implements the Serializable interface and is in your local CLASSPATH variable.
JZ011	`Number format exception encountered while parsing numeric connection property _____.` *Description:* A non-integer value was specified for a numeric connection property. *Action:* Specify an integer value for the connection property.
JZ012	`Internal Error. Please report it to Sybase technical support. Wrong access type for connection property _____.` *Action:* Contact Sybase Technical Support.

SQLState	Message/Description/Action
JZ013	`Error obtaining JNDI entry: _____` *Action:* Correct the JNDI URL or make a new entry in the directory service.
JZ0BD	`Out of range or invalid value used for method parameter.` *Action:* Verify that the parameter value in the method is correct.
JZOBE	`BatchUpdateException: Error occurred while executing batch statement: ________.`
JZOBP	`Output parameters are not allowed in Batch Update Statements.` Action:
JZ0BR	`The cursor is not positioned on a row that supports the _____ method.` *Description:* You attempted to call a ResultSet method that is invalid for the current row position (e.g., calling insertRow() when the cursor is not on the insert row). *Action:* Do not attempt to call a ResultSet method that is invalid for the current row position.
JZOBS	`Batch Statements not supported.`
JZ0BT	`The _____ method is not supported for ResultSets of type _____.` *Description:* You attempted to call a ResultSet method that is invalid for the type of ResultSet. *Action:* Do not attempt to call a ResultSet method that is invalid for the type of ResultSet.
JZ0C0	`Connection is already closed.` *Description:* The application has already called Connection.close() on this connection object; it cannot be used any more. *Action:* Fix the code so that connection object references are nulled out whenever a connection is closed.
JZ0D0	`This jConnect installation has not been registered yet. You need to install the appropriate SybDriverKey classes.` *Action:* Go to the jConnect Web site to register your jConnect software: at http://www.sybase.com/products/internet/jconnect/). Once you register, you can download the SybDriverKey classes necessary to activate the jConnect driver.
JZ0D2	`Your Sybase JDBC license expired on _____. Please obtain a new license.` *Action:* Contact Sybase to obtain a new license for your jConnect driver.

SQLState	Message/Description/Action
JZ0D3	`Your Sybase JDBC license will expire soon. Please obtain a new license. It will expire on _____.` *Action:* Contact Sybase to obtain a new license for your jConnect driver.
JZ0D4	`Unrecognized protocol in Sybase JDBC URL:_____.` *Description:* You specified a connection URL using a protocol other than TDS, which is the only protocol currently supported by jConnect. *Action:* Check the URL definition. If the URL specifies TDS as a subprotocol make sure that the entry uses the following format and capitalization: jdbc:sybase:Tds:*host*:*port* If the URL specifies JNDI as a subprotocol, make sure that it starts with: jdbc:sybase:jndi:
JZ0D5	`Error loading protocol _____.` *Action:* Check the settings for the CLASSPATH system variable.
JZ0D6	`Unrecognized version number _____ specified in setVersion. Choose one of the SybDriver.VERSION_* values, and make sure that the version of jConnect that you are using is at or beyond the version you specify.` *Action:* See message text.
JZ0D7	`Error loading url provider _____. Error message: _____` *Action:* Check the JNDI URL to make sure it is correct.
JZ0D8	`Error initializing url provider: _____` *Action:* Check the JNDI URL to make sure it is correct.
JZ0EM	`End of data.` *Action:* Report this error to Sybase Technical Support.
JZ0H0	`Unable to start thread for event handler; event name = _____.` *Action:* Report this error to Sybase Technical Support.
JZ0H1	`An event notification was received but the event handler was not found; event name = _____.` *Action:* Report this error to Sybase Technical Support.
JZ0HC	`Illegal character '_____' encountered while parsing hexadecimal number.` *Description:* A string that is supposed to represent a binary value contains a character that is not in the range (0–9, a–f) that is required for a hexadecimal number. *Action:* Check the character values in the string to make sure they are in the required range.

SQLState	Message/Description/Action
JZ0I1	`I/O Layer: Error reading stream.` *Description:* The connection was unable to read the amount requested. Most likely, the statement time-out period was exceeded and the connection timed out. *Action:* Increase the statement time-out value.
JZ0I2	`I/O layer: Error writing stream.` *Description:* The connection was unable to write the output requested. Most likely, the statement time-out period was exceeded and the connection timed out. *Action:* Increase the statement time-out value.
JZ0I3	`Unknown property. This message indicates an internal product problem. Report this error to Sybase Technical support.` *Action:* Indicates an internal product problem. Report this error to Sybase Technical Support.
JZ0I5	`An unrecognized CHARSET property was specified: _____.` *Description:* You specified a character set code for the CHARSET connection property that is not supported. *Action:* Enter a valid character set code for the connection property. See "jConnect Character-Set Converters" on page 31.
JZ0I6	`An error occurred converting UNICODE to the charset used by the server. Error message: _____` *Action:* Choose a different character set code for the CHARSET connection property on the jConnect client that can support all the characters you need to send to the server. You may need to install a different character set on the server, too.
JZ0I7	`No response from proxy gateway.` *Description:* The Cascade or security gateway is not responding. *Action:* Make sure the gateway is properly installed and running.
JZ0I8	`Proxy gateway connection refused. Gateway response: _____` *Description:* The Web server/gateway indicated by the PROXY connection property has refused your connection request. *Action:* Check the access and error logs on the proxy to determine why the connection was refused. Make sure the proxy is a JDBC gateway.

SQLState	Message/Description/Action
JZ0I9	`This InputStream was closed.` *Description:* You tried to read an InputStream obtained from getAsciiStream(), getUnicodeStream(), or getBinaryStream(), but the InputStream was already closed. The stream may have been closed because you moved to another column or cancelled the result set and there were not enough resources to cache the data. *Action:* Increase the cache size or read columns in order.
JZ0IA	`Truncation error trying to send_____.` *Description:* There was a truncation error on character set conversion prior to sending a string. The converted string is longer than the size allocated for it. *Action:* Choose a different character set code for the CHARSET connection property on the jConnect client that can support all the characters you need to send to the server. You may need to install a different character set on the server, too.
JZ0IS	`getXXXStream may not be called on a column after it has been updated in the result set.` *Description:* After updating a column in a result set, you attempted to read the updated column value using one of the following SybResultSet methods: getAsciiStream(), getUnicodeStream(), getBinaryStream(). jConnect does not support this usage. *Action:* Do not attempt to fetch input streams from columns you are updating.
JZ0J0	`Offset and/or length values exceed the actual text/image length.` *Action*: Check the offset and/or length values you used to make sure they are correct.
JZ0NC	`wasNull called without a preceding call to get a column.` *Description:* You can only call wasNull() after a call to get a column, such as getInt() or getBinaryStream(). *Action:* Change the code to move the call to wasNull().
JZ0NE	`Incorrect URL format. URL: _____. Error message: _____` *Action:* Check the format of the URL. Make sure that the port number consists only of numeric characters.
JZ0NF	`Unable to load SybSocketFactory. Make sure that you have spelled the class name correctly, that the package is fully specified, that the class is available in your class path, and that it has a public zero-argument constructor.` *Action:* See message text.

SQLState	Message/Description/Action
JZ0P1	`Unexpected result type.` *Description:* The database has returned a result that the statement cannot return to the application, or that the application is not expecting at this point. This generally indicates that the application is using JDBC incorrectly to execute the query or stored procedure. If the JDBC application is connected to an Open Server application, it may indicate an error in the Open Server application that causes the Open Server to send unexpected sequences of results. *Action:* Use the com.sybase.utils.Debug(true, “ALL”) debugging tools to try to figure out what unexpected result is seen, and to understand its causes.
JZ0P4	`Protocol error. This message indicates an internal product problem. Report this error to Sybase technical support.` *Action:* See message text.
JZ0P7	`Column is not cached; use RE-READABLE_COLUMNS property.` *Description:* With the REPEAT_READ connection property set to “false,” an attempt was made to reread a column or read a column in the wrong order. When REPEAT_READ is “false.” You can only read the column value for a row once, and you can only read columns in ascending column-index order. For example, after reading Column 3 for a row, you cannot read its value a second time and you cannot read Column 2 for the row. *Action:* Either set REPEAT_READ to “true,” or do not attempt to reread a column value and make sure that you read columns in ascending column-index order.
JZ0P8	`The RSMDA Column Type Name you requested is unknown.` *Description:* jConnect was unable to determine the name of a column type in the ResultSetMetaData.getColumnTypeName() method. *Action:* Check to make sure that your database has the most recent stored procedures for metadata.
JZ0P9	`A COMPUTE BY query has been detected. That type of result is unsupported and has been cancelled.` *Description:* The query you executed returned COMPUTE results, which are not supported by jConnect. *Action:* Change your query or stored procedure so it does not use COMPUTE BY.
JZ0PA	`The query has been cancelled and the same response discarded.` *Description:* A cancel was probably issued by another statement on the connection. *Action:* Check the chain of SQL exceptions and warnings on this and other statements to determine the cause.

SQLState	Message/Description/Action
JZ0PB	`The server does not support a requested operation.` *Description:* When jConnect creates a connection with a server, it informs the server of capabilities it wants supported and the server informs jConnect of the capabilities that it supports. This error message is sent when an application requests an operation that was denied in the original capabilities negotiation. For example, if the database does not support precompilation of dynamic SQL statements, and your code invokes SybConnection.prepareStatement(*sql_stmt*, *dynamic*), and *dynamic* is set to “true,” jConnect will generate this message. *Action:* Modify your code so that it does not request an unsupported capability.
JZ0R0	`Result set has already been closed.` *Description:* The ResultSet.close() method has already been called on the result set object; you cannot use the result set for anything else. *Action:* Fix the code so that ResultSet object references are nulled whenever a result set is closed.
JZ0R1	`Result set is IDLE as you are not currently accessing a row.` *Description:* The application has called one of the ResultSet.getXXX column-data retrieval methods, but there is no current row; the application has not called ResultSet.next(), or ResultSet.next() returned “false” to indicate that there is no data. *Action:* Check that rs.next() is set to “true” before calling rs.getXXX.
JZ0R2	`No result set for this query.` *Description:* You used Statement.executeQuery(), but the statement returned no rows. *Action:* Use executeUpdate for statements returning no rows.
JZ0R3	`Column is DEAD. This is an internal error. Please report it to Sybase technical support.` *Action:* See message text.
JZ0R4	`Column does not have a text pointer. It is not a text/image column or the column is NULL.` *Description:* You cannot update a text/image column if it is NULL. A NULL text/image column does not contain a text pointer. *Action:* Make sure that you are not trying to update or get a text pointer to a column that does not support text/image data. Make sure that you are not trying to update a text/image column that is NULL. Insert data first, then make the update.

SQLState	Message/Description/Action
JZ0RM	`refreshRow may not be called after updateRow or deleteRow.` *Description:* After updating a row in the database with SybCursorResult.updateRow(), or deleting it with SybCursorResult.deleteRow(), you used SybCursorResult.refreshRow() to refresh the row from the database. *Action:* Do not attempt to refresh a row after updating it or deleting it from the database.
JZ0S0	`Statement state machine: Statement is BUSY.` *Description:* The only time this error is raised is from the Statement.setCursorname() method, if the application is trying to set the cursor name when the statement is already in use and has non-cursor results that need to be read. *Action:* Set the cursor name on a statement before you execute any queries on it, or call Statement.cancel() before setting the cursor name, to make sure that the statement isn't busy.
JZ0S1	`Statement state machine: Trying to FETCH on IDLE statement.` *Description:* An internal error occurred on the statement. *Action:* Close the statement and open another one.
JZ0S2	`Statement object has already been closed.` *Description:* The Statement.close() method has already been called on the statement object; you cannot use the statement for anything else. *Action:* Fix the application so that statement object references are nulled out whenever a statement is closed.
JZ0S3	`The inherited method _____ cannot be used in this subclass.` *Description:* PreparedStatement does not support executeQuery(String), **executeUpdate(String)**, or execute(String). *Action:* If you want to pass a query string, use Statement, not PreparedStatement.
JZ0S4	`Cannot execute an empty (zero-length) query.` *Action:* Do not execute an empty query (““).
JZ0S8	`An escape sequence in a SQL Query was malformed: ‘_____’.` *Description:* This error results from bad escape syntax. *Action:* Check JDBC documentation for correct syntax.
JZ0S9	`Cannot execute an empty (zero-length) query.` *Action:* Do not execute an empty query (““).
JZ0SA	`Prepared Statement: Input parameter not set, index: _____.` *Action:* Make sure that each input parameter has a value.

SQLState	Message/Description/Action
JZ0SB	`Parameter index out of range: _____.` *Description:* You have attempted to get, set, or register a parameter that goes beyond the maximum number of parameters. *Action:* Check the number of parameters in your query.
JZ0SC	`Callable Statement: attempt to set the return status as an InParameter.` *Description:* You have prepared a call to a stored procedure that returns a status, but you are trying to set parameter 1, which is the return status. *Action:* Parameters that you can set start at 2 with this type of call.
JZ0SD	`No registered parameter found for output parameter.` *Description:* This indicates an application logic error. You attempted to call getXXX() or wasNull() on a parameter, but you have not read any parameters yet, or there are no output parameters. *Action:* Check to make sure that the application has registered output parameters on the CallableStatement, that the statement has been executed, and that the output parameters were read.
JZ0SE	`Invalid object type specified for setObject().` *Description:* Illegal type argument passed to PreparedStatement.setObject. *Action:* Check the JDBC documentation. The argument must be a constant from java.sql.Types.
JZ0SF	`No Parameters expected. Has query been sent?` *Description:* You tried to set a parameter on a statement with no parameters. *Action:* Make sure the query has been sent before you set the parameters.

SQLState	Message/Description/Action

JZ0SG	`An RPC did not return as many output parameters as the application had registered for it.`
	
	Description: This error occurs if you call CallableStatement.registerOutParam() for more parameters than you declared as "OUTPUT" parameters in the stored procedure. See "RPC Returns Fewer Output Parameters Than Registered" on page 103 for more information.
	
	Action: Check your stored procedures and registerOutParameter calls. Make sure that you have declared all of the appropriate parameters as "OUTPUT." Look at the line of code that reads:
	
	`create procedure yourproc (@p1 int OUTPUT, ...`
	
	Note If you receive this error while using Adaptive Server Anywhere (previously known as SQL Anywhere), upgrade to Adaptive Server Anywhere version 5.5.04.

JZ0SH	`A static function escape was used, but the metadata accessor information was not found on this server.` *Action:* Install metadata accessor information before using static function escapes.
JZ0SI	`A static function escape _____ was used which is not supported by this server.` *Action:* Do not use this escape.
JZ0SJ	`Metadata accessor information was not found on this database.` *Action:* Install metadata information before making metadata calls.
JZ0SM	`Unsupported SQL type _____.` *Action:* Do not use Types.NULL, Types.OTHER, or PreparedStatement.setObject(null)
JZ0SN	`setMaxFieldSize: field size cannot be negative.` *Action:* Use a positive value or zero (unlimited) when calling setMaxFieldSize.
JZ0T2	`Listener thread read error.` *Action:* Check your network communications.
JZ0T3	`Read operation timed out.` *Description:* The time allotted to read the response to a query was exceeded. *Action:* Increase the time-out period by calling Statement.setQueryTimeout().
JZ0T4	`Write operation timed out. Timeout in milliseconds: _____.` *Description:* The time allotted to send a request was exceeded. *Action:* Increase the timeout period by calling Statement.setQueryTimeout().

SQLState	Message/Description/Action
JZ0T5	`Cache used to store responses is full.` *Action:* Use default or larger value for the STREAM_CACHE_SIZE connection property.
JZ0T6	`Error reading tunneled TDS URL.` *Description:* The tunneled protocol failed while reading the URL header. *Action:* Check the URL you defined for the connection.
JZ0T7	`Listener thread read error -- caught ThreadDeath. Check network connection.` *Action:* Check the network connections and try to run the application again. If the threads continue to be aborted, contact Sybase Technical Support.
JZ0T9	`Request to send not synchronized. Please report this error to Sybase Technical Support.` *Action:* See message text.
JZ0TC	`Attempted conversion between an illegal pair of types.` *Description:* Conversion between a Java type and a SQL type failed. *Action:* Check the requested type conversion to make sure it is supported in the JDBC specification.
JZ0TE	`Attempted conversion between an illegal pair of types. Valid database types are: '_________.'` *Description:* The database column datatype and the datatype requested in theResultSet.getXXX() call are not implicitly convertible. *Action:* Use one of the valid datatypes listed in the error message.
JZ0US	`The SybSocketFactory connection property was set, and the PROXY connection property was set to the URL of a servlet. The jConnect driver does not support this combination. If you want to send secure HTTP from an applet running within a browser, use a proxy URL beginning with "https://".` *Action:* See message text.
JZ0CX	`____________ is an unrecognized transaction coordinator type.` *Description:* The metadata information indicates that the server supports distributed transactions, but jConnect does not support the protocol being used. *Action:* Verify that you have installed the latest metadata scripts. If the error persists, contact Sybase Technical Support.

SQLState	Message/Description/Action
JZ0XS	`The server does not support XA-style transactions. Please verify that the transaction feature is enabled and licensed on this server.` *Description:* The server to which jConnect attempted a connection does not support distributed transactions. *Action:* Do not use XADataSource with this server, or upgrade/configure the server for distributed transactions.
JZ0XU	`Current user does not have permission to do XA-style transactions. Be sure user has _______ role.` *Description:* The user connected to the database is not authorized to conduct distributed transactions. Most likely because they do not have the proper role (shown in the blank). *Action:* Grant the user the role shown in the error message, or have another user with that role conduct the transaction.
S0022	`Invalid column name '_____'.` *Description:* You attempted to reference a column by name and there is no column with that name. *Action:* Check the spelling of the column name.
ZZ00A	`The method _____ has not been completed and should not be called.` *Description:* You attempted to use a method that is not implemented. *Action:* Check the release bulletin that came with your version of jConnect for further information. You can also check the jConnect Web page at http://www.sybase.com to see whether a more recent version of jConnect implements the method. If not, do not use the method.

APPENDIX B jConnect Sample Programs

This appendix is a guide to Sybase jConnect sample programs.

It contains the following sections:

Running IsqlApp

IsqlApp allows you to issue **isql** commands from the command line and run jConnect sample programs.

The syntax for **IsqlApp** is:

```
IsqlApp [-U username] [-P password]
 [-S servername]
 [-G gateway]
 [-p {http|https}]
 [-D debug-class-list]
 [-v]
 [-I input-command-file]
 [-c command_terminator]
 [-C charset] [-L language]
 [-T sessionID]
 [-V <version {2,3,4,5}>]
```

Parameter	Description
-U	The login ID with which you want to connect to a server.
-P	The password for the specified login ID.
-S	The name of the server to which you want to connect.
-G	Gateway address. For the HTTP protocol, the URL is: http://*host:port*. To use the HTTPS protocol that supports encryption, the URL is https://*host*:*port*/*servlet_alias*.
-p	Specifies whether you want to use the HTTP protocol or the HTTPS protocol that supports encryption
-D	Turns on debugging for all classes or for just the ones you specify, separated by a comma. For example, *-D ALL* displays debugging output for all classes. *-D SybConnection, Tds* displays debugging output only for the SybConnection and Tds classes.
-v	Turns on verbose output for display or printing.
-I	Causes IsqlApp to take commands from a file instead of the keyboard. After the parameter, you specify the name of the file to use for the IsqlApp input. The file must contain command terminators (“go” by default).
-c	Lets you specify a keyword (for example, “go”) that, when entered on a line by itself, terminates the command. This lets you enter multiline commands before using the terminator keyword. If you do not specify a command terminator, each new line terminates a command.

Parameter	Description
-C	Specifies the character set for strings passed through TDS. If you don't specify a character set, IsqlApp uses the server's default charset.
-L	The language in which to display error messages returned from the server and for jConnect messages.
-T	When this parameter is set, jConnect assumes that an application is trying to resume communication on an existing TDS session held open by the TDS-tunnelling gateway. jConnect skips the login negotiations and forwards all requests from the application to the specified session ID.
-V	Enables the use version-specific characteristics. See "JCONNECT_VERSION Connection Property" on page 9.

Note You must enter a space after each option flag.

To obtain a full description of the command line options, enter:

```
java IsqlApp -help
```

The following example shows how to connect to a database on a host named "myserver" through port "3756" and run an **isql** script named "myscript":

```
java IsqlApp -U sa -P sapassword
 -S jdbc:sybase:Tds:myserver:3756
 -I $JDBC_HOME/sp/myscript -c run
```

Note An applet that provides GUI access to **isql** commands is available as:

For jConnect 4.x:
$JDBC_HOME/sample/gateway.html (UNIX)
%JDBC_HOME%\sample\gateway.html (Windows)

For jConnect 5.x:
$JDBC_HOME/sample2/gateway.html (UNIX)
%JDBC_HOME%\sample2\gateway.html (Windows)

Running jConnect Sample Programs and Code

jConnect includes several sample programs to illustrate many of the topics covered in this chapter, and to help you understand how jConnect works with various JDBC classes and methods. In addition, this section includes a sample code fragment for your reference.

Sample Applications

When you install jConnect, you can also the install sample programs. These samples include the source code so that you can review how jConnect implements various JDBC classes and methods. See the *jConnect for JDBC Installation Guide* for complete instructions for installing the sample programs.

Note The jConnect sample programs are intended for demonstration purposes only.

The sample programs are installed in the *sample* subdirectory (jConnect 4.x) or *sample2* subdirectory (jConnect 5.x) under your jConnect installation directory. The file *index.html* in the *sample* or *sample2* subdirectory contains a complete list of the samples that are available along with a description of each sample. *index.html* also lets you view and run the sample programs as applets.

Running the Sample Applets

Using your Web browser, you can run some of the sample programs as applets. This enables you to view the source code while viewing the output results.

To run the samples as applets, you need to start the Cascade gateway. See “Using the Cascade Gateway” on page 131.

Use your Web browser to open *index.html*:

For jConnect 4.x, enter:

http://localhost:8000/sample/index.html

For jConnect 5.x, enter:

http://localhost:8000/sample2/index.html

Running the Sample Programs with Adaptive Server Anywhere

All of the sample programs are compatible with Adaptive Server, but only a limited number are compatible with Adaptive Server Anywhere. Refer to *index.html* in the *sample* or *sample2* subdirectory for a current list of the sample programs that are compatible with Adaptive Server Anywhere.

To run the sample programs that are available for Adaptive Server Anywhere, you must install the *pubs2_any.sql* script on your Adaptive Server Anywhere server. This script is located in the *sample* (jConnect 4.1) *or sample2* (jConnect 5.0) subdirectory.

For Windows, go to DOS command window and enter:

```
java IsqlApp -U dba -P password
 -S jdbc:sybase:Tds:[hostname]:[port]
 -I %JDBC_HOME%\sample\pubs2_any.sql -c go
```

For UNIX, enter:

```
java IsqlApp -U dba -P password
 -S jdbc:sybase:Tds:[hostname]:[port]
 -I $JDBC_HOME/sample/pubs2_any.sql -c go
```

Sample Code

The following sample code illustrates how to invoke the jConnect driver, make a connection, issue a SQL statement, and process results.

```
import java.io.*;
 import java.sql.*;

   public class SampleCode
   {
       public static void main(String args[])
       {
           try
           {
           /*
            * Open the connection. May throw a SQLException.
            */
             Connection con = DriverManager.getConnection(
                 "jdbc:sybase:Tds:myserver:3767", "sa", "");
           /*
            * Create a statement object, the container for the SQL
            * statement. May throw a SQLException.
            */
            Statement stmt = con.createStatement();
         /*
          * Create a result set object by executing the query.
          * May throw a SQLException.
          */
            ResultSet rs = stmt.executeQuery("Select 1");
         /*
          * Process the result set.
          */

            if (rs.next())
            {
               int value = rs.getInt(1);
               System.out.println("Fetched value " + value);
            }
        }
    /*
     * Exception handling.
     */
        catch (SQLException sqe)
        {
            System.out.println("Unexpected exception : " +
                        sqe.toString() + ", sqlstate = " +
                            sqe.getSQLState());
            System.exit(1);
        }
        System.exit(0);
    }
}
```

Index