WebSphere Connection Pooling

by

Deb Erickson
Shawn Lauzon
Melissa Modjeski

Note: Before using this information and the product it supports, read the information in "Notices" on page 78.

First Edition (August 2001)

This edition applies to Version 4.0 of IBM WebSphere Application Server and to all subsequent releases and modifications until otherwise indicated in new editions.

Table of contents

1. Introduction

Each time a resource attempts to access a database, it must connect to that database. A database connection incurs overhead -- it requires resources to create the connection, maintain it, and then release it when it is no longer required. The total database overhead for an application is particularly high for Web-based applications because Web users connect and disconnect more frequently. In addition, user interactions are typically shorter, because of the nature of the Internet. Often, more effort is spent connecting and disconnecting than is spent during the interactions themselves. Further, because Internet requests can arrive from virtually anywhere, usage volumes can be large and difficult to predict.

IBM WebSphere Application Server enables administrators to establish a pool of database connections that can be shared by applications on an application server to address these overhead problems. Connection pooling spreads the connection overhead across several user requests, thereby conserving resources for future requests.

Benefits of WebSphere connection pooling
Connection pooling can improve the response time of any application that requires connections, especially Web-based applications. When a user makes a request over the Web to a resource, the resource accesses a datasource. Most user requests do not incur the overhead of creating a new connection, because the datasource might locate and use an existing connection from the pool of connections. When the request is satisfied and the response is returned to the user, the resource returns the connection to the connection pool for reuse. Again, the overhead of a disconnect is avoided. Each user request incurs a fraction of the cost of connection or disconnecting. After the initial resources are used to produce the connections in the pool, additional overhead is insignificant because the existing connections are reused.

Caching of prepared statements is another mechanism by which WebSphere connection pooling improves Web-based application response times. A cache of previously prepared statements is available on a connection. When a new prepared statement is requested on a connection, the cached prepared statement is returned if available. This caching reduces the number of costly prepared statements created, which improves response times. The cache is useful for applications that tend to prepare the same statement time and again.

In addition to improved response times, WebSphere connection pooling provides a layer of abstraction from the database which can buffer the client application and make different databases appear to work in the same manner to an application. This buffering makes it easier to switch application databases, because the application code does not have to deal with common vendor-specific SQLExceptions but, rather, with a WebSphere connection pooling exception.

Who should use WebSphere connection pooling
WebSphere connection pooling should be used in an application that meets any of the following criteria:

- It cannot tolerate the overhead of obtaining and releasing connections whenever a connection is used

- It requires JTA transactions within WebSphere
- It does not manage the pooling of its own connections
- It does not manage the specifics of creating a connection, such as the database name, user name, or password

What's New In Version 4.0?

- *Informix Dynamic Server 2000 support*

In Version 4.0, support has been added for Informix Dynamic Server 2000 (IDS 9.31). Informix Dynamic Server 2000 connection pooling support requires the Informix type-4 JDBC resource (JDBC driver package) for data access.

- *Pure JDBC 2.0 Optional Package datasource model*

In previous versions of WebSphere Application Server, the DriverManager class was used to obtain connections in single-phase commit protocol scenarios (non-JTA enabled datasources). This was necessary because many JDBC resources did not support the JDBC 2.0 Optional Package API. As these JDBC resources started providing datasources for both one-phase and two-phase commit protocols, it enabled WebSphere connection pooling to move to a complete datasource model. This means that all connections are obtained from a DataSource object. This, in turn, enables the user to use the vendor-specific datasource properties that were unavailable in earlier versions of WebSphere Application Server.

- *Added support for vendor-specific datasource properties*

Many JDBC resources have additional properties on a datasource that are not JDBC 2.0 standard properties. In previous versions of WebSphere connection pooling, these properties could only be set in the datasources.xml file and only for a subset of the supported backends. In Version 4.0, these vendor-specific properties are now configurable on the datasource properties pane in the WebSphere Administration Console.

- *Single method to create a datasource with DataSourceFactory*

The getDataSource method on the com.ibm.websphere.advanced.cm.factory.DataSourceFactory class replaces the now deprecated createJDBCDataSource and createJTADataSource methods. This new method accepts a java.util.Properties object instead of the WebSphere proprietary class com.ibm.websphere.advanced.cm.factory.Attributes. This change is important to applications that create their own datasources on demand.

- *Improved stale connection recovery*

Stale connections are those connections which for some reason can no longer be used. This can happen, for example, if the database server is momentarily shut down or if the network is experiencing problems. In all of these cases, the connections are no longer usable by the application and the connection pool needs to be flushed and rebuilt. This type of support was added in Version 3.5.2, and is improved in Version 4.0. More vendor-specific error codes have been added to the mapping that results in a StaleConnectionException. In addition, whenever a StaleConnectionException is thrown by the application server, the entire pool of connections is destroyed.

- *Default user name and password*

In previous versions of WebSphere connection pooling, datasources did not allow for default user name and password properties. This required the application code to provide a user name and password for those databases that did not support the getConnection method without username and password as parameters (getConnection()). In Version 4.0, the datasource object has user and password properties that can be specified on the datasource properties panel in the Administrative Console. These defaults are used when getConnection() with no user name and password is called. If a user name and password are specified on getConnection(user, password), these values override the defaults.

Document conventions

The remainder of this document discusses the details of the WebSphere connection pooling programming model and implementation. In most cases, WebSphere connection pooling works the same for both Advanced Edition for Multiplatforms and Advanced Single Server Edition (formerly Standard Edition), except for the graphical interfaces to the administrative console. All references to the user interface in this document will be specific to the administrative console in WebSphere Advanced Edition for Multiplatforms (AE).

Features or modifications added in Version V4.0 are marked as follows:

Version *4.0* - New in Version 4.0

2. JDBC 2.0 specifications

WebSphere connection pooling is WebSphere's implementation of the JDBC 2.0 Optional Package API. Therefore, it is beneficial to have a basic understanding of the JDBC 2.0 Core API and the JDBC 2.0 Optional Package API, which make up the JDBC 2.0 API, before delving into the specifics of WebSphere connection pooling.

The JDBC API was designed to provide a Java application programming interface to data access of relational databases. JDBC 1.0 provided the basic functionality for this access. JDBC 2.0 API extends the JDBC 1.0 API to provide additional functionality, simplify coding, and increase performance.

JDBC 2.0 is divided into the JDBC 2.0 Core API, which is a superset of JDBC 1.0, and the JDBC 2.0 Optional Package API, which provides support for JNDI (Java Naming and Directory Interface), connection pooling, and JTA (Java Transaction API).

JDBC 2.0 Core API

WebSphere applications can access a relational database directly through a JDBC driver using the JDBC 2.0 Core API. Accessing a relational database in this manner is done through three basic steps:

- Establish a connection through the DriverManager class
- Send SQL queries or updates to the DBMS

- Process the results

The following code fragment shows a simple example of obtaining and using a connection directly through a JDBC driver:

```
try {
  //establish a connection through the DriverManager
  Class.forName("COM.ibm.db2.jdbc.app.DB2Driver");
  String url = "jdbc:db2:sample";
  String username = "dbuser";
  String password = "passwd";
  java.sql.Connection conn =
     java.sql.DriverManager.getConnection(url,username,
                                          password);

  //query the database
  java.sql.Statement stmt = conn.createStatement();
  java.sql.ResultSet rs =
      stmt.executeQuery("SELECT EMPNO, FIRSTNME, LASTNAME
                         FROM EMPLOYEE");

  //process the results
  while (rs.next()) {
     String empno = rs.getString("EMPNO");
     String firstnme = rs.getString("FIRSTNME");
     String lastname = rs.getString("LASTNAME");
     //work with results
  }
} catch (java.sql.SQLException sqle) {
     //handle the SQLException
} finally {
   try {
      if(rs != null) rs.close();
   } catch (java.sql.SQLException sqle) {
      //can ignore
   }
   try {
      if(stmt != null) stmt.close();
   } catch (java.sql.SQLException sqle) {
      //can ignore
   }
   try {
       if(conn != null) conn.close();
   } catch (java.sql.SQLException sqle) {
       //can ignore
   }
}
```

In this example, the first action is to establish a connection to the database. This is done by loading and registering the JDBC driver and then requesting a connection from DriverManager. DriverManager is a JDBC 1.0 class and is the basic service for managing a set of JDBC drivers. It is necessary to load the driver class before the call to getConnection(), because DriverManager

can establish a connection only to a driver that has registered with it. Loading the driver class also registers it with DriverManager.

After a connection has been obtained, the database is queried by creating a statement and executing a query on that statement. The query results are put into a ResultSet object.

Lastly, the results are processed by stepping through the result set and pulling the data from each record retrieved.

Important: In this example, only a single connection is obtained. This connection does not belong to a pool of connections and is not managed by WebSphere connection pooling. It is the responsibility of the application to manage the use of this connection.

The JDBC 2.0 Core API is defined by the classes and interfaces in the java.sql.* package. The following tables summarize the classes and interfaces included in the JDBC 2.0 Core API:

java.sql.* class summary

Class name	Description
Date	A thin wrapper around a millisecond value that allows JDBC to identify this as an SQL DATE.
DriverManager	The basic service for managing a set of JDBC drivers.
DriverPropertyInfo	Driver properties for making a connection.
SQLPermission	The permission for which SecurityManager checks when code that is running in an applet calls one of the setLogWriter methods.
Time	A thin wrapper around java.util.Date that allows JDBC to identify this as an SQL TIME value.
Timestamp	A thin wrapper around java.util.Date that allows the JDBC API to identify this as an SQL TIMESTAMP value.
Types	The class that defines the constants that are used to identify generic SQL types, called JDBC types.

java.sql.* interface summary

Interface name	Description
Array	The mapping in the Java programming language for the SQL type ARRAY.
Blob	The representation (mapping) in the Java programming language of an SQL BLOB value.
CallableStatement	The interface used to execute SQL stored procedures.
Clob	The mapping in the Java programming language for the SQL CLOB type.
Connection	A connection (session) with a specific database.
DatabaseMetaData	Comprehensive information about the database as a whole.
Driver	The interface that every driver class must implement
PreparedStatement	An object that represents a precompiled SQL statement. .
Ref	The mapping in the Java programming language of an SQL REF

	value, which is a reference to an SQL structured type value in the database.
ResultSet	A table of data representing a database result set, which is usually generated by executing a statement that queries the database.
ResultSetMetaData	An object that can be used to get information about the types and properties of the columns in a ResultSet object.
SQLData	The interface used for the custom mapping of SQL user-defined types.
SQLInput	An input stream that contains a stream of values representing an instance of an SQL structured or distinct type.
SQLOutput	The output stream for writing the attributes of a user-defined type back to the database.
Statement	The object used for executing a static SQL statement and obtaining the results produced by it.
Struct	The standard mapping in the Java programming language for an SQL structured type.

For more information on the details of JDBC 2.0 Core API, refer to the JDBC 2.0 Core API specification at http://java.sun.com.

JDBC 2.0 Optional Package API

The JDBC 2.0 Optional Package API is an extension of the JDBC 2.0 Core API. The optional extension supports integration with the new Java standards, including JNDI (Java Naming and Directory Interface), JTA (Java Transaction API), and EJB (Enterprise JavaBeans) as well as providing support for connection pooling and the JavaBeans specification.

In JDBC 1.0 and the JDBC 2.0 Core API, DriverManager was used exclusively for obtaining a connection to a database. The database URL, username, and password were used in the getConnection request. In the JDBC 2.0 Optional Package API, the datasource provides a new means of obtaining connections to a database. The benefit of using the DataSource model is the creation and management of the connection factory is centralized. Applications do not need to have specific information such as the database name, user name, or password in order to obtain a connection to the database.

The steps for obtaining and using a connection in the JDBC 2.0 Optional Package API paradigm differ slightly from the JDBC 2.0 Core API example. Using the extensions, the following steps need to occur:

- Retrieve a DataSource object from the JNDI naming service
- Obtain a Connection object from the datasource
- Send SQL queries or updates to the DBMS
- Process the results

The following code fragment shows how to obtain and use a connection in the JDBC 2.0 Optional Package API through a datasource:

```
try {
```

```
   //Retrieve a DataSource through the JNDI Naming Service

   java.util.Properties parms = new java.util.Properties();
   parms.setProperty(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");

   //Create the Initial Naming Context
   javax.naming.Context ctx = new
      javax.naming.InitialContext(parms);

   //Lookup through the naming service to retrieve a
   //DataSource object
   //In this example, SampleDB is the datasource
   javax.sql.DataSource ds =
      (javax.sql.DataSource)
          ctx.lookup("java:comp/env/jdbc/SampleDB");

   //Obtain a Connection from the DataSource
  java.sql.Connection conn =
      ds.getConnection();

   //query the database
   java.sql.Statement stmt = conn.createStatement();
   java.sql.ResultSet rs =
      stmt.executeQuery("SELECT EMPNO, FIRSTNME, LASTNAME
                          FROM EMPLOYEE");

   //process the results
   while (rs.next()) {
      String empno = rs.getString("EMPNO");
      String firstnme = rs.getString("FIRSTNME");
      String lastname = rs.getString("LASTNAME");
     // work with results
   }
} catch (java.sql.SQLException sqle) {
   //handle SQLException
} finally {
    try {
       if (rs != null) rs.close();
    } catch (java.sql.SQLException sqle) {
       //can ignore
     }
    try {
       if (stmt != null) stmt.close();
    } catch (java.sql.SQLException sqle) {
       //can ignore
    }
    try {
       if (conn != null) conn.close();
    } catch (java.sql.SQLException sqle) {
       //can ignore
    }
}
```

In this example, the first action is to retrieve a DataSource object from the JNDI name space. This is accomplished by creating a Properties object of parameters used to set up an InitialContext object. Once a context is obtained, a lookup on the context is performed to find the specific datasource necessary, in this case, the SampleDB datasource. In this example, it is assumed the datasource has already been created and bound into JNDI by the WebSphere administrator. Creation of the datasource will be covered in Section 6 of this paper.

After a DataSource object is obtained, the application code calls getConnection() on the datasource to get a Connection object. Once the connection is acquired, the querying and processing steps are the same as for the JDBC 1.0 / JDBC 2.0 Core API example.

As opposed to the JDBC 2.0 Core API example, the connection obtained in this example from the datasource is a pooled connection. This means that the Connection object is obtained from a pool of connections managed by WebSphere connection pooling.

Version *4.0*

The initial context factory name has changed from com.ibm.ejs.ns.jndi.CNInitialContextFactory (Version 3.5) to com.ibm.websphere.naming.WsnInitialContextFactory (Version 4.0).

Version *4.0*

In Version 4.0, the JNDI lookup string has changed. In Version 3.5, the lookup would be the following:

```
javax.sql.DataSource ds =
          (javax.sql.DataSource)
              ctx.lookup("jdbc/SampleDB");
```

In Version 4.0, the lookup is:

```
javax.sql.DataSource ds =
          (javax.sql.DataSource)
              ctx.lookup("java:comp/env/jdbc/SampleDB");
```

Both lookups work in Version 4.0, but all new applications should be written with the Version 4.0 syntax, which is the J2EE 1.2-compliant lookup method. (SampleDB is the name of the datasource being retrieved.)

The JDBC 2.0 Optional Package API is defined by the classes and interfaces in the javax.sql.* package. The following tables summarize the classes and interfaces included in the JDBC 2.0 Optional Package API:

javax.sql.* class summary

Class name	Description
ConnectionEvent	Provides information about the source of a connection related event.
RowSetEvent	Generated when something important happens in the life of a rowset, like when a column value changes.

javax.sql.* interface summary

Interface name	Description
ConnectionEventListener	Registers to receive events generated by PooledConnection.
ConnectionPoolDataSource	Factory for PooledConnection objects.
DataSource	Factory for Connection objects.
PooledConnection	Provides hooks for connection pool management.
RowSet	Adds support to the JDBC API for the JavaBeans component model.
RowSetInternal	A RowSet object presents itself to a reader or writer as an instance of RowSetInternal.
RowSetListener	Implemented by a component that wants to be notified when a significant event happens in the life of a RowSet.
RowSetMetaData	Extends ResultSetMetaData with methods that allow a metadata object to be initialized.
RowSetReader	Implementing this interface enables registration with a RowSet object that supports the reader/writer paradigm.
RowSetWriter	Implementing this interface enables registration with a RowSet object that supports the reader/writer paradigm.
XAConnection	Provides support for distributed transactions.
XADataSource	Factory for XAConnection objects.

For more information on the details of JDBC 2.0 Optional Package API, refer to the JDBC 2.0 Optional Package API Specification at http://java.sun.com.

3. Connection pooling programming model and advanced features

As stated earlier, WebSphere connection pooling is the implementation of the JDBC 2.0 Optional Package API specification. Therefore, the connection pooling programming model is as specified in the JDBC 2.0 Core and JDBC 2.0 Optional Package API specifications. This means that applications obtaining their connections through a datasource created in WebSphere Application Server can benefit from JDBC 2.0 features such as pooling of connections and JTA-enabled connections. In addition, WebSphere connection pooling provides additional features that enable administrators to tune the pool for best performance and provide applications with WebSphere exceptions that enable programmers to write applications without knowledge of common vendor-specific SQLExceptions. Not all vendor-specific SQLExceptions are mapped to WebSphere exceptions; applications must still be coded to deal with vendor-specific SQLExceptions. However, the most common, recoverable exceptions are mapped to WebSphere exceptions.

WebSphere connection pooling API

WebSphere provides a public API to enable users to configure a WebSphere datasource. In most cases, use of this API is not necessary, because the datasource is configured by the administrator in the WebSphere Administrative Console. However, in the case that an application needs to

create a WebSphere datasource object on demand, this API has been provided. The complete API specification can be seen by viewing javadoc output for the class com.ibm.websphere.advanced.cm.factory.DataSourceFactory. In this section, only those methods necessary to create a datasource and bind it into the JNDI namespace through application code are detailed.

To create a datasource on demand in an application, the application must do the following:
1. Create a Properties object with datasource properties
2. Obtain a datasource from the factory
3. Bind the datasource into JNDI

The following code fragment shows how an application would create a datasource and bind it into JNDI:

```
import com.ibm.websphere.advanced.cm.factory.DataSourceFactory;
//code removed ...
try {

  //Create a properties file for the DataSource
   java.util.Properties prop = new java.util.Properties();
   prop.put(DataSourceFactory.NAME, "SampleDB");
   prop.put(DataSourceFactory.DATASOURCE_CLASS_NAME,
      "COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource");
   prop.put(DataSourceFactory.DESCRIPTION, "My sample
       datasource");
    prop.put("databaseName", "sample");

    //Obtain a DataSource from the factory
   DataSource ds = DataSourceFactory.getDataSource(prop);

   //Bind the DataSource into JNDI
   DataSourceFactory.bindDataSource(ds);
} catch (ClassNotFoundException cnfe) {
   //check the class path for all necessary classes
} catch (CMFactoryException cmfe) {
   //Example of exception: incorrect properties
} catch (NamingException ne) {
   //Example of exception:
   //datasource by this name may already exist
}
```

In this example, the first step is to create a DataSourceFactory. This factory is the means by which the application creates a datasource in WebSphere connection pooling. To create a datasource for binding into JNDI, the application must first create a Properties object to hold the datasource configuration properties. The only properties required on the datasource from a WebSphere perspective are:

- NAME -The name of the datasource. This is used to identify the datasource when it is bound into JNDI.
- DATASOURCE_CLASS_NAME - The complete name of the datasource class that can be found in the JDBC resource .zip/.jar file (often referred to as the JDBC driver

package). This datasource class is used to create connections to the database. The class specified here must implement javax.sql.ConnectionPoolDataSource or javax.sql.XADataSource.

However, depending on the datasource class specified in the DATASOURCE_CLASS_NAME property, there may be other vendor-specific properties required. In this example, the databaseName property is also required, because DB2ConnectionPoolDataSource is being used. For more information on these vendor-specific properties, see Appendix D for the list of required properties for all of the supported datasource or see the vendor's documentation for the complete list of properties supported for a datasource.

After a Properties object is created, the application can create a new DataSource object by calling getDataSource() on the factory, passing the Properties object in as a parameter. This creates an object of type DataSource, but this datasource is not bound into JNDI yet. Therefore, other applications that need to use the same datasource, and hence the same connection pool, cannot do so because only this instance of the application has the datasource. Binding a datasource into JNDI is done by calling bindDataSource() on the factory, as shown in this example. At this point, other applications can share the same datasource by retrieving it from JNDI using the name specified on creation. The datasource must be bound into JNDI before use, because it is the retrieval from JNDI that creates the functional datasource.

See the javadoc output for the DataSourceFactory class for additional information. All other APIs specific to WebSphere connection pooling are not public APIs. For applications that use a WebSphere datasource, the JDBC 2.0 Core and JDBC 2.0 Optional Package APIs should be followed.

WebSphere datasources

The datasource obtained through WebSphere is a datasource that implements the JDBC 2.0 Optional Package API. It provides pooling of connections and, depending on the vendor-specific datasource selected, may provide connections capable of participating in two-phase commit protocol transactions (JTA-enabled). On a WebSphere datasource, there are properties specific to WebSphere connection pooling that are not defined on a javax.sql.DataSource object. The following are the properties specific to a WebSphere datasource object; they can be found in com.ibm.websphere.advanced.cm.factory.DataSourceFactory:

- CONN_TIMEOUT - Maximum number of seconds an application waits for a connection from the pool before timing out and throwing com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException to the application. The default value is 180 seconds (3 minutes). Any non-negative integer is a valid value. Setting this value to 0 disables the connection timeout. This value can also be changed by calling setLoginTimeout() on the datasource. (ConnectionWaitTimeoutException is discussed in later in this paper.) If setLoginTimeout is called on the datasource, this sets the timeout for all applications that are using that datasource. For this reason, it is recommended that setLoginTimeout not be used. Instead, the connection timeout property should be set on the datasource during configuration.

- DATASOURCE_CLASS_NAME - Specifies the vendor's DataSource object through which WebSphere connection pooling connects to the database. For example, DB2 could be COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource. The datasource specified must implement javax.sql.ConnectionPoolDataSource or javax.sql.XADataSource. If a class that implements javax.sql.XADataSource is specified, the datasource is JTA-enabled. Specifying this property is required.

- DISABLE_AUTO_CONN_CLEANUP - Specifies whether or not the connection is closed at the end of a transaction. The default is *false*, which indicates that when a transaction is completed, WebSphere connection pooling closes the connection and all associated resources and returns the connection to the pool. This means that any use of the connection after the transaction has ended results in a StaleConnectionException, because the connection has been closed and returned to the pool. This mechanism ensures that connections are not held indefinitely by the application. If the value is set to *true*, the connection is not returned to the pool at the end of a transaction. In this case, the application must return the connection to the pool by explicitly calling close(). If the application does not close the connection, the pool will run out of connections for other applications to use. This is different from orphaning connections, because connections in a transaction cannot be orphaned.

- ERROR_MAP - This property should not be used by most applications. It is to be used only when it is necessary to map an additional SQLException to a WebSphere exception. Before using this property, contact IBM support for additional information.

- IDLE_TIMEOUT - The number of seconds that a connection can remain free in the pool before the connection is removed from the pool and closed. The default value is 1800 seconds (30 minutes). Any non-negative integer is a valid value. Connections need to be idled out of the pool because keeping connections open to the database can cause memory problems with the database in some cases. Not all connections are idled out of the pool, even if they are older than the IDLE_TIMEOUT number of seconds. A connection is not idled if removing this connection would cause the pool to shrink below its minimum value. Setting this value to 0 disables the idle timeout.

- MAX_POOL_SIZE - The maximum number of connections that the connection pool can hold. By default, this value is 10. Any positive integer is a valid value. If this value is set to 0 or less, a connection cannot be obtained from the database and the application is thrown a ConnectionWaitTimeoutException. The maximum pool size can affect the performance of an application. Larger pools require more overhead when demand is high, because more connections are open to the database at peak demand. These connections persist until they are idled out of the pool (see IDLE_TIMEOUT for more information). On the other hand, if the maximum is smaller, there might be longer wait times or possible connection timeout errors during peak times. The database must be able to support the maximum number of connections used by WebSphere Application Server in addition to any load it may have from other sources.

- MIN_POOL_SIZE - The minimum number of connections that the connection pool may hold. By default, this value is 1. Any non-negative integer is a valid value. If the property is set to 0, the pool can shrink to zero connections, which is a valid configuration. The minimum pool size can affect the performance of an application. Smaller pools require less overhead when the demand is low because fewer connections are being held open to the database. On the other hand, when the demand is high, the first applications experience a slow response, because new connections have to be created if all others in the pool are in use. The pool does not start out with the minimum number of connections. As applications request connections, the pool grows up to the minimum number. After the pool has reached the minimum number of connections, it does not shrink beyond this minimum unless an exception occurs that requires the entire pool to be destroyed.

- ORPHAN_TIMEOUT - The number of seconds that an application is allowed to hold an inactive connection. The default is 1800 seconds (30 minutes). Any non-negative integer is a valid value. If there is no activity on an allocated connection for longer than the ORPHAN_TIMEOUT number of seconds, the connection is marked for orphaning. After another ORPHAN_TIMEOUT number of seconds, if the connection still has no activity, the connection is returned to the pool. If the application tries to use the connection again, it is thrown a StaleConnectionException. Connections that are enlisted in a transaction are not orphaned. Setting this value to 0 disables the orphan timeout. (StaleConnectionException is discussed later in this paper.)

- STATEMENT_CACHE_SIZE - The number of cached prepared statements to keep for an entire connection pool. The default value is 100. Any non-negative integer is a valid value. When a statement is cached, it helps performance, because a statement is retrieved from the cache if a matching statement is found, instead of creating a new prepared statement, which is a more costly operation. The statement cache size does not change the programming model, only the performance of the application. The statement cache size is the number of cached statements for the entire pool, not per connection.

In addition to these properties, any property listed for the vendor-specific datasource can be set on a WebSphere datasource. For more information about what is available on a specific vendor's datasource, see the vendor's documentation. See Appendix D for the minimum required properties for each supported JDBC resource's datasource.

User names and passwords

User name and password values have a great impact on how an application obtains connections, the properties of those connections, and how those connections behave. In most cases, the application does not supply a user name and password when calling getConnection() on a datasource. In these cases, it is dependent on the vendor-specific DataSource class used as to what the default user name and password is. In the case of DB2 on Windows NT, for example, it is generally the user name and password under which the requester is operating. Although this

works for very simple applications, most applications require a user name and password for security reasons.

When a user name and password are specified that match the user name and password of a connection already allocated in the current transaction, the new request will share the allocated connection. This means that multiple requesters can share the same connection object to the database as long as they are in the same transaction. Calling close() on the connection object does not return the connection to the free pool until all requesters have called close().

Applications often run into problems when more than one connection is obtained and each has a different user name. When the second connection with a different user name is obtained from the pool, a different connection to the database is returned to the application. This means that there are two physical connections being held to the database by the application, so no connection sharing occurs. If the datasource used to create the connection is not JTA-enabled (a one-phase commit datasource) and there is a global transaction in force, errors occur, because there are two one-phase resources enlisted in the same global transaction. For example, the following code causes an error in the application:

```
//DataSource ds is a non JTA datasource (one-phase commit)
//Global transaction is in force
java.sql.Connection conn = ds.getConnection("db2admin",
                                            "db2admin");
//work with conn

java.sql.Connection conn2 = ds.getConnection("bill",
                                             "myc0nn");
//error will result
```

An error results with this code because the second getConnection() call returns a different connection than the first because of the different user name and password. This means that *two* different one-phase-only connections are being enlisted in the global transaction. This causes an error because only a single one-phase resource may be enlisted in the global transaction.

On the other hand, the following code does not cause an error, even when the datasource is not JTA-enabled, because the two getConnection() calls return the same Connection object.

```
//DataSource ds is a non JTA datasource (one-phase commit)
//Global transaction is in force
java.sql.Connection conn = ds.getConnection("db2admin",
                                            "db2admin");
       //work with conn

java.sql.Connection conn2 = ds.getConnection("db2admin",
                                             "db2admin");
//works because conn and conn2 are the same
//connection to the database
```

There are no errors from this code, because the second getConnection() call returns the same connection as the first. This is known as *connection sharing*, and it happens when multiple

requests for a connection are made with the same user name, same password, and in the same transaction. If close() is called on both conn and conn2, the connection is not freed to the pool until both Connection objects have been closed.

The essence of this discussion is this: If the application is participating in a global transaction, ensure that only a single one-phase connection (a connection from a non-JTA-enabled datasource) participates in the transaction. If more than one connection (either JTA-enabled or not) becomes involved in a transaction in which there is already a one-phase (non-JTA) connection, errors occur. If the datasource is JTA-enabled, it is valid to obtain as many connections in a transaction as desired without errors. If the user name, password, and transaction are all the same, the application can share the same Connection object.

WebSphere exceptions

WebSphere connection pooling monitors specific SQLExceptions thrown from the database. A set of these exceptions are mapped to WebSphere-specific exceptions. WebSphere connection pooling provides these exceptions to ease development by not requiring the developer to know all of the database-specific SQLExceptions that could be thrown for very common occurrences. In addition, monitoring the SQLExceptions enables WebSphere connection pooling and, therefore, the application to recover from common occurrences such as intermittent network or database outages.

ConnectionWaitTimeoutException

This exception (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException) indicates that the application has waited for the connection timeout (CONN_TIMEOUT) number of seconds and has not been returned a connection. This can occur when the pool is at its maximum and all of the connections are in use by other applications for the duration of the wait. In addition, there are no connections currently in use that the application can share, because either the user name and password are different or it is in a different transaction. The following code sample shows how to use ConnectionWaitTimeoutException:

```
java.sql.Connection conn = null;
javax.sql.DataSource ds = null
try {
   //Retrieve a DataSource through the JNDI Naming Service

   java.util.Properties parms = new java.util.Properties();
   setProperty.put(Context.INITIAL_CONTEXT_FACTORY,
      "com.ibm.websphere.naming.WsnInitialContextFactory");

   //Create the Initial Naming Context
   javax.naming.Context ctx = new
      javax.naming.InitialContext(parms);

   //Lookup through the naming service to retrieve a
   //DataSource object
   javax.sql.DataSource ds =
      (javax.sql.DataSource)
          ctx.lookup("java:comp/env/jdbc/SampleDB");
```

```
   conn = ds.getConnection();
   //work on connection
} catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {
      //notify the user that the system could not provide a
      //connection to the database
}  catch (java.sql.SQLException sqle) {
      //deal with exception
}
```

In all cases in which ConnectionWaitTimeoutException is caught, there is very little to do in terms of recovery. It usually doesn't make sense to retry the getConnection() method, because if a longer wait time is required, the connection timeout should be set higher. Therefore, if this exception is caught by the application, the administrator should review the expected usage of the application and tune the connection pool and the database accordingly. Catching this exception is optional, because ConnectionWaitTimeoutException is a subclass of SQLException; any application that catches SQLException automatically catches this exception as well.

StaleConnectionException
This exception (com.ibm.websphere.ce.cm.StaleConnectionException) indicates that the connection currently being held is no longer valid. This can occur for numerous reasons, including the following:

- The application tries to get a connection and fails, as when the database is not started.
- A connection is no longer usable due to a database failure. When an application tries to use a connection it has previously obtained, the connection is no longer valid. In this case, all connections currently in use by an application could get this error when they try to use the connection.
- The application using the connection has already called close() and then tries to use the connection again.
- The connection has been orphaned because the application had not used it in at most two times the orphan timeout; then the application tries to use the orphaned connection.
- The application tries to use a JDBC resource, such as a statement, obtained on a now-stale connection.

Version *4.0*
Note the change of packaging for StaleConnectionException in Version 4.0. In Version 3.5, this class was com.ibm.ejs.cm.portability.StaleConnectionException. This class will is available in Version 4.0 but is deprecated. All new applications should be built using com.ibm.websphere.ce.cm.StaleConnectionException.

Explicitly catching StaleConnectionException is not required in an application. Because applications are already required to catch java.sql.SQLException and StaleConnectionException extends SQLException, StaleConnection Exception is automatically caught in the general catch-block. However, explicitly catching StaleConnectionException makes it possible for an application to recover from bad connections. The most common time for StaleConnectionException to be thrown is the first time that a connection is used, just after it is retrieved. Because connections are pooled, a database failure is not detected until the operation immediately following its retrieval from the pool, which is the first time communication to the

database is attempted. And it is only when a failure is detected that the connection is marked stale. StaleConnectionException occurs less often if each method that accesses the database gets a new connection from the pool. Examining the sequence of events that occur when a database fails to service a JDBC request shows that this occurs because all connections currently handed out to an application are marked stale; the more connections the application has, the more connections can be stale.

Generally when StaleConnectionException is caught, the transaction in which the connection was involved needs to be rolled back and a new transaction begun with a new connection. Details on how to do this can be broken down into three categories:

- Connection in auto-commit mode
- Connection not in auto-commit and transaction begun in same method as database access
- Connection not in auto-commit and transaction begun in different method from database access

Connection in auto-commit mode

By default, any connection obtained from a one-phase datasource (implements javax.sql.ConnectionPoolDataSource) is in auto-commit mode. When in auto-commit mode, each database action is executed and committed in a single database transaction. Servlets often use connections in auto-commit mode, because transaction semantics are not necessary. Enterprise applications do not usually use connections in auto-commit mode. Auto-commit can be explicitly disabled by calling setAutoCommit() on a Connection object.

When StaleConnectionException is caught from a connection in auto-commit mode, recovery is a simple matter of closing all of the associated JDBC resources and retrying the operation with a new connection. An example of this follows:

```
public void myBMPMethod() throws java.rmi.RemoteException
{
   //numOfRetries indicates how many times to retry on a
   //StaleConnectionException

   boolean retry = false;
   int numOfRetries = 0;
   java.sql.Connection conn = null;
   java.sql.Statement stmt = null;

   do {
      try {
         //assumes a datasource already obtained from JNDI
         conn = ds.getConnection();
         conn.setAutoCommit(true);
         stmt = conn.createStatement();
         stmt.execute("INSERT INTO EMPLOYEES VALUES
                        (0101, 'Bill', 'R', 'Smith')");
      } catch(com.ibm.websphere.ce.cm.StaleConnectionException
            sce) {
        if(numOfRetries < 2) {
```

```
            retry = true;
            numOfRetries++;
         } else {
            retry = false;
            //throw error indicating a connection
            //cannot be obtained
         }
      } catch (java.sql.SQLException sqle) {
         //handle other database problem
      } finally {
         //always cleanup JDBC resources
         try {
         if(stmt != null) stmt.close();
         } catch (java.sql.SQLException sqle) {
            //usually can be ignored
         }
         try {
            if(conn != null) conn.close();
         } catch (java.sql.SQLException sqle) {
            //usually can be ignored
         }
      }
   } while (retry);
}
```

In this example, after the connection is retrieved, auto-commit is set to *true*. For clarity, set auto-commit either on or off every time a new connection is retrieved. As a performance enhancement, WebSphere does not call setAutoCommit() on the driver is autoCommit is already on; the same goes for setting autoCommit to *off*. Next, a statement is created and an INSERT is executed, which automatically begins a transaction at the database, executes the statement, and commits it. If no exception occurs, the finally block is executed, which closes the statement and connection, and the method exits. If a StaleConnectionException occurs, retry is set to *true*, the JDBC resources are closed, and the loop is re-executed.

It is often necessary to limit the number of retries on StaleConnectionException, because the cause of the exception might not be transient. Therefore, this example uses a variable to indicate the number of retries allowed before the user is notified that the application could not obtain a valid connection. The number of retries and the time to wait between retries is dependent upon the database. If it is observed that the database experiences problems for 30 seconds before connections can be obtained, it might be worthwhile to retry every 5 seconds for 30 seconds (6 retries spaced 5 seconds apart) before notifying the user that a connection cannot be obtained.

Connection not in auto-commit mode and transaction begun in same method as database access

If a connection does not have auto-commit enabled, multiple database statements can be executed in the same transaction. Because each transaction uses a significant number of resources, fewer transactions result in better performance. Therefore, if a connection is used for executing more than one statement, turn off auto-commit mode and use transactions to group a number of statements into one unit of work. Keep in mind that if a transaction has too many statements, the database can experience problems due to lack of memory.

Transactions can be started explicitly or implicitly. A servlet or session bean with bean-managed transactions (BMT) can start a transaction explicitly by calling begin() on a javax.transaction.UserTransaction object, which can be retrieved from naming or from the bean's EJBContext object. This is known as a *global transaction*. To commit a global transaction, call commit() on the UserTransaction object; to roll back the transaction, call rollback(). Entity beans and non-BMT session beans cannot explicitly begin global transactions.

If a global transaction is not started, a local transaction is started automatically by the database when a connection that is not in auto-commit mode is used. (Recall that if the connection is in auto-commit mode, each statement is executed in its own transaction.) To commit a local transaction, call commit() on the Connection object; call rollback() to roll back the transaction. After commit() or rollback() is called, the next statement executed with that connection is in a new local transaction. Local transactions have the advantage in that they can be started inside beans that have container-managed transactions (CMT), which include all entity beans and non-BMT session beans. Nested transactions are not allowed; if a global transaction is already started, local transactions are not allowed.

Global transactions can also be started implicitly by the container. The transaction demarcation attribute on the deployment descriptor determines if the container should start a transaction when a method is called on the bean. For example, if TX_REQUIRED is set, a global transaction is started automatically by the container on method invocation if a transaction is not already in effect. For more information about the different transaction demarcation attributes, see the Enterprise JavaBeans specification.

If a transaction is begun in the same method as the database access, recovery is straightforward and similar to the case of using a connection in auto-commit mode. The following example shows a BMT session bean that uses the UserTransaction object to handle the transaction:

```
public void myBMTMethod() throws java.rmi.RemoteException
{
   //get a userTransaction
   javax.transaction.UserTransaction tran =
      getSessionContext().getUserTransaction();

   //retry indicates whether to retry or not
   //numOfRetries states how many retries have
   // been attempted
   boolean retry = false;
   int numOfRetries = 0;

   java.sql.Connection conn = null;
   java.sql.Statement stmt = null;

   do {
      try {
        //begin a transaction
        tran.begin();
        //Assumes that a datasource has already been obtained
        //from JNDI
```

```
        conn = ds.getConnection();
        conn.setAutoCommit(false);
        stmt = conn.createStatement();
        stmt.execute("INSERT INTO EMPLOYEES VALUES
                        (0101, 'Bill', 'R', 'Smith')");
        tran.commit();
        retry = false;
      } catch(com.ibm.websphere.ce.cm.StaleConnectionException
              sce)
         {
         //if a StaleConnectionException is caught
         // rollback and retry the action
          try {
            tran.rollback();
         } catch (java.lang.Exception e) {
            //deal with exception
            //in most cases, this can be ignored
         }
         if (numOfRetries < 2) {
            retry = true;
            numOfRetries++;
         } else {
           retry = false;
         }
      } catch (java.sql.SQLException sqle) {
         //deal with other database exception
      } finally {
         //always cleanup JDBC resources
          try {
             if(stmt != null) stmt.close();
          } catch (java.sql.SQLException sqle) {
             //usually can ignore
          }
          try {
             if(conn != null) conn.close();
          } catch (java.sql.SQLException sqle) {
             //usually can ignore
          }
      }
   } while (retry) ;
}
```

Because the transaction is begun in the same method as the data access, recovery is fairly straightforward. When a StaleConnectionException is caught, the transaction is rolled back and the method retried. The rest of the execution is self-explanatory. If a StaleConnectionException occurs somewhere during execution of the try block, the transaction is rolled back, the retry flag is set to *true*, and the transaction is retried. Again, the number of retries is limited, because the exception might not be transient.

Connection not auto-commit and transaction begun in different method from database access

When a transaction is begun in a different method from the database access, an exception needs to be thrown from the data access method to the transaction access method so that it can

retry the operation. In an ideal situation, a method can throw an application-defined exception, indicating that the failure can be retried. However this is not always allowed, and often a method is defined only to throw a particular exception. This is the case with the ejbLoad and ejbStore methods on an enterprise bean. The next two examples explain each of these scenarios.

Example 1: Database access method can throw application exception

When the method that accesses the database is free to throw whatever exception it needs, the best practice is to catch StaleConnectionException and rethrow some application exception that can be interpreted to retry the method. The following example shows an EJB client calling a method on an entity bean with transaction demarcation TX_REQUIRED, which means that the container begins a global transaction when insertValue is called:

```
public class MyEJBClient {
   ... other methods here ...
   public void myEJBClientMethod()
   {
      MyEJB myEJB = myEJBHome.findByPrimaryKey("myEJB");

      boolean retry = false;

      do {
         try {
            retry = false;
            myEJB.insertValue();
         }
         catch(RetryableConnectionException retryable) {
       retry = true;
         }
         catch(Exception e) { /* handle some other problem */ }
      } while (retry);
  }
}

public class MyEJB implements javax.ejb.EntityBean {
   ... other methods here ...
   public void insertValue() throws RetryableConnectionException,
      java.rmi.RemoteException   {
      try
      {
         conn = ds.getConnection();
         conn.setAutoCommit(false);
         stmt = conn.createStatement();
         stmt.execute("INSERT INTO my_table VALUES (1)");
      }
      catch(com.ibm.websphere.ce.cm.StaleConnectionException
         sce) {
            getSessionContext().setRollbackOnly();
            throw new RetryableConnectionException();
      }
      catch(java.sql.SQLException sqle) {
         //handle other database problem
      }
      finally {
```

```
            //always cleanup JDBC resources
             try {
                if(stmt != null) stmt.close();
             } catch (java.sql.SQLException sqle) {
                //usually can ignore
             }
             try {
                if(conn != null) conn.close();
             } catch (java.sql.SQLException sqle) {
                //usually can ignore
              }
        }
    }
}
```

MyEJBClient first gets a MyEJB bean from the home interface, which is assumed to have been previously retrieved from JNDI. It then calls insertValue() on the bean. The method on the bean gets a connection and tries to insert a value into a table. If one of the methods fails with a StaleConnectionException, it marks the transaction for rollbackOnly (which forces the caller to roll back this transaction) and throws a new RetryableConnectionException, cleaning up the resources before the exception is thrown. The RetryableConnectionException is simply an application-defined exception that tells the caller that the method can be retried. The caller monitors RetryableConnectionException and, if it is caught, simply retries the method. In this example, because the container is beginning and ending the transaction, no transaction management is needed in the client or the server. Of course, the client could have started a global transaction and the behavior would still be the same, provided that the client also committed or rolled back the transaction.

Example 2: Database access method can throw only RemoteException or EJBException

Not all methods are allowed to throw exceptions defined by the application. If you are using bean-managed persistence (BMP), the ejbLoad() and ejbStore() methods are used to store the bean's state. The only exceptions that can be thrown from these methods are java.rmi.RemoteException or javax.ejb.EJBException, so something similar to the previous example cannot be used. Even if a subclass of RemoteException or EJBException were created, WebSphere would wrap the exception with java.rmi.RemoteException, so the caller would simply see RemoteException, not the subclass.

If you are using container-managed persistence (CMP), the container persists the bean, and it is the container that sees StaleConnectionException. If a stale connection is detected, by the time the exception is returned to the client it is simply a RemoteException, and so a simple catch block does not suffice. The container gets a connection from the pool every time a new method is called, so if the client retries the method, a new non-stale exception should be retrieved (remember that the pool is cleared when a connection is found to be stale).

Fortunately, there is a way to determine if the root cause of a RemoteException is a StaleConnectionException. When RemoteException is thrown to wrap another exception, the original exception is usually retained. All RemoteException instances have a detail property,

which is of type java.lang.Throwable. With this detail, we can trace back to the original exception and, if it is a StaleConnectionException, retry the transaction. In reality, when one of these RemoteExceptions flows from one JVM to the next, the detail is lost, so it is better to start a transaction in the same server as the database access occurs. For this reason, the following example shows an entity bean accessed by a session bean with bean-managed transaction demarcation. In the following example, a session bean starts a transaction and then calls an entity bean, but the code could be easily modified:

```
public class MySessionBean extends javax.ejb.SessionBean {
   ... other methods here ...
   public void mySessionBMTMethod() throws
      java.rmi.RemoteException
   {
      javax.transaction.UserTransaction tran =
          getSessionContext().getUserTransaction();

      boolean retry = false;

      do {
         try {
            retry = false;
            tran.begin();
            // causes ejbLoad() to be invoked
            myBMPBean.myMethod();
            // causes ejbStore() to be invoked
            tran.commit();
        }
        catch(java.rmi.RemoteException re) {
           try { tran.rollback();
           }
           catch(Exception e) {
              //can ignore
           }
           if (causedByStaleConnection(re))
              retry = true;
           else
              throw re;
        }
        catch(Exception e) {
           // handle some other problem
        }
        finally {
           //always cleanup JDBC resources
           try {
             if(stmt != null) stmt.close();
           } catch (java.sql.SQLException sqle) {
              //usually can ignore
           }
           try {
              if(conn != null) conn.close();
           } catch (java.sql.SQLException sqle) {
              //usually can ignore
           }
        }
      } while (retry);
   }
```

```
   public boolean causedByStaleConnection(java.rmi.RemoteException
      remoteException)
   {
      java.rmi.RemoteException re = remoteException;
      Throwable t = null;

      while (true) {
         t = re.detail;

         try { re = (java.rmi.RemoteException)t; }
         catch(ClassCastException cce) {
            return (t instanceof
            com.ibm.websphere.ce.cm.StaleConnectionException);
          }
      }
   }
}

public class MyEntityBean extends javax.ejb.EntityBean {
   ... other methods here ...
   public void ejbStore() throws java.rmi.RemoteException
   {
      try {
         conn = ds.getConnection();
         conn.setAutoCommit(false);
         stmt = conn.createStatement();
         stmt.execute("UPDATE my_table SET value=1 WHERE
            primaryKey=" + myPrimaryKey);
      }
      catch(com.ibm.websphere.ce.cm.StaleConnectionException
            sce) {
         //always cleanup JDBC resources
          try {
             if(stmt != null) stmt.close();
          } catch (java.sql.SQLException sqle) {
             //usually can ignore
          }
          try {
             if(conn != null) conn.close();
          } catch (java.sql.SQLException sqle) {
             //usually can ignore
          }

        // rollback the tran when method returns
        getEntityContext().setRollbackOnly();
        throw new java.rmi.RemoteException("Exception occurred in
                                        ejbStore", sce);
       }
      catch(java.sql.SQLException sqle) {
         // handle some other problem
      }
   }
}
```

In mySessionBMTMethod(), the session bean first retrieves a UserTransaction object from the session context and then begins a global transaction. Next, it calls a method on the entity bean, which calls the ejbLoad() method. If ejbLoad() runs successfully, the client then commits the

transaction, causing the ejbStore() method to be called. In ejbStore(), the entity bean gets a connection and writes its state to the database; if the connection retrieved is stale, the transaction is marked rollbackOnly and a new RemoteException that wraps the StaleConnectionException is thrown. That exception is then caught by the client, which cleans up the JDBC resources, rolls back the transaction, and calls causedByStaleConnection(), which determines if a StaleConnectionException is buried somewhere in the exception. If the method returns `true`, the retry flag is set and the transaction is retried; otherwise, the exception is rethrown to the caller.

The causedByStaleConnection() method looks through the chain of detail attributes to find the original exception. Multiple wrapping of exceptions can occur by the time the exception finally gets back to the client, so the method keeps searching until it encounters a non-RemoteException. If this final exception is a StaleConnectionException, we found it and *true* is returned; otherwise, there was no StaleConnectionException in the list (because StaleConnectionException can never be cast to a RemoteException), and *false* is returned.

If we are talking to a CMP bean instead of to a BMP bean, the session bean is exactly the same. The CMP bean's ejbStore() method would most likely be empty, and the container after calling it would persist the bean with generated code. If a stale connection exception occurs during persistence, it is wrapped with a RemoteException and returned to the caller. The causedByStaleConnection() method would again look through the exception chain and find the root exception, which would be StaleConnectionException.

Auto connection cleanup

Auto connection cleanup is the standard mode in which WebSphere connection pooling operates. This indicates that at the end of a transaction, the transaction manager closes all connections that have enlisted in that transaction. This allows the transaction manager to ensure that connections are not being held for long periods of time and that the pool is not maxed out.

One ramification of having the transaction manager close the connections and return the connection to the free pool after a transaction ends is that an application cannot obtain a connection in one transaction and try to use it in another transaction. If the application tries this, a StaleConnectionException is thrown because the connection is already closed.

However, if the application needs to hold a connection outside of the scope of the transaction, the administrator can configure a datasource to not automatically clean up connections obtained from that datasource by using the disableAutoConnectionCleanup property on the datasource. This option should be set only if the application always closes its own connection. When this property is set to `true`, the pool quickly runs out of connections if the application does not close all connections as soon as they are finished being used.

WebSphere portability layers

WebSphere connection pooling has implemented portability layers to make all supported database platforms look similar to the application. In these layers, we monitor some common database-specific SQLExceptions and map them to WebSphere-specific exceptions (for example,

StaleConnectionException). This makes application development easier because database-specific error codes do not need to be known for these very common errors. It also makes it easy to change the database to a different platform if necessary. However, these are WebSphere-specific errors; if the application is ported to a different web application server, the application must be changed to support the database-specific error codes.

Connection pooling worst practices

The following very common problems with applications should be avoided, because they most often result in unexpected failures:

Do not close connections in a finalize method

If an application waits to close a connection or other JDBC resource until the finalize() method, the connection is not closed until the object that obtained it is garbage-collected, because that is when the finalize() method is called. This leads to problems if the application is not thorough about closing JDBC resources such as ResultSet objects. Databases can quickly run out of the memory required to store the information about all of the JDBC resources currently open. In addition, the pool can quickly run out of connections to service other requests.

Do not declare connections as static objects

It is never recommended that connections be declared as static objects. This is because if the connection is declared as static, it is possible that the same connection gets used on different threads at the same time. This causes a great deal of difficulty, within both connection pooling and the database. Therefore, the connection should always be obtained and released within the method that requires it. This does not cost the application much in the way of performance because the Connection object they obtain is from a pool of connections, where the overhead for establishing the connection has already been incurred.

Do not declare Connection objects as instance variables in servlets

In a servlet, all variables declared as instance variables act as if they are class variables. For example, if a servlet is defined with an instance variable

```
Connection conn = null;
```

this variable acts as if it is static. This implies that all instances of the servlet would use the same Connection object. This is because a single servlet instance can be used to serve multiple Web requests in different threads.

Do not manage data access in CMP beans

If a CMP bean is written so that it manages its own data access, this data access might be part of a global transaction. Generally, it is best to have data access occur in a BMP session bean.

Connection pooling best practices

Most of the best practices have been mentioned elsewhere in this paper. Following are the remaining that have not been explicitly called out:

Obtain and close connection in same method

It is recommended that an application obtain and close its connection in the same method that requires a connection. This keeps the application from holding resources that are not being used and leaves more available connections in the pool for other applications. In addition, it removes the temptation to use the same connection in multiple transactions, which by default is not allowed. Lastly, make sure to declare the Connection object in the same method as the getConnection() call in a servlet; otherwise, the connection object works as if it is a static variable (see the previous section for problems with this). There may be times when it is not feasible to close a connection in the same method in which the connection was obtained. For example, if the isolation level is Read Committed and locks are obtained, closing the connection or result set causes these locks to be dropped. If it is not desirable to have the locks dropped until a later time, the connection should not be closed until after it is safe to drop the locks.

If you opened it, close it
It is recommended that all JDBC resources that have been obtained by an application be explicitly closed by that application. WebSphere tries to clean up JDBC resources on a connection after the connection has been closed. However, this behavior should not be relied upon, especially if the application might be migrated to another platform in the future, because this might require rewriting the code.

For servlets, obtain datasource in init() method
For performance reasons, it is usually a good idea to put the JNDI lookup for the datasource into the init() method of the servlet. Because the datasource is simply a factory for connections that does not change (unless the application specifically calls set methods on the datasource), retrieving it in this method ensures that the lookup happens only once.

4. Connection pooling implementation details

This section focus on how WebSphere connection pooling manages the pool of Connection objects. It explains when the pool expands, when it contracts, and what happens under error conditions. Lastly, the statement caching mechanism is discussed.

Connection object lifecycle
A Connection object is always in one of three states: *DoesNotExist*, *InFreePool*, or *InUse*. Before a connection is created, it must be in the *DoesNotExist* state. After it is created, it can be either in the *InUse* or the *InFreePool* state, depending on whether it has been allocated to an application.

Figure 1 shows the three valid states of a Connection object. Between these three states are lines with arrows indicating transitions between states. These transitions are labeled with guarding conditions. A *guarding condition* is one in which `true` indicates when the transition can be taken into another legal state. The entire list of guarding conditions and descriptions is shown in Table 1. For example, the transition from *InFreePool* to *InUse* can be made if and only if the application has called DataSource.getConnection (*getConnection*) and a free connection is available in the pool (*freeConnectionAvailable*). This transition can be specified as follows:

```
InFreePool -> InUse:
     getConnection AND
     freeConnectionAvailable
```

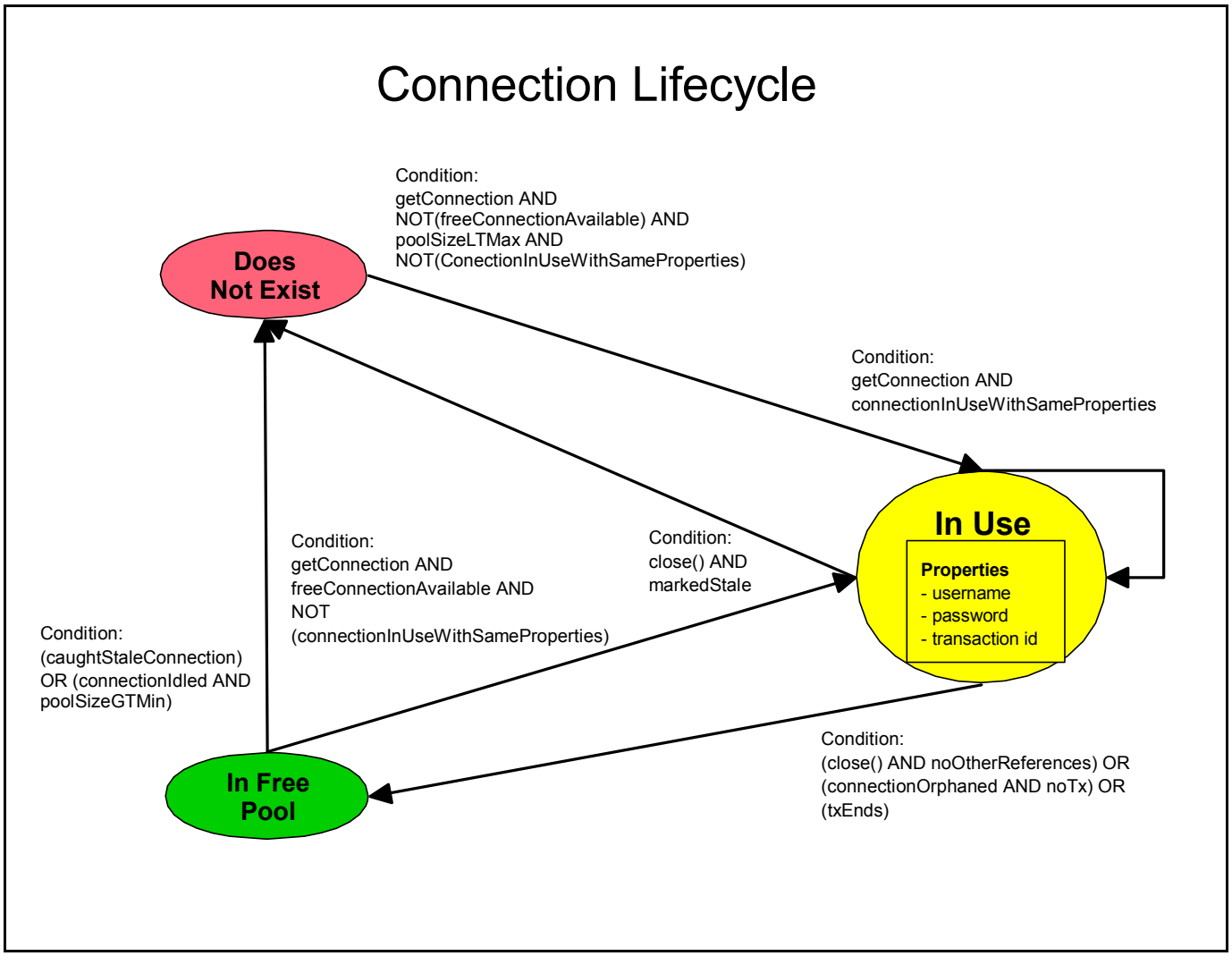

Figure 1. Connection Lifecycle

Condition	Description
caughtStaleConnection	A StaleConnectionException has been caught
close	Application calls close method on the Connection object
connectionIdled	Connection has been idled and removed from the free pool
connectionInUseWith SameProperties	Connection already being used by this application with the same user name, password, and transaction ID
connectionOrphaned	Connection has been orphaned from the application and returned to the free pool
freeConnectionAvailable	Connection available for use in the free pool
getConnection	Application calls getConnection method on DataSource object.
markedStale	Connection has been marked as stale
noTx	No transaction is in force
otherReference	The application or transaction manager is still holding a reference to this Connection object
poolSizeGTMin	Connection pool size is greater than the minimum pool size (minimum number of connections)
poolSizeLTMax	Pool size is less than the maximum pool size (maximum number of connections)
txEnds	The transaction has ended

Table 1. Connection Lifecycle Transaction Conditions

The first set of transitions covered are those in which the application has requested a connection from WebSphere connection pooling. In some of these scenarios, a new connection to the database results. In others, the connection might be retrieved from the connection pool or shared with another request for a connection.

Every connection begins its lifecycle in the *DoesNotExist* state. When an application server starts up, the connection pool does not exist. Therefore, there are no connections. The first connection is not created until an application requests its first connection. Additional connections are created as needed, according to the guarding condition. The transition from *DoesNotExist* to *InUse* is as follows:

```
DoesNotExist -> InUse:
      getConnection AND
      NOT(freeConnectionAvailable) AND
      poolSizeLTMax AND
      NOT(conectionInUseWithSameProperties)
```

This transition specifies that a Connection object is not created unless the following conditions occur:

- The application calls getConnection() on the DataSource (*getConnection)*
- No connections are available for use in the free pool (*NOT(freeConnectionAvailable*))
- The pool size is less than the maximum pool size (*poolSizeLTMax*)

- There are no free connections currently in the pool that have the same user name, password, and transaction ID as a connection that is currently in use (*NOT(connectionInUseWithSameProperties)*)

This transition implies the following about how connection pooling works:

- All connections begin in the *DoesNotExist* state and are only created when the application requests a connection. This implies that the pool is grown from 0 to the minimum number of connections as the application requests new connections. The pool is *not* created with the minimum number of connections when the server starts up.
- If a connection is already in use by the application with the same user name, password, and transaction ID (that is, it's in the same transaction), the connection is shared by two or more requests for a connection. In this case, a new connection is not created.

The idea of connection sharing is seen in the transition on the *InUse* state. The transition is as follows:

```
InUse -> InUse:
      getConnection AND
      connectionInUseWithSameProperties
```

This transition states that if an application requests a connection (*getConnection)* with the SAME user name, password, and transaction ID as a connection that is already in use *(connectionInUseWithSameProperties*), the existing connection is shared.

This transition implies the following about how connection pooling works:

- Connections can be shared by the same user (user name and password) but only within the same transaction. Because a transaction is normally associated with a single thread, connections should never be shared across threads. It is possible to see the same connection on multiple threads at the same time, but this is an error state and is induced by application programming error.

The transition from the *InFreePool* state to the *InUse* state is the most common transition when the application requests a connection from the pool. This transition is stated as follows:

```
InFreePool -> InUse:
      getConnection AND
      freeConnectionAvailable AND
      NOT (connectionInUseWithSameProperties)
```

This transition states that a connection will be placed in use from the free pool if the application has issued a getConnection() call on the DataSource *(getConnection*), a connection is available for use in the connection pool (*freeConnectionAvailable*), and no connection is already in use in the transaction with the same user name and password (*NOT(connectionInUseWithSameProperties*)).

This transition implies the following about connection pooling:

- Any request for a connection that can be fulfilled by a connection from the free pool does not result in creating a new connection to the database. Therefore, if there is never more than one connection being used at a time from the pool by any number of applications, the pool never grows beyond a size of 1. This may be less than the minimum number of connections specified for the pool. The only way that a pool grows to the minimum number of connections is if the application has multiple concurrent requests for connections that must result in a new connection being created.

All of the transitions so far have covered the scenarios of getting a connection for use by the application. At this point, the second set of transitions that result in a connection being closed and either returned to the free pool or destroyed will be covered.

Connections should be explicitly closed by the application by calling close() on the Connection object. In most scenarios, this results in the following transition:

```
InUse -> InFreePool:
      (close AND NOT(otherReferences)) OR
      (connectionOrphaned AND noTx) OR
      (txEnds)
```

The transition from the *InUse* state to the *InFreePool* state can be caused by three different conditions:

- If the application calls close() (*close*) and there are no references (*NOT(otherReferences)*) either by the application (application sharing) or by the transaction manager (who holds a reference when the connection is enlisted in a transaction), the Connection object is returned to the free pool.
- If the connection has not been used by the application for the orphan timeout amount of time (*connectionOrphaned*) and there is no transactional context (*noTx*), the connection is marked for return to the pool and returned to the free pool during the next orphan timeout.
- If the connection was enlisted in a transaction but the transaction manager has ended the transaction (*txEnds*), the connection is closed by default and returned to the pool.

This transition implies the following about connection pooling:

- When the application calls close() on a connection, it is returning the connection to the pool of free connections; it is not closing the connection to the database.
- When the application calls close() on a connection, if the connection is currently being shared, it is not returned to the free pool. Instead, after the last reference to the connection is dropped by the application, the connection is returned to the pool.
- When the application calls close() on a connection enlisted in a transaction, the connection is not returned to the free pool. Because the transaction manager must also hold a reference to the connection object, it cannot be returned to the free pool until the transaction has ended. This is because once a connection is enlisted in a transaction, it cannot be used in any other transaction by any other application until after the transaction has been completed.

There is a case in which the application's calling close() can result in the connection to the database being closed, bypassing the return of the connection to the pool. This happens if one of

the connections in the pool has been determined to be *stale*. A connection is considered stale if it can no longer be used to contact the database. One reason for a connection being marked as stale is if the database server has been shut down. When a connection is marked as stale, the entire pool is cleansed, because it is very likely that all of the connections are stale for the same reason. This cleansing includes marking all of the currently *InUse* connections as stale so they may be destroyed upon closing. The following transition states the behavior on a call to close() when the connection is marked as stale:

```
InUse -> DoesNotExist:
      close AND
      markedStale
```

This transition states that if the application has called close() on the connection (*close*) and it has been marked as stale during the pooling cleansing step (*markedStale*), the connection object is closed to the database and not returned to the pool.

Lastly, connections can be closed to the database and removed from the pool. The following transition shows this:

```
InFreePool -> DoesNotExist:
      caughtStaleConnection OR
      (connectionIdled AND
      poolSizeGTMin)
```

This transition states that there are two cases in which a connection is removed from the free pool and destroyed. First, if a StaleConnectionException is caught (*caughtStaleConnection*), all connections currently in the free pool are destroyed. This is because most likely all connections in the pool are stale. Second, if the connection has been in the free pool for longer than the idle timeout amount (*connectionIdled*) and the pool size is greater than the minimum number of connections (*poolSizeGTMin*), the connection is removed from the free pool and destroyed. This mechanism enables the pool to shrink back to its minimum size when the demand for connections is reduced.

WebSphere statement cache

WebSphere provides a mechanism for caching previously prepared statements. Caching prepared statements improves response times, because an application can reuse a PreparedStatement on a connection if it exists in that connection's cache, bypassing the need to create a new PreparedStatement.

When an application creates a PreparedStatement on a connection, the connection's cache is first searched to determine if a PreparedStatement with the same SQL string already exists. This search is done by using the entire string of SQL statements in the prepareStatement() method. If a match is found, the cached PreparedStatement is returned for use. If it is not, a new PreparedStatement is created and returned to the application.

As the prepared statements are closed by the application, they are returned to the connection's cache of statements. By default, only 100 prepared statements can be kept in cache for the entire pool of connections. For example, if there are ten connections in the pool, the number of cached prepared statements for those ten connections cannot exceed 100. This ensures that a limited number of prepared statements are concurrently open to the database, which helps to avoid resource problems with a database.

Elements are removed from the connection's cache of prepared statements only when the number of currently cached prepared statements exceeds the statementCacheSize (by default 100). If a prepared statement needs to be removed from the cache, it is removed and added to a vector of discarded statements. As soon as the method in which the prepared statement was removed has ended, the prepared statements on the discarded statements vector are closed to the database. Therefore, at any given time, there might be 100 plus the number of recently discarded statements open to the database. The extra prepared statements are closed after the method ends.

The number of prepared statements to hold in the cache is configurable on the datasource. Each cache should be tuned according to the applications need for prepared statements. For more details on tuning the prepared statement cache, see Section 6.

Databases and JDBC datasources supported

One of the features of WebSphere connection pooling is that the application is isolated from differences in the database platforms. This buffering includes mapping some database-specific exceptions to WebSphere-specific exceptions. The benefit of this mapping is that applications can be written to the WebSphere exceptions; they not have to enumerate all of the vendor-specific SQLExceptions that would map to these WebSphere exceptions. In addition, this mapping makes it easier to change the datastore for the application without disrupting the application. Lastly, this buffering masks anomalies in how the database or JDBC datasource supports the JDBC 2.0 specification. However, applications can still be written to the vendor-specific SQLExceptions and ignore the WebSphere exceptions.

In order to support this layer of abstraction, WebSphere connection pooling incorporates portability layers for all of the different databases and JDBC resources (often referred to as JDBC drivers) supported. The most current list of supported databases and JDBC resources can be found at: http://www.ibm.com/software/webservers/appserv/doc/latest/prereq.html

If a database platform or JDBC resource that you are using is not listed on this page, it is not a supported configuration. However, due to the design of connection pooling, most JDBC resource datasources can be configured to be used. To do this, the administrator can type in the DataSource implementation class when creating the WebSphere datasource instead of selecting a supported DataSource from the datasouce drop-down list.

Before calling IBM support with a problem, determine if the JDBC resource and database version being used is supported by checking the Web site.

5. Application programming with connection pooling

There are several steps you can take in your application code to successfully utilize WebSphere connection pooling.

Cache JNDI lookups: From a performance perspective, JNDI lookups are an expensive operation. For this reason, an application should perform these operations as infrequently as possible. This applies not only to DataSource lookups, but also to the lookup of the javax.transaction.UserTransaction object in the situation of client-managed transactions. Create a separate method that performs these lookup operations. Call this method from the servlet's init() method or from the enterprise bean's ejbActivate() method.

WebSphereSamples.ConnPool.EmployeeBMPBean

```
private void getDS() {
      try {
// Note the new Initial Context Factory interface
// available in Version 4.0

            Hashtable parms = new Hashtable();
            parms.put(Context.INITIAL_CONTEXT_FACTORY,

"com.ibm.websphere.naming.WsnInitialContextFactory");
            InitialContext ctx = new InitialContext(parms);

// Perform a naming service lookup to get the DataSource object.
            ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");

      } catch (Exception e) {
            System.out.println("Naming service exception: " +
                                    e.getMessage());
            e.printStackTrace();
      }
}
public void ejbActivate() throws java.rmi.RemoteException {
      getDS();
}
```

Because the lookup is in a separate method rather than a part of the init() or ejbActivate() method, it can also be called separately, if for some reason the object becomes invalid.

WebSphereSamples.ConnPool.EmployeeBMPBean

```
private void loadByEmpNo(String empNoKey) throws
javax.ejb.FinderException{
      if (ds == null) getDS();
}
```

Opening Connections: In general, an application should open only one connection to the database at a time. As mentioned previously in this document, if two getConnection() calls with the same parameters are issued in the same global transaction, only a single connection is allocated. As a matter of fact, you are allowed to open only one single-phase-commit connection within a global transaction. However, if the application is not running in a global transaction (as

is the case with most servlets), two getConnection() calls result in two separate connections. Not only does this utilize more resources than necessary, but it also causes your connection pool to fill twice as fast, often resulting in ConnectionWaitTimeoutExceptions.

If the application requires multiple simultaneous connections, close each connection as soon as it is no longer required, to free that connection up for another user.

Only one ResultSet object is allowed per statement object. If an application opens a second ResultSet, the first ResultSet is implicitly closed. Any attempts to access the first ResultSet throws an exception.

Closing connections: It is very important that ResultSet, Statement, PreparedStatement, and Connection objects get closed properly by your application. If Connections are not closed properly, users can experience long waits for connections to time-out and be returned to the free pool. Unclosed ResultSet, Statement, or PreparedStatement objects unnecessarily hold resources at the database. To ensure that these objects are closed in both correct execution and exception or error states, always close ResultSet, Statement, PreparedStatement, and Connection object in the *finally* section of a try/catch block. If exceptions are thrown on the close() call, these can be ignored.

WebSphereSamples.ConnPool.EmployeeBMPBean

```
private void loadByEmpNo(String empNoKey) throws
javax.ejb.FinderException{
      try {
            conn = ds.getConnection();
            ps = conn.prepareStatement(sql);
            ps.setString(1, empNoKey);
            rs = ps.executeQuery();
            //Work with ResultSet
      } catch (SQLException sq) {
            //Handle Exception
      } finally {
            if (rs != null) {
                  try {
                        rs.close();
                  } catch (Exception e) {
                        System.out.println("Close Resultset Exception: "
                                          + e.getMessage());
                  }
            }
            if (ps != null) {
                  try {
                        ps.close();
                  } catch (Exception e) {
                        System.out.println("Close Statement Exception: "
                                          + e.getMessage());
                  }
            }
            if (conn != null) {
                  try {
                        conn.close();
                  } catch (Exception e) {
```

```
                        System.out.println("Close connection exception: "
                                                + e.getMessage());
                    }
                }
            }
}
```

Stale connection exceptions: Stale connections are those connections that for some reason can no longer be used. How an object handles StaleConnectionExceptions is largely based on the type of transaction in which the object is participating. See section 3 of this document for more details.

Connection wait timeout: If an object attempts to obtain a database connection from the pool but no connection is available, a ConnectionWaitTimeoutException will be thrown. The connection timeout value on the connection pool, which defaults to 3 minutes, specifies the amount of time the object waits for a connection before the ConnectionWaitTimeoutException is thrown. See previous sections of this document for more information on the connection timeout value.

Instances of ConnectionWaitTimeoutException generally do not result in a retry of the getConnection() call, because you would generally not expect an end user to wait for more than three minutes to receive a response to their request.

WebSphereSamples.ConnPool.EmployeeList

```
try {
       conn = ds.getConnection();
         //Work with Connection
}
catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {
       System.out.println("Connection Wait Timeout Exception during
                           get connection or process SQL: " +
                           cw.getMessage());

//In general, we do not want to retry after this exception,
//so set the retry count to 0
                retryCount=0;
}
```

Frequent occurrences of ConnectionWaitTimeoutException often indicate a poorly tuned connection pool or an over-utilized database server.

Handling exceptions: Depending on your application, you might choose to handle SQLException, StaleConnectionException, and so on at the object level and throw a more generic exception to the client, or you might choose to throw the specific exception to the client and allow it to be handled there. How you choose to handle these exceptions will depend on your application.

For example, if an object uses bean-managed transactions, it controls when a transaction begins and ends. For this reason, the object can determine where a StaleConnectionException occurred,

and whether it is valid to retry the transaction. However, if an object uses container-managed transactions, it does not know what may have transpired in the transaction prior to its call. In this situation, the exception must be passed to the object's client, which is in a better situation to decide if the operation is retriable.

6. Administration of connection pools

This section describes how to create and configure a datasource in the adminstrative console.

In WebSphere, datasources and connections pools generally have a one-to-one relationship (this changes when clones are introduced). When an administrator configures a datasource, she is essentially configuring an entire pool of connections for use by applications. After the datasource is defined, all an application need do is look up the datasource from JNDI and use it to obtain connections. One of the benefits of using WebSphere datasources is that applications need to know only the datasource name for retrieval from JNDI. Attributes such as user name, password, and database name do not have to be specified by an application.

There are two approaches to creating a datasource in WebSphere. The first approach, detailed previously, is to have an application create the datasource on demand. The application must have detailed information about how to set up a datasource, such as the database name, default user name and password, and the datasource class used to connect to the database. Because it is not practical for an application to have such detailed information, this approach is not used often. The second approach is to have the administrator create the datasources in the WebSphere Administrative Console. Centralizing the creation of datasources in this manner allows applications to obtain connections to a database without needing the particulars of getting a connection. The only information required by an application in this scenario is the datasource's JNDI name, which is determined when the datasource is created. This is the most common scenario for creating datasources. The rest of this section details how to create a datasource in the WebSphere Advanced Edition for Multiplatforms administrative console, the different properties for a datasource, and how to tune the connection pool for best performance. The tips and techniques detailed in this section for tuning the connection pool are useful for all datasources, no matter how they are created.

Creating the datasource

The first step towards connecting to a database is to create the datasource in the WebSphere Administrative Console. The following steps are involved in creating a datasource:

1. Create a JDBC provider
2. Create a datasource

These steps can be accomplished separately or at the same time by using the Create Data Source wizard.

Step 1: Create a JDBC provider

In this step, the JDBC provider (often referred to as the JDBC driver) package is defined and installed on a node. To create a JDBC provider in the Advanced Edition administrative console, select **Resources > JDBC Providers**; right-click and select **New**.

The information needed for this step is as follows:

- Name - A required property used to identify the JDBC provider on the machine. The name is used when creating the datasource to identify which JDBC provider should be used by the datasource for connections to the database.
- Description - An optional field used to supply detailed information about the JDBC resource.
- Implementation Class - A required field that defines the datasource class used to connect to a database. A datasource class can be selected from the list, or a different class can be typed. If a class is in the list, it is a configuration supported in WebSphere connection pooling. Any class typed in is not a supported configuration. Unsupported configurations will not have the advantages of WebSphere exceptions by default. This means that applications using unsupported configurations must catch vendor-specific SQLExceptions instead of the WebSphere exceptions unless the errorMap property on the datasource is used to map the vendor-specific SQLExceptions to WebSphere exceptions. In addition, there are no guarantees of how well these unsupported configurations will work. For more information on supported configurations, please refer to "Databases and JDBC datasources supported" on page 33.
- Node on which to install JDBC resource - A required property that indicates the WebSphere node on which to install the JDBC provider. Select one from the list of the WebSphere nodes in the domain.
- Location of the JDBC provider - Required for the WebSphere server to locate the JDBC provider. In this step, after the node on which to install the JDBC provider is selected, a file dialog opens to aid in locating the .jar or .zip file that contains the datasource class selected for the Implementation Class property. There might be more than one .jar or .zip file needed to be selected. For example, Informix requires the addition of ifxjdbc.jar and ifxjdbcx.jar to this list.

After the required information is entered and accepted, the JDBC provider is installed on a node. Then, any datasource on the specified node that requires a connection to the datastore defined by the implementation class can use this JDBC provider.

Step 2: Create a datasource

To create a datasource, select the Data Sources folder under a specific JDBC Provider on the adimistrative console and select **New**. Two types of information that are required when creating a datasource are general properties and connection pooling properties. General properties are those specific to the datasource implementation class being used for the datasource. Connection pooling properties are those used to tune the pool of connections within the application server.

On the General properties tab, the following properties are listed:

- Name - A required property that identifies the datasource.

- JNDI Name - The name under which the datasource will be bound into the JNDI namespace under the jdbc subcontext. If the JNDI name is not specified, the datasource is bound into JNDI using the name property.
- Description - A text field where comments on the purpose of the datasource can be placed.
- Database Name - The name of the database to which the datasource is providing connections. This is not a required field, because different JDBC resources require different properties to create a datasource. For more on the specific properties that a datasource implementation class requires, view the online help or Appendix D of this document.
- JDBC Provider - The JDBC provider created in step 1. This is a required field.
- User - Default user name for a connection. When this field is filled in, all getConnection() requests use the default user name to obtain the connection. If a user name and password is specified in the getConnection() request, those values override the default values. If the User property is specified, the Password property must also be specified.
- Password - Default password used with the default user when a getConnection() request is made. If the Password property is specified, the User property must also be specified.
- Custom Properties - A grid of property/value pairs that is intended to be used to specify vendor-specific properties on the datasource. Any properties that the datasource implementation class requires must be placed in the grid. These values can be determined by viewing the online help or by reading Appendix D of this document. In addition, any property that the vendor supports on the datasource implementation class may be specified in this grid.

On the Connection Pooling tab, the following properties are listed:

- Minimum Pool Size - The minimum number of connections that the connection pool can hold. By default, this value is 1. Any non-negative integer is a valid value for this property. The minimum pool size can affect the performance of an application. Smaller pools require less overhead when the demand is low because fewer connections are being held open to the database. On the other hand, when the demand is high, the first applications will experience a slow response because new connections will have to be created if all others in the pool are in use.

- Maximum Pool Size - The maximum number of connections the connection pool can hold. By default, this value is 10. Any positive integer is a valid value for this property. The maximum pool size can affect the performance of an application. Larger pools require more overhead when demand is high because there are more connections open to the database at peak demand. These connections persist until they are idled out of the pool. On the other hand, if the maximum is smaller, there might be longer wait times or possible connection timeout errors during peak times. The database must be able to support the maximum number of connections in the application server in addition to any load that it may have outside of the application server.

- Connection Timeout - The maximum number of seconds that an application waits for a connection from the pool before timing out and throwing a ConnectionWaitTimeoutException to the application. The default value is 180 seconds (3 minutes). Any non-negative integer is a valid value for this property. Setting this value to 0 disables the connection timeout.

- Idle Timeout - The number of seconds that a connection can remain free in the pool before the connection is removed from the pool. The default value is 1800 seconds (30 minutes). Any non-negative integer is a valid value for this property. Connections need to be idled out of the pool because keeping connections open to the database can cause memory problems with the database. However, not all connections are idled out of the pool, even if they are older than the Idle Timeout setting. A connection is not idled if removing the connection would cause the pool to shrink below its minimum size. Setting this value to 0 disables the idle timeout.

- Orphan Timeout - The number of seconds that an application is allowed to hold an inactive connection. The default is 1800 seconds (30 minutes). Any non-negative integer is a valid value for this property. If there has been no activity on an allocated connection for longer than the Orphan Timeout setting, the connection is marked for orphaning. After another Orphan Timeout number of seconds, if the connection still has had no activity, the connection is returned to the pool. If the application tries to use the connection again, it is thrown a StaleConnectionException. Connections that are enlisted in a transaction are not orphaned. Setting this value to 0 disables the orphan timeout.

- Statement Cache Size (**Version *4.0***) - The number of cached prepared statements to keep for an entire connection pool. The default value is 100. Any non-negative integer is a valid value for this property. When a statement is cached, it helps performance, because a statement is retrieved from the cache if a matching statement is found, instead of creating a new prepared statement (a more costly operation). The statement cache size does not change the programming model, only the performance of the application. The statement cache size is the number of cached statements for the entire pool, not for each connection.

Setting the statement cache size in the administrative console is a new option in Version 4.0. In previous versions, this value could be set only by using a datasources.xml file. Use of datasources.xml has been deprecated in Version 4.0.

- Disable Auto Connection Cleanup (**Version *4.0***) - Whether or not the connection is closed at the end of a transaction. The default is *false*, which indicates that when a transaction is completed, WebSphere closes the connection and returns it to the pool. This means that any use of the connection after the transaction has ended results in a StaleConnectionException, because the connection has been closed and returned to the pool. This mechanism ensures that connections are not held indefinitely by the application. If the value is set to *true*, the connection is not returned to the pool at the end of a transaction. In this case, the application must return the connection to the pool by

calling close(). If the application does not close the connection, the pool can run out of connections for other applications to use.

After the required properties have been entered and accepted, a datasource is created and bound into JNDI by using the JNDI Name setting. At this point, any application that requires a connection to the database specified in the datasource can simply retrieve the datasource from JNDI and obtain a connection by using the getConnection() method.

Tuning the connection pool

Performance improvements can be made by correctly tuning the parameters on the connection pool. This section details each of the properties found on the Connection Pooling tab and how they can be tuned for optimal performance.

- Minimum Pool Size - The minimum number of connections that the connection pool can hold open to the database. The default value is 1. In versions 3.5 and 4.0, the pool does not create the minimum number of connections to the database up front. Instead, as additional connections are needed, new connections to the database are created, growing the pool. After the pool has grown to the minimum number of connections, it does not shrink below the minimum.

  The correct minimum value for the pool can be determined by examining the applications that are using the pool. If it is determined, for example, that at least four connections are needed at any point in time, the minimum number of connections should be set to 4 to ensure that all requests can be fulfilled without connection wait timeout exceptions.

  At off-peak times, the pool shrinks back to this minimum number of connections. A good rule of thumb is to keep this number as small as possible to avoid holding connections unnecessarily open.

- Maximum Pool Size - The maximum number of connections that the connection pool can hold open to the database. The pool holds this maximum number of connections open to the database at peak times. At off-peak times, the pool shrinks back to the minimum number of connections.

  The best practice is to ensure that only one connection is required on a thread at any time. This avoids possible deadlocks when the pool is at maximum capacity and no connections are left to fulfill a connection request. Therefore, with one connection per thread, the maximum pool size can be set to the maximum number of threads. When using servlets, this can be determined by looking at the MaxConnections property in the Servlet Engine.

  If multiple connections are required on a thread, the maximum pool size value can be determined using the following formula:

```
T * (C - 1) + 1
```

where T is the maximum number of threads and C is the number of concurrent database connections necessary per thread. For example, if the application uses two threads, each of which requires three connections, and a maximum of 10 applications could be using the pool at peak times, the resulting value would be (2 * 10) * (3 - 1) + 1, or 41.

The database must be tuned to be able to handle the maximum number of connections in the pool plus any additional connections required by programs other than the application server. For example, if the maximum pool size is set to 25, indicating that the application server may hold 25 connections through this datasource at peak times, and if there is a second application (or even a second WebSphere datasource) that requires an additional 30 connections, ensure that the database can serve 55 connections concurrently.

- Connection Timeout - The maximum number of seconds that an application waits for a connection from the pool before timing out and throwing ConnectionWaitTimeoutException to the application. If applications are catching ConnectionWaitTimeoutException often, it usually means one of two things: The connection timeout property is set too low, or the connection pool is always at maximum capacity and cannot find a free connection for the application to use. If the exception is being caused by the property's being set too low, the solution is to set it to a higher value. If the exception is caused by too few connections in the pool, the maximum pool size setting needs to be investigated.

- Idle Timeout - The number of seconds that a connection can remain free in the pool before the connection is removed from the pool. Setting the idle timeout too low can cause performance problems because the overhead of closing a connection to the database is frequently encountered. In addition, because the connections have been closed, when peak times hit, not enough connections are in the pool, and new connections need to be created. On the other hand, holding connections in the pool for great amounts of time can result in memory problems in some databases. The database may hold information on all of the transactions that have occurred on the connection. When the connection is held open for a long time, this information builds up on the database server, which then runs out of memory.

  The Idle Timeout property is application-dependent and often requires trial and error to find an optimal value.

- Orphan Timeout - The number of seconds that an application is allowed to hold an inactive connection. This timeout ensures that a rogue application does not unnecessarily hold connections out of the free pool. If this value is set too low, it is possible that while an application is processing data, the unused connection can be orphaned and returned to the pool. This results in a StaleConnectionException being thrown to the application after the application tries to use the orphaned connection or one of its associated resources. If an application is experiencing large volumes of these exceptions and it is

determined that they are not due to a database or network failure, the orphan timeout should be investigated.

Applications that are written to obtain a connection, use it, and close it immediately after use do not tend not to have problems with orphaning of connections. A connection that is actively processing at the database is not orphaned, even if the processing takes longer than the Orphan Timeout number of seconds.

In the current implementation, a connection is marked for orphaning after it has been unused by the owning application for the Orphan Timeout number of seconds. Then, after another Orphan Timeout number of seconds, the connection is freed to the pool if the connection is still marked for orphaning. The only way that a connection can move from the orphaned state to the *InUse* state is if the owing application uses the connection again before it is freed to the pool. Connections participating in a transaction are never orphaned.

The Orphan Timeout property is application-dependent. If an application is continuously receiving StaleConnectionException, the root cause of the exception throwing needs to be determined. This can be done by viewing the message of the exception. If the message states that the class is already closed, it is most likely that the connection is orphaned. If an SQLException is part of the StaleConnectionException message, the exception was caused by the database. If the message indicates that a connection was orphaned, a workaround is to set the orphan timeout to a larger value while the application is reviewed to ensure that connections are closed as required.

- Statement Cache Size - The number of cached prepared statements to keep for an entire connection pool. The number of prepared statements created at the database can be drastically reduced by tuning the cache. The recommended value for the statement cache size can be determined using the following formula:

```
PrepStmtCache > diffPrepStmts * min(concurrent user threads,
                                    connectionPoolSize)
```

If the application prepares 10 different prepared statements (meaning 10 unique SQL strings), the maximum pool size is 20, and the number of concurrent threads is 20, the prepared statement cache size should be greater than 10 * (min(20,20)), or 200.

If the statement cache is set smaller than 200 in this example, statements may be removed from the queue because of space limitations before they can be reused. This means that once an application needs to reuse the prepared statement, it has already been closed to the database and, hence, a new prepared statement exactly like the one that may have been recently destroyed needs to be recreated.

This formula uses the minimum of the number of concurrent threads or the connection pool size to determine the cache size. This is because each connection holds its own

cache of statements. If there are 10 different prepared statements that can be created on a connection, and there are 20 connections, each which might have the 10 different prepared statements, this results in the entire pool containing 200 cached prepared statements.

Appendix A - WebSphere connection pooling vs. Connection Manager

Starting with IBM WebSphere Application Server Version 3.0, the JDBC 2.0 Optional Package API was implemented to provide connection pooling capability to make more efficient use of database connections.

Prior to Version 3.0, the JDBC 2.0 Optional Package API was not available. Therefore, WebSphere Application Server provided this capability in the IBM WebSphere Connection Manager. A servlet would communicate with the Connection Manager, which maintained a pool of open database connections to JDBC or ODBC database products. When the servlet received a connection from the pool, it could communicate directly with the database using its APIs.

The WebSphere Connection Manager was supported but deprecated in Version 3.5. It was highly recommended that all applications be migrated to the connection pooling model. In Version 4.0, the Connection Manager model is no longer available. Therefore, all applications using the Connection Manager model must be migrated to the connection pooling model.

Appendix B - Deprecated in Version 4.0

Two methods in com.ibm.websphere.advanced.cm.factory.DataSourceFactory are deprecated in Version 4.0. These methods are createJTADataSource() and createJDBCDataSource(); both required a com.ibm.websphere.advanced.cm.factory.Attributes object as the parameter. These methods have been replaced in Version 4.0 with the getDataSource() method, which requires a java.util.Properties object as the parameter. The deprecated methods still function but it is not recommended that new applications be written using the deprecated method.

According to the JDBC 2.0 Core API specification, the DriverManager class has been deprecated. Therefore, any application using this class should be rewritten to use WebSphere connection pooling, which uses the datasource method described in the JDBC 2.0 Optional Package API to obtain connections to the database. For more information, see the JDBC 2.0 Core API specification.

The package com.ibm.ejs.dbm.jdbcext.* has been deprecated. Applications that use this package can still retrieve a datasource, but new datasources cannot be created or bound into JNDI by using this interface. All new datasources must be created by using com.ibm.websphere.advanced.cm.factory.DataSourceFactory.

The class com.ibm.ejs.cm.portability.StaleConnectionException has been deprecated. Applications currently using this class will still function, but it is recommended that new applications be written using com.ibm.websphere.ce.cm.StaleConnectionException.

Appendix C - No longer supported in Version 4.0

The Connection Manager in WebSphere 2.0 was deprecated in Version 3.x. In Version 4.0, the Connection Manager is no longer supported. If an application attempts to obtain a connection using the Version 2.0 Connection Manager, an error indicating that this feature is no longer supported is displayed.

ConnectionPreemptedException, introduced in Version 3.02, was not supported in Version 3.5, but it was still available through Version 3.02 connection pooling. In Version 4.0, this exception no longer exists. StaleConnectionException has replaced ConnectionPremptedException in all cases.

The datasources.xml file has been removed in Version 4.0. The functionality provided through the use of this file is now part of the administrative console. For example, the statement cache size can now be set during datasource configuration on the administrative console.

Appendix D - Minimum required properties for vendor-specific datasources

The following list shows the datasource classes supported in Version 4.0 and their required properties. In Version 4.0, specific fields are designated for the databaseName, user, and password properties. Those properties' inclusion in the list does not mean that they should be added to the datasource properties list. Rather, inclusion on the list simply means that a value is typically required for that field.

DB2
COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource (one-phase commit protocol)
COM.ibm.db2.jdbc.DB2XADataSource (two-phase commit protocol)
Requires the following properties:

- databaseName - The name of the database from which the datasource will obtain connections. Example: "Sample"

DB2 for iSeries
iSeries Toolbox driver:
com.ibm.as400.access.AS400JDBCConnectionPoolDataSource (one-phase commit protocol)
com.ibm.as400.access.AS400JDBCXADataSource (two-phase commit protocol)
Requires the following properties:

- serverName - The name of the server from which the datasource will obtain connections. Example: "myserver.mydomain.com"

iSeries Native driver:
com.ibm.db2.jdbc.app.DB2StdConnectionPoolDataSource (one-phase commit protocol)
com.ibm.db2.jdbc.app.DB2StdXADataSource (two-phase commit protocol)
No required properties.

Oracle
oracle.jdbc.pool.OracleConnectionPoolDataSource (one-phase commit protocol)
oracle.jdbc.xa.client.OracleXADataSource (two-phase commit protocol)
Requires the following properties:

- URL - The URL that indicates the database from which the datasource will obtain connections. Example: "jdbc:oracle:thin:@myServer:1521:myDatabase", where myServer is the server name, 1521 is the port it is using for communication, and myDatabase is the database name.
- user - The user name used when obtaining connections. Example: "scott"
- password - The password for the specified user name. Example: "tiger"

Sybase
com.sybase.jdbc2.jdbc.SybConnectionPoolDataSource (one-phase commit protocol)
com.sybase.jdbc2.jdbc.SybXADataSource (two-phase commit protocol)
Requires the following properties:

- serverName - The name of the database server. Example: "myserver.mydomain.com"
- portNumber - The TCP/IP port number through which all communications to the server take place. Example: 4100

Merant
com.merant.sequelink.jdbcx.datasource.SequeLinkDataSource (one- and two-phase commit protocol)
Requires the following properties:

- serverName - The name of the server in which SequeLinkServer resides. Example: "myserver.mydomain.com"
- portNumber - The TCP/IP port that SequeLinkServer uses for communication. By default, SequeLinkServer uses port 19996. Example: "19996"
- databaseName - The name of the database from which the datasource will obtain connections. Example: "Sample"
- user - The user name used when obtaining connections. Example: "scott"
- password - The password for the specified user name. Example: "tiger"
- disable2Phase - By default, this datasource always creates two-phase connections. To use one-phase connections, set this property to *true*.

The same DataSource implementation class is used for both non-JTA and JTA-enabled datasources. Also, this class is found in sljcx.jar, not sljc.jar.

InstantDB
Com.ibm.ejs.cm.portability.IDBConnectionPoolDataSource (one-phase commit protocol)
Requires the following properties:

- url - The URL that indicates the database from which the datasource will obtain connections. Example: "jdbc:idb:*configuration_file*" where *configuration_file* is the name of the IDB configuration file.

Informix
com.informix.jdbcx.IfxConnectionPoolDataSource (one-phase commit protocol)
com.informix.jdbcx.IfxXADataSource (two-phase commit protocol)
Requires the following properties:

- serverName - The name of the Informix instance on the server. Example: "ol_myserver"
- portNumber - The port on which the instances is listening. Example: "1526"
- ifxIFXHOST - The physical name of the database server. Example: "myserver.mydomain.com"
- databaseName - The name of the database from which the datasource will obtain connections. Example: "Sample"
- user - The user name used when obtaining connections. Example: "scott"
- password - The password for the specified user name. Example: "tiger"
- informixLockModeWait - Although not required, this property enables you to set the number of seconds that Informix waits for a lock. By default, Informix throws an exception if it cannot immediately acquire a lock. Example: "600"

Appendix E - Code Sample: Servlet with no global transaction

WebSphereSamples\ConnPool\EmployeeList.java

```
//====================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//====================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

// Import JDBC packages and naming service packages.  Note the lack
// of an IBM Extensions package import.  This is no longer required.

import java.sql.*;
```

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

public class EmployeeList extends HttpServlet {
    private static DataSource ds   = null;
    private static String title    = "Employee List";

// ************************************************************
// * Initialize servlet when it is first loaded.              *
// * Get information from the properties file, and look up the    *
// * DataSource object from JNDI to improve performance of the    *
// * the servlet's service methods.                           *
// ************************************************************
    public void init(ServletConfig config)
    throws ServletException
    {
        super.init(config);
        getDS();
    }

// ************************************************************
// * Perform the JNDI lookup for the DataSource object.       *
// * This method is invoked from init(), and from the service     *
// * method of the DataSource is null                         *
// ************************************************************
    private void getDS() {
   try {
   // Note the new Initial Context Factory interface available in WebSphere 4.0
      Hashtable parms = new Hashtable();
      parms.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.ibm.websphere.naming.WsnInitialContextFactory");
      InitialContext ctx = new InitialContext(parms);
      // Perform a naming service lookup to get the DataSource object.
      ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
      }
      catch (Exception e) {
         System.out.println("Naming service exception: " + e.getMessage());
         e.printStackTrace();
      }
   }

// ************************************************************
// * Respond to user GET request                              *
// ************************************************************
   public void doGet(HttpServletRequest req, HttpServletResponse res) throws
     ServletException, IOException {
      Connection conn  = null;
      Statement stmt = null;
      ResultSet rs = null;
      Vector employeeList = new Vector();

// Set retryCount to the number of times you would like to retry after
// a StaleConnectionException
      int retryCount = 5;

// If the Database code processes successfully, we will set error = false
      boolean error = true;

      if (ds==null)
         getDS();

      do {
         try {
         // Get a Connection object conn using the DataSource factory.
            conn = ds.getConnection();
         // Run DB query using standard JDBC coding.
            stmt = conn.createStatement();
```

```
            String query   = "Select FirstNme, MidInit, LastName " +
                             "from Employee ORDER BY LastName";
            rs   = stmt.executeQuery(query);
            while (rs.next()) {
               employeeList.addElement(rs.getString(3) + ", " +
                                       rs.getString(1) + " " + rs.getString(2));
            }
//Set error to false to indicate successful completion of the database work
            error=false;
         }
         catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {

        // This exception is thrown if a connection can not be obtained from the
        // pool within a configurable amount of time.  Frequent occurrences of
        // this exception indicate an incorrectly tuned connection pool

            System.out.println("Connection Wait Timeout Exception during get
connection or process SQL: " + cw.getMessage());

//In general, we do not want to retry after this exception,
// so set the retry count to 0
            retryCount=0;
         }
         catch (com.ibm.websphere.ce.cm.StaleConnectionException sc) {

// This exception indicates that the connection to the database is no longer valid.
// Retry the operation several times to attempt to obtain a valid connection, display
// an error message if the connection still can not be obtained.

            System.out.println("Stale Connection Exception during get connection or
process SQL: " +  sc.getMessage());

            if (--retryCount == 0) {
               System.out.println("Five stale connection exceptions, displaying error
page.");
            }
         }
         catch (SQLException sq) {
            System.out.println("SQL Exception during get connection or process SQL: "
                               + sq.getMessage());

//In general, we do not want to retry after this exception, so set retry count to 0
            retryCount=0;
         }
         catch (Exception e) {
            System.out.println("Exception in get connection or process SQL: " +
                               e.getMessage());

//In general, we do not want to retry after this exception, so set retry count to 5
            retryCount=0;
         }
         finally {
            // Always close the connection in a finally statement to ensure proper
            // closure in all cases. Closing the connection does not close and
            // actual connection, but releases it back to the pool for reuse.
            if (rs != null) {
               try {
                  rs.close();
               }
               catch (Exception e) {
                  System.out.println("Close Resultset Exception: " +
                                     e.getMessage());
               }
            }
            if (stmt != null) {
               try {
                  stmt.close();
```

```
                }
                catch (Exception e) {
                   System.out.println("Close Statement Exception: " +
                           e.getMessage());
                }
             }
             if (conn != null) {
                try {
                   conn.close();
                }
                catch (Exception e) {
                   System.out.println("Close connection exception: " +
                            e.getMessage());
                }
             }
           }
         }
         while ( error==true && retryCount > 0 );
         // Prepare and return HTML response, prevent dynamic content from being cached
         // on browsers.
         res.setContentType("text/html");
         res.setHeader("Pragma", "no-cache");
         res.setHeader("Cache-Control", "no-cache");
         res.setDateHeader("Expires", 0);
         try {
             ServletOutputStream out = res.getOutputStream();
             out.println("<HTML>");
             out.println("<HEAD><TITLE>" + title + "</TITLE></HEAD>");
             out.println("<BODY>");
             if (error==true) {
                out.println("<H1>There was an error processing this request.</H1>
Please try the request again, or contact " +
              " the <a href='mailto:sysadmin@my.com'>System Administrator</a>");
             }
             else if (employeeList.isEmpty()) {
                out.println("<H1>Employee List is Empty</H1>");
             }
             else {
                out.println("<H1>Employee List </H1>");
                for (int i = 0; i < employeeList.size(); i++) {
                   out.println(employeeList.elementAt(i) + "<BR>");
                 }
             }
             out.println("</BODY></HTML>");
             out.close();
          }
          catch (IOException e) {
            System.out.println("HTML response exception: " + e.getMessage());
          }
       }
    }
```

Appendix F - Code Sample: Servlet with user transaction

WebSphereSamples\ConnPool\EmployeeListTran.java

```
//====================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
```

```
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
// Import JDBC packages and naming service packages.  Note the lack
// of an IBM Extensions package import.  This is no longer required.
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import javax.transaction.*;

public class EmployeeListTran extends HttpServlet {
    private static DataSource ds   = null;
    private UserTransaction ut = null;
    private static String title    = "Employee List";

// *************************************************************
// * Initialize servlet when it is first loaded.               *
// * Get information from the properties file, and look up the  *
// * DataSource object from JNDI to improve performance of the  *
// * the servlet's service methods.                            *
// *************************************************************
    public void init(ServletConfig config)
    throws ServletException
    {
        super.init(config);
        getDS();
    }

// *************************************************************
// * Perform the JNDI lookup for the DataSource and            *
// * User Transaction objects.                                 *
// * This method is invoked from init(), and from the service  *
// * method of the DataSource is null                          *
// *************************************************************
    private void getDS() {
        try {
         // Note the new Initial Context Factory interface available in WebSphere 4.0

            Hashtable parms = new Hashtable();
            parms.put(Context.INITIAL_CONTEXT_FACTORY,
                      "com.ibm.websphere.naming.WsnInitialContextFactory");
            InitialContext ctx = new InitialContext(parms);
            // Perform a naming service lookup to get the DataSource object.
            ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
            ut = (UserTransaction) ctx.lookup("java:comp/UserTransaction");
        } catch (Exception e) {
            System.out.println("Naming service exception: " + e.getMessage());
            e.printStackTrace();
        }
    }

// *************************************************************
// * Respond to user GET request                               *
```

```
// ************************************************************
    public void doGet(HttpServletRequest req, HttpServletResponse res) throws
ServletException, IOException
    {
        Connection conn  = null;
        Statement stmt = null;
        ResultSet rs = null;
        Vector employeeList = new Vector();

// Set retryCount to the number of times you would like to retry after a
// StaleConnectionException
        int retryCount = 5;
        // If the Database code processes successfully, we will set error = false
        boolean error = true;
        do {
            try {
                //Start a new Transaction
                ut.begin();
                // Get a Connection object conn using the DataSource factory.

                conn = ds.getConnection();
                // Run DB query using standard JDBC coding.
                stmt = conn.createStatement();
                String query   = "Select FirstNme, MidInit, LastName " +
                                 "from Employee ORDER BY LastName";
                rs   = stmt.executeQuery(query);
                while (rs.next()) {
                    employeeList.addElement(rs.getString(3) + ", " + rs.getString(1) +
                                            " " + rs.getString(2));
                }
           //Set error to false to indicate successful completion of the database work
                error=false;
            } catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {

              // This exception is thrown if a connection can not be obtained from the
              // pool within a configurable amount of time.  Frequent occurrences of
              // this exception indicate an incorrectly tuned connection pool

                System.out.println("Connection Wait Timeout Exception during get
connection or process SQL: " + c.getMessage());

//In general, we do not want to retry after this exception, so set retry count to 0
//and rollback the transaction
                try {
                    ut.setRollbackOnly();
                }
                catch (SecurityException se) {

//Thrown to indicate that the thread is not allowed to roll back the transaction.
                  System.out.println("Security Exception setting rollback only! " +
                                     se.getMessage());
                }
                catch (IllegalStateException ise) {
//Thrown if the current thread is not associated with a transaction.
                  System.out.println("Illegal State Exception setting rollback only! "
                                     + ise.getMessage());
                }
                catch (SystemException sye) {
          //Thrown if the transaction manager encounters an unexpected error condition
                    System.out.println("System Exception setting rollback only! " +
                                       sye.getMessage());
                }
                retryCount=0;
            }
            catch (com.ibm.websphere.ce.cm.StaleConnectionException sc) {

   // This exception indicates that the connection to the database is no longer valid.
```

```
   //Rollback the transaction, then retry several times to attempt to obtain a valid
   //connection, display an error message if the connection still can not be obtained.

               System.out.println("Stale Connection Exception during get connection or
process SQL: " + sc.getMessage());
               try {
                  ut.setRollbackOnly();
               }
               catch (SecurityException se) {

     //Thrown to indicate that the thread is not allowed to roll back the transaction.
                  System.out.println("Security Exception setting rollback only! " +
                                     se.getMessage());
               }
               catch (IllegalStateException ise) {

     //Thrown if the current thread is not associated with a transaction.
                  System.out.println("Illegal State Exception setting rollback only! "
                                     + ise.getMessage());
                }
                catch (SystemException sye) {
    //Thrown if the transaction manager encounters an unexpected error condition
                   System.out.println("System Exception setting rollback only! " +
                                      sye.getMessage());
                }
                if (--retryCount == 0) {
                   System.out.println("Five stale connection exceptions, displaying
error page.");
                }
             }
             catch (SQLException sq) {
              System.out.println("SQL Exception during get connection or process SQL: "
                                 + sq.getMessage());

//In general, we do not want to retry after this exception, so set retry count to 0
//and rollback the transaction
               try {
                  ut.setRollbackOnly();
               }
               catch (SecurityException se) {

     //Thrown to indicate that the thread is not allowed to roll back the transaction.
                 System.out.println("Security Exception setting rollback only! " +
                                    se.getMessage());
                }
                catch (IllegalStateException ise) {
     //Thrown if the current thread is not associated with a transaction.
                  System.out.println("Illegal State Exception setting rollback only! "
                                     + ise.getMessage());
                }
                catch (SystemException sye) {
     //Thrown if the transaction manager encounters an unexpected error condition
                   System.out.println("System Exception setting rollback only! " +
                                      sye.getMessage());
                }
                retryCount=0;
             }
             catch (NotSupportedException nse) {

  //Thrown by UserTransaction begin method if the thread is already associated with a
  //transaction and the Transaction Manager implementation does not support nested
  //transactions.
               System.out.println("NotSupportedException on User Transaction begin: "
                                  + nse.getMessage());
             }
             catch (SystemException se) {
```

```
   //Thrown if the transaction manager encounters an unexpected error condition
               System.out.println("SystemException in User Transaction: " +
                     se.getMessage());
            }
            catch (Exception e) {
               System.out.println("Exception in get connection or process SQL: " +
                                  e.getMessage());
//In general, we do not want to retry after this exception, so set retry count to 5
//and rollback the transaction
               try {
                  ut.setRollbackOnly();
               }
               catch (SecurityException se) {
  //Thrown to indicate that the thread is not allowed to roll back the transaction.
                  System.out.println("Security Exception setting rollback only! " +
                                     se.getMessage());
               }
               catch (IllegalStateException ise) {
  //Thrown if the current thread is not associated with a transaction.
                  System.out.println("Illegal State Exception setting rollback only! "
                                     + ise.getMessage());
                }
                catch (SystemException sye) {
  //Thrown if the transaction manager encounters an unexpected error condition
                    System.out.println("System Exception setting rollback only! " +
                                       sye.getMessage());
               }
               retryCount=0;
            }
             finally {

               // Always close the connection in a finally statement to ensure proper
               // closure in all cases. Closing the connection does not close and
               // actual connection, but releases it back to the pool for reuse.

               if (rs != null) {
                  try {
                     rs.close();
                  }
                  catch (Exception e) {
                     System.out.println("Close Resultset Exception: " +
                                        e.getMessage());
                  }
               }
               if (stmt != null) {
                  try {
                     stmt.close();
                  }
                  catch (Exception e) {
                     System.out.println("Close Statement Exception: " +
                                        e.getMessage());
                  }
               }
               if (conn != null) {
                  try {
                     conn.close();
                  }
                  catch (Exception e) {
                     System.out.println("Close connection exception: " +
                                        e.getMessage());
                   }
               }
               try {
                   ut.commit();
               }
               catch (RollbackException re) {
```

```
 //Thrown to indicate that the transaction has been rolled back rather than committed.
                   System.out.println("User Transaction Rolled back! " +
                                      re.getMessage());
                }
                catch (SecurityException se) {
 //Thrown to indicate that the thread is not allowed to commit the transaction.
                System.out.println("Security Exception thrown on transaction commit: "
                                   + se.getMessage());
                }
                catch (IllegalStateException ise) {
 //Thrown if the current thread is not associated with a transaction.
                   System.out.println("Illegal State Exception thrown on transaction
commit: " + ise.getMessage());
                }
                catch (SystemException sye) {
 //Thrown if the transaction manager encounters an unexpected error condition
                  System.out.println("System Exception thrown on transaction commit: "
                                     + sye.getMessage());
                }
                catch (Exception e) {
                   System.out.println("Exception thrown on transaction commit: " +
                                      e.getMessage());
                }
             }
        } while ( error==true && retryCount > 0 );

        // Prepare and return HTML response, prevent dynamic content from being cached
        // on browsers.
        res.setContentType("text/html");
        res.setHeader("Pragma", "no-cache");
        res.setHeader("Cache-Control", "no-cache");
        res.setDateHeader("Expires", 0);
        try {
            ServletOutputStream out = res.getOutputStream();
            out.println("<HTML>");
            out.println("<HEAD><TITLE>" + title + "</TITLE></HEAD>");
            out.println("<BODY>");
            if (error==true) {
                out.println("<H1>There was an error processing this request.</H1>
Please try the request again, or contact " +
" the <a href='mailto:sysadmin@my.com'>System Administrator</a>");
            }
            else if (employeeList.isEmpty()) {
               out.println("<H1>Employee List is Empty</H1>");
            }
            else {
               out.println("<H1>Employee List </H1>");
               for (int i = 0; i < employeeList.size(); i++) {
                  out.println(employeeList.elementAt(i) + "<BR>");
               }
            }
            out.println("</BODY></HTML>");
            out.close();
        }
        catch (IOException e) {
           System.out.println("HTML response exception: " + e.getMessage());
        }
    }
}
```

Appendix G - Code Sample: Session bean with container-managed transaction

WebSphereSamples\ConnPool\ShowEmployeesCMTBean.java

```
//====================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//====================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

import java.rmi.RemoteException;
import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

/*****************************************************************************
* This bean is designed to demonstrate Database Connections in a
* Container Managed Transaction Session Bean.  Its transaction attribute
*
* should be set to TX_REQUIRED or TX_REQUIRES_NEW.
*
*****************************************************************************
*/
public class ShowEmployeesCMTBean implements SessionBean {
       private javax.ejb.SessionContext mySessionCtx = null;
       final static long serialVersionUID = 3206093459760846163L;

       private javax.sql.DataSource ds;

//****************************************************************************
//* ejbActivate calls the getDS method, which does the JNDI lookup for the DataSource.
//* Because the DataSource lookup is in a separate method, we can also invoke it from
//* the getEmployees method in the case where the DataSource field is null.
//****************************************************************************
public void ejbActivate() throws java.rmi.RemoteException {
       getDS();
}
/**
 * ejbCreate method
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.RemoteException {}
/**
 * ejbPassivate method
 * @exception java.rmi.RemoteException
 */
public void ejbPassivate() throws java.rmi.RemoteException {}
/**
 * ejbRemove method
 * @exception java.rmi.RemoteException
 */
public void ejbRemove() throws java.rmi.RemoteException {}
```

```
//*****************************************************************************
//* The getEmployees method runs the database query to retrieve the employees.
//* The getDS method is only called if the DataSource variable is null.
//* Because this session bean uses Container Managed Transactions, it cannot retry the
//* transaction on a StaleConnectionException.  However, it can throw an exception to
//* its client indicating that the operation is retriable.                    *
//*****************************************************************************

public Vector getEmployees() throws
com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException, SQLException,
RetryableConnectionException {
   Connection conn  = null;
   Statement stmt = null;
   ResultSet rs = null;
   Vector employeeList = new Vector();

   if (ds == null) getDS();

   try {
      // Get a Connection object conn using the DataSource factory.
      conn = ds.getConnection();
      // Run DB query using standard JDBC coding.
      stmt = conn.createStatement();
      String query   = "Select FirstNme, MidInit, LastName " +
                       "from Employee ORDER BY LastName";
      rs   = stmt.executeQuery(query);
      while (rs.next()) {
         employeeList.addElement(rs.getString(3) + ", " + rs.getString(1) + " " +
                                 rs.getString(2));
      }
   }
   catch (com.ibm.websphere.ce.cm.StaleConnectionException se) {

// This exception indicates that the connection to the database is no longer valid.
// Rollback the transaction, and throw an exception to the client indicating they
// can retry the transaction if desired.

System.out.println("Stale Connection Exception during get connection or process SQL: "
+ se.getMessage());

System.out.println("Rolling back transaction and throwing
RetryableConnectionException");

      mySessionCtx.setRollbackOnly();
      throw new RetryableConnectionException(se.toString());
      }
      catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {

// This exception is thrown if a connection can not be obtained from the
// pool within a configurable amount of time.  Frequent occurrences of
// this exception indicate an incorrectly tuned connection pool

System.out.println("Connection Wait Timeout Exception during get connection or process
SQL: " + cw.getMessage());
       throw cw;
      }
      catch (SQLException sq) {

//Throwing a remote exception will automatically roll back the container managed
//transaction

        System.out.println("SQL Exception during get connection or process SQL: " +
                           sq.getMessage());
        throw sq;
      }
      finally {
```

```
         // Always close the connection in a finally statement to ensure proper
         // closure in all cases. Closing the connection does not close and
         // actual connection, but releases it back to the pool for reuse.

         if (rs != null) {
            try {
               rs.close();
            }
            catch (Exception e) {
               System.out.println("Close Resultset Exception: " +
                                   e.getMessage());
            }
         }
         if (stmt != null) {
            try {
               stmt.close();
            }
            catch (Exception e) {
               System.out.println("Close Statement Exception: " +
                                  e.getMessage());
            }
         }
         if (conn != null) {
            try {
               conn.close();
            }
            catch (Exception e) {
               System.out.println("Close connection exception: " +
                                  e.getMessage());
            }
         }
      }
      return employeeList;
}
/**
 * getSessionContext method
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
       return mySessionCtx;
}
//*****************************************************************************
//* The getDS method performs the JNDI lookup for the DataSource.             *
//* This method is called from ejbActivate, and from getEmployees if the DataSource
//* object is null.                                                           *
//*****************************************************************************

private void getDS() {
       try {
       // Note the new Initial Context Factory interface available in WebSphere 4.0

              Hashtable parms = new Hashtable();
              parms.put(Context.INITIAL_CONTEXT_FACTORY,
                           "com.ibm.websphere.naming.WsnInitialContextFactory");
              InitialContext ctx = new InitialContext(parms);
              // Perform a naming service lookup to get the DataSource object.
              ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
       }
       catch (Exception e) {
              System.out.println("Naming service exception: " + e.getMessage());
              e.printStackTrace();
       }
}
/**
 * setSessionContext method
 * @param ctx javax.ejb.SessionContext
```

```
 * @exception java.rmi.RemoteException
 */
public void setSessionContext(javax.ejb.SessionContext ctx) throws
java.rmi.RemoteException {
      mySessionCtx = ctx;
}
}
```

WebSphereSamples\ConnPool\ShowEmployeesCMTHome.java

```
//====================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//====================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * This is a Home interface for the Session Bean
 */
public interface ShowEmployeesCMTHome extends javax.ejb.EJBHome {

/**
 * create method for a session bean
 * @return WebSphereSamples.ConnPool.ShowEmployeesCMT
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
WebSphereSamples.ConnPool.ShowEmployeesCMT create() throws javax.ejb.CreateException,
java.rmi.RemoteException;
}
```

WebSphereSamples\ConnPool\ShowEmployeesCMT.java

```
//====================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//====================END_PROLOG=========================================

package WebSphereSamples.ConnPool;
```

```
/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface ShowEmployeesCMT extends javax.ejb.EJBObject {

/**
 *
 * @return java.util.Vector
 */
java.util.Vector getEmployees() throws java.sql.SQLException,
java.rmi.RemoteException, com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException,
WebSphereSamples.ConnPool.RetryableConnectionException;
}
```

WebSphereSamples\ConnPool\RetryableConnectionException.java

```
//====================START_PROLOG============================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//====================END_PROLOG==============================================

package WebSphereSamples.ConnPool;

/**
 * Exception indicating that the operation can be retried
 * Creation date: (4/2/2001 10:48:08 AM)
 * @author: Administrator
 */
public class RetryableConnectionException extends Exception {
/**
 * RetryableConnectionException constructor.
 */
public RetryableConnectionException() {
      super();
}
/**
 * RetryableConnectionException constructor.
 * @param s java.lang.String
 */
public RetryableConnectionException(String s) {
      super(s);
}
}
```

Appendix H - Code Sample: Session bean with bean-managed transaction

WebSphereSamples\ConnPool\ShowEmployeesBMTBean.java

```
//====================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//====================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

import java.rmi.RemoteException;
import java.util.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;
import javax.transaction.*;

/*******************************************************************************
* This bean is designed to demonstrate Database Connections in a
* Bean-Managed Transaction Session Bean.  Its transaction attribute
* should be set to TX_BEANMANAGED.
*******************************************************************************
*/
public class ShowEmployeesBMTBean implements SessionBean {
       private javax.ejb.SessionContext mySessionCtx = null;
       final static long serialVersionUID = 320609345976084616 3L;

       private javax.sql.DataSource ds;

       private javax.transaction.UserTransaction userTran;

//*******************************************************************************
//* ejbActivate calls the getDS method, which makes the JNDI lookup for the DataSource
//* Because the DataSource lookup is in a separate method, we can also invoke it from
//* the getEmployees method in the case where the DataSource field is null.
//*******************************************************************************
public void ejbActivate() throws java.rmi.RemoteException {
       getDS();
}
/**
 * ejbCreate method
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
public void ejbCreate() throws javax.ejb.CreateException, java.rmi.RemoteException {}
/**
 * ejbPassivate method
 * @exception java.rmi.RemoteException
```

```
 */
public void ejbPassivate() throws java.rmi.RemoteException {}
/**
 * ejbRemove method
 * @exception java.rmi.RemoteException
 */
public void ejbRemove() throws java.rmi.RemoteException {}

//*****************************************************************************
//* The getEmployees method runs the database query to retrieve the employees.
//* The getDS method is only called if the DataSource or userTran variables are null.
//* If a StaleConnectionException occurs, the bean retries the transaction 5 times,
//* then throws an EJBException.                                              *
//*****************************************************************************

public Vector getEmployees() throws EJBException {
   Connection conn  = null;
   Statement stmt = null;
   ResultSet rs = null;
   Vector employeeList = new Vector();

   // Set retryCount to the number of times you would like to retry after a
   //StaleConnectionException
   int retryCount = 5;

   // If the Database code processes successfully, we will set error = false
   boolean error = true;

   if (ds == null || userTran == null) getDS();
      do {
         try {
              //try/catch block for UserTransaction work
              //Begin the transaction
            userTran.begin();
            try {
                 //try/catch block for database work
                 //Get a Connection object conn using the DataSource factory.
               conn = ds.getConnection();
               // Run DB query using standard JDBC coding.
               stmt = conn.createStatement();
               String query   = "Select FirstNme, MidInit, LastName " +
                                "from Employee ORDER BY LastName";
               rs   = stmt.executeQuery(query);
               while (rs.next()) {
                  employeeList.addElement(rs.getString(3) + ", " +
                                          rs.getString(1) + " " + rs.getString(2));
               }
//Set error to false, as all database operations are successfully completed
               error = false;
            }
            catch (com.ibm.websphere.ce.cm.StaleConnectionException se) {

// This exception indicates that the connection to the database is no longer valid.
// Rollback the transaction, and throw an exception to the client indicating they
// can retry the transaction if desired.

System.out.println("Stale Connection Exception during get connection or process SQL: "
+ se.getMessage());
            userTran.rollback();
            if (--retryCount == 0) {

//If we have already retried the requested number of times, throw an EJBException.
               throw new EJBException("Transaction Failure: " + se.toString());
            }
            else {
               System.out.println("Retrying transaction, retryCount = " +
                                  retryCount);
```

```
            }
         }
         catch (com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException cw) {

// This exception is thrown if a connection can not be obtained from the
// pool within a configurable amount of time.  Frequent occurrences of
// this exception indicate an incorrectly tuned connection pool

            System.out.println("Connection Wait Timeout Exception during get
connection or process SQL: " + cw.getMessage());
            userTran.rollback();
            throw new EJBException("Transaction failure: " + cw.getMessage());
         }
         catch (SQLException sq) {
              // This catch handles all other SQL Exceptions
       System.out.println("SQL Exception during get connection or process SQL: " +
                          sq.getMessage());
            userTran.rollback();
            throw new EJBException("Transaction failure: " + sq.getMessage());
         }
         finally {
// Always close the connection in a finally statement to ensure proper
// closure in all cases. Closing the connection does not close and
// actual connection, but releases it back to the pool for reuse.

            if (rs != null) {
               try {
                  rs.close();
               }
               catch (Exception e) {
                  System.out.println("Close Resultset Exception: " + e.getMessage());
               }
            }
            if (stmt != null) {
               try {
                  stmt.close();
            }
            catch (Exception e) {
               System.out.println("Close Statement Exception: " + e.getMessage());
            }
         }
         if (conn != null) {
            try {
               conn.close();
            }
            catch (Exception e) {
               System.out.println("Close connection exception: " + e.getMessage());
            }
         }
      }
      if (!error) {
         //Database work completed successfully, commit the transaction
         userTran.commit();
      }
      //Catch UserTransaction exceptions
      }
      catch (NotSupportedException nse) {

//Thrown by UserTransaction begin method if the thread is already associated with a
//transaction and the Transaction Manager implementation does not support nested
//transactions.

         System.out.println("NotSupportedException on User Transaction begin: " +
                            nse.getMessage());
         throw new EJBException("Transaction failure: " + nse.getMessage());
      }
      catch (RollbackException re) {
```

```
//Thrown to indicate that the transaction has been rolled back rather than committed.
         System.out.println("User Transaction Rolled back! " + re.getMessage());
         throw new EJBException("Transaction failure: " + re.getMessage());
      }
      catch (SystemException se) {
         //Thrown if the transaction manager encounters an unexpected error condition
         System.out.println("SystemException in User Transaction: "+ se.getMessage());
         throw new EJBException("Transaction failure: " + se.getMessage());
      }
      catch (Exception e) {
         //Handle any generic or unexpected Exceptions
         System.out.println("Exception in User Transaction: " + e.getMessage());
         throw new EJBException("Transaction failure: " + e.getMessage());
      }
   }
   while (error);
   return employeeList;
}
/**
 * getSessionContext method comment
 * @return javax.ejb.SessionContext
 */
public javax.ejb.SessionContext getSessionContext() {
       return mySessionCtx;
}

//*****************************************************************************
//* The getDS method performs the JNDI lookup for the DataSource.
//* This method is called from ejbActivate, and from getEmployees if the DataSource
//* object is null.
//*****************************************************************************
private void getDS() {
   try {
      // Note the new Initial Context Factory interface available in WebSphere 4.0
      Hashtable parms = new Hashtable();
      parms.put(Context.INITIAL_CONTEXT_FACTORY,
         "com.ibm.websphere.naming.WsnInitialContextFactory");
      InitialContext ctx = new InitialContext(parms);

      // Perform a naming service lookup to get the DataSource object.
      ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
      //Create the UserTransaction object
      userTran = mySessionCtx.getUserTransaction();
   }
   catch (Exception e) {
      System.out.println("Naming service exception: " + e.getMessage());
      e.printStackTrace();
}
}
/**
 * setSessionContext method
 * @param ctx javax.ejb.SessionContext
 * @exception java.rmi.RemoteException
 */
public void setSessionContext(javax.ejb.SessionContext ctx) throws
java.rmi.RemoteException {
       mySessionCtx = ctx;
}
}
```

WebSphereSamples\ConnPool\ShowEmployeesBMTHome.java

```
//====================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
```

```
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * This is a Home interface for the Session Bean
 */
public interface ShowEmployeesBMTHome extends javax.ejb.EJBHome {

/**
 * create method for a session bean
 * @return WebSphereSamples.ConnPool.ShowEmployeesBMT
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
WebSphereSamples.ConnPool.ShowEmployeesBMT create() throws javax.ejb.CreateException,
java.rmi.RemoteException;
}
```

WebSphereSamples\ConnPool\ShowEmployeesBMT.java

```
//==================START_PROLOG=======================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG=========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface ShowEmployeesBMT extends javax.ejb.EJBObject {

/**
 *
 * @return java.util.Vector
 */
java.util.Vector getEmployees() throws java.rmi.RemoteException,
javax.ejb.EJBException;
}
```

Appendix J - Code Sample: Entity bean with bean-managed persistence (container-managed transaction)

\WebSphereSamples\ConnPool\EmployeeBMPBean.java

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

import java.rmi.RemoteException;
import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

/**
 * This is an Entity Bean class with five BMP fields
 * String firstName, String lastName, String middleInit
 * String empNo, int edLevel
 */
public class EmployeeBMPBean implements EntityBean {
       private javax.ejb.EntityContext entityContext = null;
       final static long serialVersionUID = 3206093459760846163L;

       private java.lang.String firstName;
       private java.lang.String lastName;
       private String middleInit;
       private javax.sql.DataSource ds;
       private java.lang.String empNo;
       private int edLevel;
/**
 * ejbActivate method
 * @exception java.rmi.RemoteException
 * ejbActivate calls getDS(), which perfoms the
 * JNDI lookup for the datasource.
 */
public void ejbActivate() throws java.rmi.RemoteException {
       getDS();
}
/**
 * ejbCreate method for a BMP entity bean
 * @return WebSphereSamples.ConnPool.EmployeeBMPKey
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception javax.ejb.CreateException
 * @exception java.rmi.RemoteException
 */
```

```
public WebSphereSamples.ConnPool.EmployeeBMPKey ejbCreate(String empNo, String
firstName, String lastName, String middleInit, int edLevel) throws
javax.ejb.CreateException, java.rmi.RemoteException {

   Connection conn = null;
   PreparedStatement ps = null;

   if (ds == null) getDS();

   this.empNo = empNo;
   this.firstName = firstName;
   this.lastName = lastName;
   this.middleInit = middleInit;
   this.edLevel = edLevel;

   String sql = "insert into Employee (empNo, firstnme, midinit, lastname, edlevel)
values (?,?,?,?,?)";

   try {
      conn = ds.getConnection();
      ps = conn.prepareStatement(sql);
      ps.setString(1, empNo);
      ps.setString(2, firstName);
      ps.setString(3, middleInit);
      ps.setString(4, lastName);
      ps.setInt(5, edLevel);

      if (ps.executeUpdate() != 1){
         System.out.println("ejbCreate Failed to add user.");
         throw new CreateException("Failed to add user.");
      }
   }
   catch (com.ibm.websphere.ce.cm.StaleConnectionException se) {

// This exception indicates that the connection to the database is no longer valid.
// Rollback the transaction, and throw an exception to the client indicating they
// can retry the transaction if desired.

System.out.println("Stale Connection Exception during get connection or process SQL: "
+  se.getMessage());
      throw new CreateException(se.getMessage());
   }
   catch (SQLException sq) {
      System.out.println("SQL Exception during get connection or process SQL: " +
                         sq.getMessage());
   throw new CreateException(sq.getMessage());
   }
   finally {
      // Always close the connection in a finally statement to ensure proper
      // closure in all cases. Closing the connection does not close and
      // actual connection, but releases it back to the pool for reuse.
      if (ps != null) {
         try {
            ps.close();
         }
         catch (Exception e) {
            System.out.println("Close Statement Exception: " + e.getMessage());
         }
      }
      if (conn != null) {
         try {
            conn.close();
         }
         catch (Exception e) {
            System.out.println("Close connection exception: " + e.getMessage());
         }
      }
```

```
   }
   return new EmployeeBMPKey(this.empNo);
}
/**
 * ejbFindByPrimaryKey method
 * @return WebSphereSamples.ConnPool.EmployeeBMPKey
 * @param primaryKey WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception java.rmi.RemoteException
 * @exception javax.ejb.FinderException
 */
public WebSphereSamples.ConnPool.EmployeeBMPKey
ejbFindByPrimaryKey(WebSphereSamples.ConnPool.EmployeeBMPKey primaryKey) throws
java.rmi.RemoteException, javax.ejb.FinderException {
       loadByEmpNo(primaryKey.empNo);
       return primaryKey;
}
/**
 * ejbLoad method
 * @exception java.rmi.RemoteException
 */
public void ejbLoad() throws java.rmi.RemoteException {
       try {
              EmployeeBMPKey pk = (EmployeeBMPKey) entityContext.getPrimaryKey();
              loadByEmpNo(pk.empNo);
       } catch (FinderException fe) {
              throw new RemoteException("Cannot load Employee state from database.");
       }
}
/**
 * ejbPassivate method
 * @exception java.rmi.RemoteException
 */
public void ejbPassivate() throws java.rmi.RemoteException {}
/**
 * ejbPostCreate method for a BMP entity bean
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception java.rmi.RemoteException
 */
public void ejbPostCreate(String empNo, String firstName, String lastName, String
middleInit, int edLevel) throws java.rmi.RemoteException {}
/**
 * ejbRemove method
 * @exception java.rmi.RemoteException
 * @exception javax.ejb.RemoveException
 */
public void ejbRemove() throws java.rmi.RemoteException, javax.ejb.RemoveException {

   if (ds == null)
      GetDS();

   String sql = "delete from Employee where empNo=?";
   Connection con = null;
   PreparedStatement ps = null;

   try {
      con = ds.getConnection();
      ps = con.prepareStatement(sql);
      ps.setString(1, empNo);
      if (ps.executeUpdate() != 1){
         throw new RemoteException("Cannot remove employee: " + empNo);
      }
   }
   catch (com.ibm.websphere.ce.cm.StaleConnectionException se) {

// This exception indicates that the connection to the database is no longer valid.
// Rollback the transaction, and throw an exception to the client indicating they
// can retry the transaction if desired.
```

```
System.out.println("Stale Connection Exception during get connection or process SQL: "
+ se.getMessage());
      throw new RemoteException(se.getMessage());
   }
   catch (SQLException sq) {
      System.out.println("SQL Exception during get connection or process SQL: " +
                         sq.getMessage());
      throw new RemoteException(sq.getMessage());
   }
   finally {
      // Always close the connection in a finally statement to ensure proper
      // closure in all cases. Closing the connection does not close and
      // actual connection, but releases it back to the pool for reuse.
      if (ps != null) {
         try {
            ps.close();
         }
         catch (Exception e) {
            System.out.println("Close Statement Exception: " + e.getMessage());
         }
      }
      if (con != null) {
         try {
            con.close();
         }
         catch (Exception e) {
            System.out.println("Close connection exception: " + e.getMessage());
         }
      }
   }
}
/**
 * ejbStore method
 * @exception java.rmi.RemoteException
 */
public void ejbStore() throws java.rmi.RemoteException {

   if (ds == null)
      getDS();

   String sql = "update Employee set firstnme= ?, midinit= ?, lastname= ?, edlevel=?
where empno= ?";

   Connection con = null;
   PreparedStatement ps = null;
   int retVal=-1;
   try {
      con = ds.getConnection();
      ps = con.prepareStatement(sql);
      ps.setString(1, firstName);
      ps.setString(2, middleInit);
      ps.setString(3, lastName);
      ps.setInt(4, edLevel);
      ps.setString(5, empNo);
      retVal = ps.executeUpdate();
      if (retVal != 1){
         throw new RemoteException("Cannot store employee: " + empNo);
      }
   }
   catch (com.ibm.websphere.ce.cm.StaleConnectionException se) {

// This exception indicates that the connection to the database is no longer valid.
// Rollback the transaction, and throw an exception to the client indicating they
// can retry the transaction if desired.
```

```
System.out.println("Stale Connection Exception during get connection or process SQL: "
+ se.getMessage());
      throw new RemoteException(se.getMessage());
   }
   catch (SQLException sq) {

      System.out.println("SQL Exception during get connection or process SQL: " +
                         sq.getMessage());

      throw new RemoteException(sq.getMessage());
   }
   finally {
      // Always close the connection in a finally statement to ensure proper
      // closure in all cases. Closing the connection does not close and
      // actual connection, but releases it back to the pool for reuse.
      if (ps != null) {
         try {
            ps.close();
         }
         catch (Exception e) {
            System.out.println("Close Statement Exception: " + e.getMessage());
         }
      }
      if (con != null) {
         try {
            con.close();
         }
         catch (Exception e) {
            System.out.println("Close connection exception: " + e.getMessage());
         }
      }
   }
}
/**
 * Get the employee's edLevel
 * Creation date: (4/20/2001 3:46:22 PM)
 * @return int
 */
public int getEdLevel() {
      return edLevel;
}
/**
 * getEntityContext method
 * @return javax.ejb.EntityContext
 */
public javax.ejb.EntityContext getEntityContext() {
      return entityContext;
}
/**
 * Get the employee's first name
 * Creation date: (4/19/2001 1:34:47 PM)
 * @return java.lang.String
 */
public java.lang.String getFirstName() {
      return firstName;
}
/**
 * Get the employee's last name
 * Creation date: (4/19/2001 1:35:41 PM)
 * @return java.lang.String
 */
public java.lang.String getLastName() {
      return lastName;
}
/**
 * get the employee's middle initial
 * Creation date: (4/19/2001 1:36:15 PM)
```

```
 * @return char
 */
public String getMiddleInit() {
	return middleInit;
}
/**
 * Lookup the DataSource from JNDI
 * Creation date: (4/19/2001 3:28:15 PM)
 */
private void getDS() {
	try {
	// Note the new Initial Context Factory interface available in WebSphere 4.0

		Hashtable parms = new Hashtable();
		parms.put(Context.INITIAL_CONTEXT_FACTORY,
				"com.ibm.websphere.naming.WsnInitialContextFactory");
		InitialContext ctx = new InitialContext(parms);
		// Perform a naming service lookup to get the DataSource object.
		ds = (DataSource)ctx.lookup("java:comp/env/jdbc/SampleDB");
	}
	catch (Exception e) {
		System.out.println("Naming service exception: " + e.getMessage());
		e.printStackTrace();
	}
}
/**
 * Load the employee from the database
 * Creation date: (4/19/2001 3:44:07 PM)
 * @param empNo java.lang.String
 */
private void loadByEmpNo(String empNoKey) throws javax.ejb.FinderException{

	String sql = "select empno, firstnme, midinit, lastname, edLevel from employee
where empno = ?";
	Connection conn = null;
	PreparedStatement ps = null;
	ResultSet rs = null;

	if (ds == null) getDS();

	try {
		// Get a Connection object conn using the DataSource factory.
		conn = ds.getConnection();
		// Run DB query using standard JDBC coding.
		ps = conn.prepareStatement(sql);
		ps.setString(1, empNoKey);
		rs = ps.executeQuery();
		if (rs.next()) {
			empNo= rs.getString(1);
			firstName=rs.getString(2);
			middleInit=rs.getString(3);
			lastName=rs.getString(4);
			edLevel=rs.getInt(5);
		}
		else {
		   throw new ObjectNotFoundException("Cannot find employee number " +
							empNoKey);
		}
	}
	catch (com.ibm.websphere.ce.cm.StaleConnectionException se) {

// This exception indicates that the connection to the database is no longer valid.
// Rollback the transaction, and throw an exception to the client indicating they
// can retry the transaction if desired.

System.out.println("Stale Connection Exception during get connection or process SQL: "
+ se.getMessage());
```

```
            throw new FinderException(se.getMessage());
         }
         catch (SQLException sq) {
            System.out.println("SQL Exception during get connection or process SQL: " +
                               sq.getMessage());
            throw new FinderException(sq.getMessage());
         }
         finally {
                // Always close the connection in a finally statement to ensure proper
                // closure in all cases. Closing the connection does not close and
                // actual connection, but releases it back to the pool for reuse.
            if (rs != null) {
               try {
                  Rs.close();
               }
               catch (Exception e) {
                  System.out.println("Close Resultset Exception: " + e.getMessage());
               }
            }
            if (ps != null) {
               try {
                  ps.close();
               }
               catch (Exception e) {
                  System.out.println("Close Statement Exception: " + e.getMessage());
               }
            }
            if (conn != null) {
               try {
                  conn.close();
               }
               catch (Exception e) {
                  System.out.println("Close connection exception: " + e.getMessage());
               }
           }
        }
}
/**
 * set the employee's education level
 * Creation date: (4/20/2001 3:46:22 PM)
 * @param newEdLevel int
 */
public void setEdLevel(int newEdLevel) {
       edLevel = newEdLevel;
}
/**
 * setEntityContext method
 * @param ctx javax.ejb.EntityContext
 * @exception java.rmi.RemoteException
 */
public void setEntityContext(javax.ejb.EntityContext ctx) throws
java.rmi.RemoteException {
       entityContext = ctx;
}
/**
 * set the employee's first name
 * Creation date: (4/19/2001 1:34:47 PM)
 * @param newFirstName java.lang.String
 */
public void setFirstName(java.lang.String newFirstName) {
       firstName = newFirstName;
}
/**
 * set the employee's last name
 * Creation date: (4/19/2001 1:35:41 PM)
 * @param newLastName java.lang.String
 */
```

```
public void setLastName(java.lang.String newLastName) {
	lastName = newLastName;
}
/**
 * set the employee's middle initial
 * Creation date: (4/19/2001 1:36:15 PM)
 * @param newMiddleInit char
 */
public void setMiddleInit(String newMiddleInit) {
	middleInit = newMiddleInit;
}
/**
 * unsetEntityContext method
 * @exception java.rmi.RemoteException
 */
public void unsetEntityContext() throws java.rmi.RemoteException {
	entityContext = null;
}
}
```

\WebSphereSamples\ConnPool\EmplyeeBMPHome.java

```
//====================START_PROLOG============================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//====================END_PROLOG==============================================

package WebSphereSamples.ConnPool;

/**
 * This is a Home interface for the Entity Bean
 */
public interface EmployeeBMPHome extends javax.ejb.EJBHome {

/**
 *
 * @return WebSphereSamples.ConnPool.EmployeeBMP
 * @param empNo java.lang.String
 * @param firstName java.lang.String
 * @param lastName java.lang.String
 * @param middleInit java.lang.String
 * @param edLevel int
 */
WebSphereSamples.ConnPool.EmployeeBMP create(java.lang.String empNo, java.lang.String
firstName, java.lang.String lastName, java.lang.String middleInit, int edLevel) throws
javax.ejb.CreateException, java.rmi.RemoteException;
/**
 * findByPrimaryKey method comment
 * @return WebSphereSamples.ConnPool.EmployeeBMP
 * @param key WebSphereSamples.ConnPool.EmployeeBMPKey
 * @exception java.rmi.RemoteException
 * @exception javax.ejb.FinderException
```

```
 */
WebSphereSamples.ConnPool.EmployeeBMP
findByPrimaryKey(WebSphereSamples.ConnPool.EmployeeBMPKey key) throws
java.rmi.RemoteException, javax.ejb.FinderException;
}
```

\WebSphereSamples\ConnPool\EmployeeBMP.java

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
 * This is an Enterprise Java Bean Remote Interface
 */
public interface EmployeeBMP extends javax.ejb.EJBObject {

/**
 *
 * @return int
  */
int getEdLevel() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getFirstName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getLastName() throws java.rmi.RemoteException;
/**
 *
 * @return java.lang.String
 */
java.lang.String getMiddleInit() throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newEdLevel int
 */
void setEdLevel(int newEdLevel) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newFirstName java.lang.String
 */
void setFirstName(java.lang.String newFirstName) throws java.rmi.RemoteException;
/**
 *
```

```
 * @return void
 * @param newLastName java.lang.String
 */
void setLastName(java.lang.String newLastName) throws java.rmi.RemoteException;
/**
 *
 * @return void
 * @param newMiddleInit java.lang.String
 */
void setMiddleInit(java.lang.String newMiddleInit) throws java.rmi.RemoteException;
}
```

\WebSphereSamples\ConnPool\EmployeeBMPKey.java

```
//==================START_PROLOG=========================================
//
//   5630-A23, 5630-A22,
//   (C) COPYRIGHT International Business Machines Corp. 1997
//   All Rights Reserved
//   Licensed Materials - Property of IBM
//   US Government Users Restricted Rights - Use, duplication or
//   disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
//
//   IBM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING
//   ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
//   PURPOSE. IN NO EVENT SHALL IBM BE LIABLE FOR ANY SPECIAL, INDIRECT OR
//   CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF
//   USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
//   OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE
//   OR PERFORMANCE OF THIS SOFTWARE.
//
//==================END_PROLOG===========================================

package WebSphereSamples.ConnPool;

/**
* This is a Primary Key Class for the Entity Bean
**/
public class EmployeeBMPKey implements java.io.Serializable {
       public String empNo;
       final static long serialVersionUID = 3206093459760846163L;

/**
* EmployeeBMPKey() constructor
*/
public EmployeeBMPKey()  {
}
/**
* EmployeeBMPKey(String key) constructor
*/
public EmployeeBMPKey(String key)  {
       empNo = key;
}
/**
* equals method
* - user must provide a proper implementation for the equal method. The generated
*   method assumes the key is a String object.
*/
public boolean equals (Object o)  {
       if (o instanceof EmployeeBMPKey)
              return empNo.equals(((EmployeeBMPKey)o).empNo);
       else
              return false;
}
/**
* hashCode method
* - user must provide a proper implementation for the hashCode method. The generated
*    method assumes the key is a String object.
```

```
*/
public int hashCode ()  {
       return empNo.hashCode();
}
}
```

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web

sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, or other countries, or both:

- DB2
- IBM
- WebSphere

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows NT is a trademark of Microsoft Corporation in the United States, other countries, or both.

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.