Sun Microsystems Inc.

JDBC 2.0 Standard Extension API

This is the final specification of the JDBC^{TM} 2.0 Standard Extension API. The JDBC API is the Java^{TM} standard call-level API for data access.

Please send technical comments on this specification to:

jdbc@eng.sun.com

Please send product and business questions to:

jdbc-business@eng.sun.com

Version 1.0

Seth White and Mark Hapner

December 7, 1998 3:18 pm

Contents

1 Introduction 4

2 Goals 5

3 Overview of Features 6

4 Interfaces and Classes 7

5 JNDI and the JDBC API 11

6 Connection Pooling 16

7 Distributed Transactions 24

8 Rowsets 35

9 Rowset implementations 38

Appendix A: Implementing a JDBC Data Source 46

Appendix B: Implementing Connection Pooling 50

Appendix C: Implementing RowSetReader 52

Appendix D: Implementing RowSetWriter 55

Appendix E: RowSet Implementation Classes 60

Appendix F: Change History 61

1 Introduction

1.1 Preface

This document contains the final specification of the JDBC 2.0 Standard Extension API.

1.2 Target audience

This specification is intended for everyone who has an interest in JDBC technology, including Java developers writing applications using the JDBC API, JDBC driver vendors, and other software vendors who plan to provide support for JDBC in their products.

1.3 Terminology

In this document we refer to the initial release of the JDBC API as the JDBC 1.0 API. The second release of the JDBC API, which is partially described in this document, is referred to as the JDBC 2.0 API.

1.4 Background

Over the past year and a half, beginning in July 1997, Java Software, our industry partners, and the Java developer community have worked together to define the JDBC 2.0 API. The JDBC 2.0 API has been separated into two parts: the JDBC 2.0 Core API and the JDBC 2.0 Standard Extension API. At the time of this writing the JDBC 2.0 Core API has been made final and is included as a core part of the Java platform as specified in the JDK 1.2. This document contains the specification of the JDBC 2.0 Standard Extension API. See the document entitled *JDBC 2.0 API* for an overview of the JDBC 2.0 API and a detailed treatment of the JDBC 2.0 Core API.

1.5 Acknowledgments

We would like to thank everyone who has worked on or used JDBC technology for helping to make the JDBC API successful. We especially thank all those who sent us their comments and suggestions regarding this specification during the API review process.

2 Goals

This chapter lists specific goals for the JDBC 2.0 Standard Extension API. General goals for the JDBC API are listed in the JDBC 2.0 API specification.

2.1 Keep the JDBC Core API small and focused.

Keeping the JDBC Core API small and focused will help to preserve the ease-of-use of the JDBC API as a whole. Factoring the JDBC API into two parts—the JDBC Core API and the JDBC Standard Extension API—lets Java developers first learn the JDBC Core API, which provides sufficient functionality to begin using JDBC, and then move on to the JDBC Standard Extension API when needed.

2.2 Package JDBC in a way that is compatible with other standard extensions.

There are several important components of the Java platform which are related to the JDBC API that are themselves packaged as standard extensions. These include the Java Naming and Directory Interface (JNDI) as well as the Java Transaction API (JTA). The JDBC Standard Extension API should contain all of the functionality needed to enable use of the JDBC API with these other standard extensions.

2.3 Provide packaging for specialized data access facilities.

There are many aspects of data access that have not yet been fully addressed by the JDBC API. These include on-line analytical processing (OLAP), data warehousing, heterogeneous database access, bulk loading, etc. Creating the JDBC Standard Extension API will allow these more specialized data access facilities to easily be added to the JDBC API in the future, if desired.

3 Overview of Features

This chapter gives an overview of the contents of the JDBC 2.0 Standard Extension API.

3.1 JNDI for naming databases

The Java Naming and Directory Interface (JNDI) can be used in addition to the JDBC driver manager to manage data sources and connections. When an application uses JNDI, it specifies a logical name that identifies a particular database instance and JDBC driver for accessing that database. This has the advantage of making the application code independent of a particular JDBC driver and JDBC URL.

3.2 Connection Pooling

The JDBC 2.0 Standard Extension API specifies ‘hooks’ that allow connection pooling to be implemented on top of the JDBC driver layer. This allows for a single connection cache that spans the different JDBC drivers that may be in use. Since creating and destroying database connections is expensive, connection pooling is important for achieving good performance, especially for server applications.

3.3 Distributed transaction support

Support for distributed transactions allows a JDBC driver to support the standard 2-phase commit protocol used by the Java Transaction API (JTA). JDBC driver support for distributed transactions allows developers to write Enterprise JavaBeans that are transactional across multiple DBMS servers.

3.4 Rowsets

As its name implies, a rowset encapsulates a set of rows. A rowset may or may not maintain an open database connection. When a rowset is ‘disconnected’ from its data source, updates performed on the rowset are propagated to the underlying database using an optimistic concurrency control algorithm.

Rowsets add support to the JDBC API for the JavaBeans component model. A rowset object is a Java Bean. A rowset implementation may be serializable. Rowsets can be created at design time and used in conjunction with other JavaBeans components in a visual JavaBeans builder tool to construct an application.

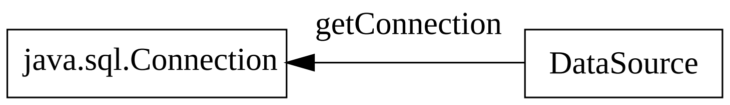
4 Interfaces and Classes

The JDBC 2.0 Standard Extension API is contained in the `javax.sql` package. A list of the classes and interfaces that are found in the `javax.sql` package is given below. Interfaces are listed in normal type, while classes are listed in bold. See the separate API documentation for a complete description of the new classes and interfaces.

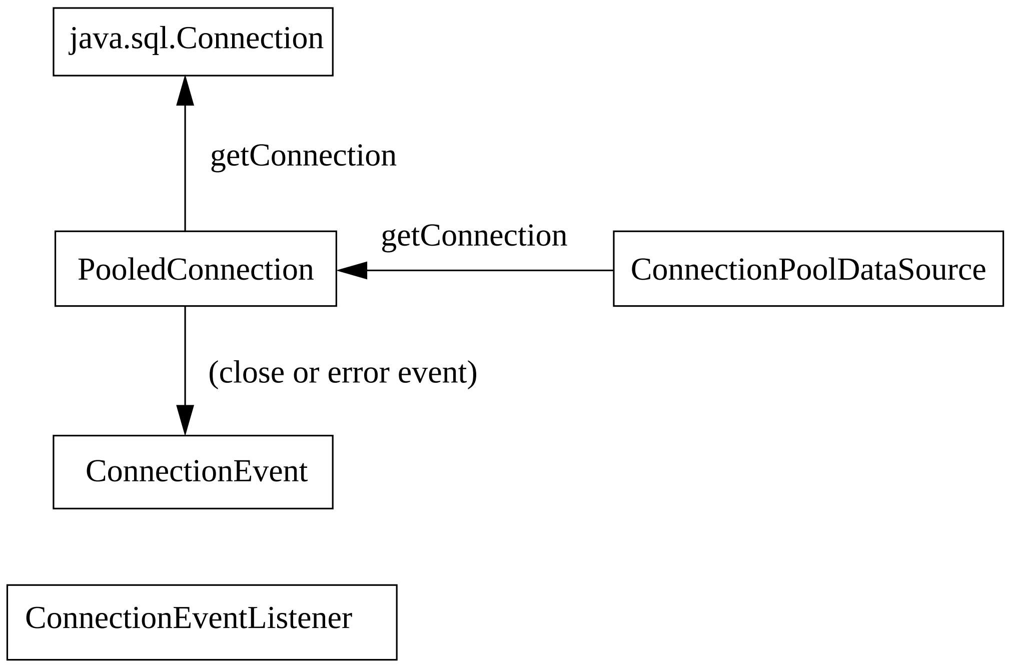
javax.sql.ConnectionEvent
javax.sql.ConnectionEventListener
javax.sql.ConnectionPoolDataSource
javax.sql.DataSource
javax.sql.PooledConnection
javax.sql.RowSet
javax.sql.RowSetEvent
javax.sql.RowSetInternal
javax.sql.RowSetListener
javax.sql.RowSetMetaData
javax.sql.RowSetReader
javax.sql.RowSetWriter
javax.sql.XAConnection
javax.sql.XADataSource

The figures below group the new classes and interfaces according to their basic areas of functionality: JDBC and JNDI, connection pooling, distributed transactions, and rowsets; and illustrate some of the important relationships between types.

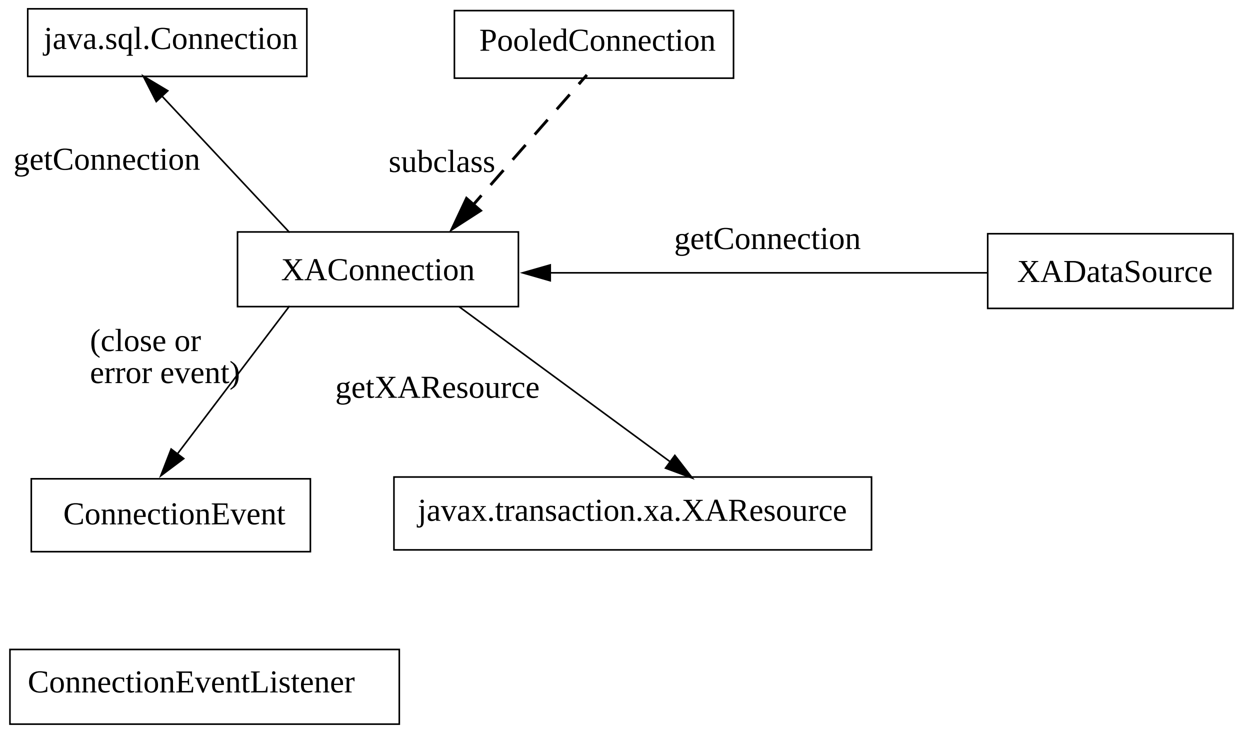
JDBC and JNDI



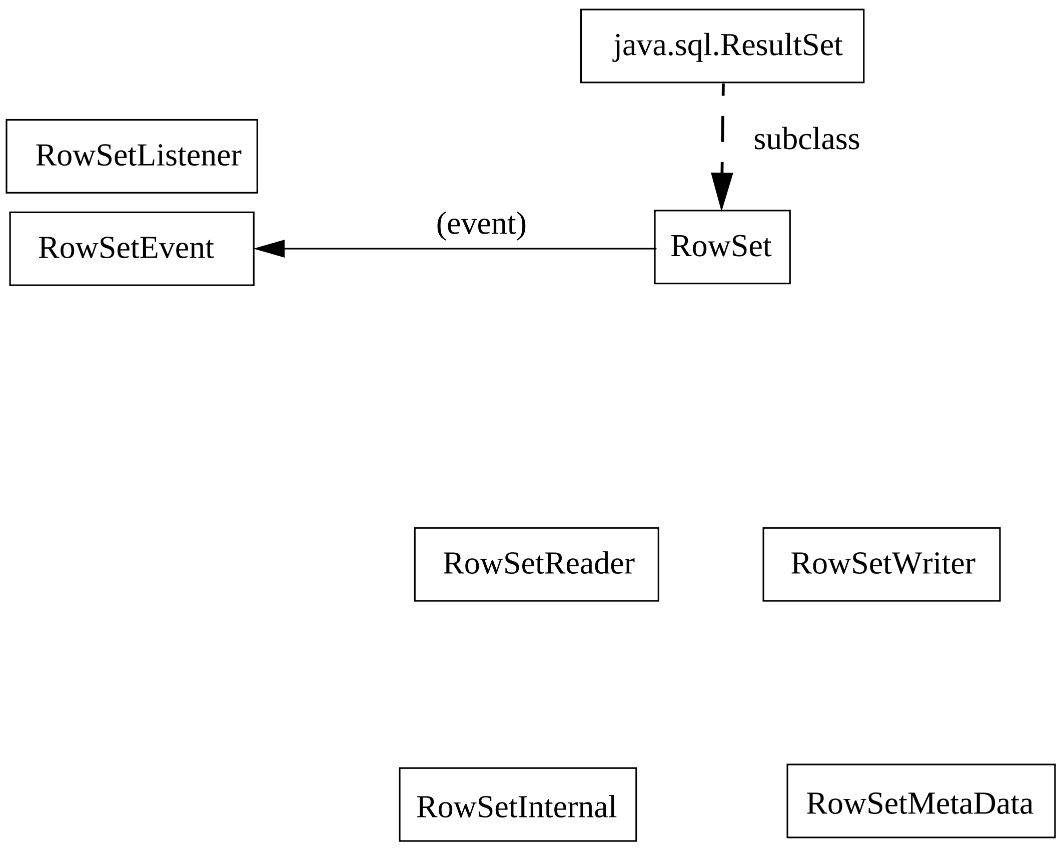
Connection Pooling



Distributed Transactions



RowSets


java.sql.ResultSet
subclass
RowSetListener
RowSetEvent
(event)
RowSet
RowSetReader
RowSetWriter
RowSetInternal
RowSetMetaData

5 JNDI and the JDBC API

This chapter describes how to use the Java Naming and Directory Interface (JNDI) with the JDBC 2.0 API. JNDI provides a uniform way for applications to find and access remote services over the network. JDBC applications are interested specifically in accessing database services, but in general a remote service accessed via JNDI could be any enterprise service, including a messaging service or an application specific service. As we shall see, using JNDI makes JDBC applications easier to manage.

5.1 Motivation

One way for an application to connect to a database is by using the JDBC driver manager. Under this approach, the JDBC driver that will be used to create a database connection must first be registered with the JDBC driver manager. For example, when the driver class includes an appropriate static initializer, the statement below loads and registers the JDBC driver implemented by the `SomeJDBCDriverClassName` class.

```
Class.forName(“SomeJDBCDriverClassName”);
```

One drawback of this approach is that usually the JDBC driver class name identifies a particular JDBC driver vendor, which can make the code that loads the driver specific to a particular vendor’s product and, therefore, non-portable.

In addition, an application needs to specify a JDBC URL when connecting to a database via the driver manager. For example,

```
Connection con = DriverManager.getConnection(
                  “jdbc:vendorX_subprotocol:machineY:portZ”);
```

The code above demonstrates that a JDBC URL may not only be specific to a particular vendor’s JDBC product, but also to a particular machine name and port number on that machine. This can make the application difficult to maintain as the computing environment changes. Using JNDI solves the problems outlined above by allowing an application to specify a logical name that JNDI associates with a particular data source. This makes deployment and maintenance of JDBC applications easier.

5.2 JDBC data sources

A JDBC data source object is a Java programming language object that implements the `javax.sql.DataSource` interface. A data source object is a factory for JDBC connections.

```
public interface DataSource {
```

```
            Connection getConnection() throws SQLException;
            Connection getConnection(String username,
                     String password) throws SQLException;

             ...
    }
```

Like other interfaces in the JDBC API such as `java.sql.Connection` and `java.sql.ResultSet`, implementations of `javax.sql.DataSource` will be provided by JDBC driver vendors as part of their JDBC 2.0 products. Appendix A, which is included primarily as an aid for JDBC driver implementors, contains a sample data source implementation

5.3 Data source properties

A class that implements the `DataSource` interface provides a set of properties that need to be given values so that a connection to a particular data source can be made. These properties, which follow the design pattern specified for JavaBeans^{TM} components, are usually set when a data source object is deployed. Examples of data source properties include the location of the database server, the name of the database, the network protocol to use to communicate with the server, and so on.

5.3.1 Standard properties

The JDBC 2.0 API specifies the following standard names for data source properties:

Table 1: Standard Data Source Properties

Property Name	Type	Description
databaseName	`String`	name of a particular database on a server
dataSourceName	`String`	a data source name; used to name an underlying `XADataSource`, or `ConnectionPoolDataSource` when pooling of connections is done
description	`String`	description of this data source
networkProtocol	`String`	network protocol used to communicate with the server
password	`String`	a database password
portNumber	`int`	port number where a server is listening for requests
roleName	`String`	the initial SQL rolename
serverName	`String`	database server name

Table 1: Standard Data Source Properties

Property Name	Type	Description
user	`String`	user’s account name

A data source is only required to support the *description* property. Support for the other properties listed above is not required since they may not be needed by all data sources. If a data source does need to use one of the properties listed above, however, it must use the standard property name. The standard properties are included in the JDBC specification to encourage uniformity among data sources from different vendors. For example, this standardization allows a utility to be written that lists available data sources and their descriptions along with other information if it is available.

A data source object that supports a property must supply getter and setter methods for it. For example, if `sds` denotes a data source object that supports the *serverName* property, the following methods must be provided. See the JavaBeans API specification for a full description of properties and their accessor methods.

```
sds.setServerName(“my_database_server”);
String prop = sds.getServerName();
```

Note that the getter and setter methods for a property are defined on the implementation class and not in the `DataSource` interface. This creates some separation between the management API for `DataSource` objects and the API used by applications. Applications shouldn't need to access/change properties, but management tools can get at them using introspection.

5.3.2 Vendor-specific properties

A data source may contain a property that is not included on the list of standard properties in Section 5.3.1. In this case a vendor-specific name for the property should be used.

5.4 Data source names

The JDBC 2.0 API specifies a simple policy for assigning JNDI names to data sources. All JDBC data sources should be registered in the *jdbc* naming subcontext of a JNDI namespace, or in one of its child subcontexts. The parent of the jdbc subcontext is the root naming context.

5.5 Examples

A data source object can be created, deployed, and managed within JNDI separately from the Java applications that use it. This section contains examples that illustrate the use of data source objects.

5.5.1 Creating a data source

The example below registers a data source object with a JNDI naming service.

```
SampleDataSource sds = new SampleDataSource();

sds.setServerName(“my_database_server”);
sds.setDatabaseName(“my_database”);

Context ctx = new InitialContext();
ctx.bind(“jdbc/EmployeeDB”, sds);
```

The first line of code in the example creates a data source object. The `SampleDataSource` class implements the `javax.sql.DataSource` interface. (The `SampleDataSource` class would be supplied by a JDBC driver vendor.) The next two lines initialize the data source’s properties. Then a Java object that references the initial JNDI naming context is created by calling the `InitialContext()` constructor, which is provided by JNDI. System properties (not shown) are used to tell JNDI the service provider to use.

The JNDI name space is hierarchical—similar to the directory structure of many file systems. The data source object is bound to a logical JNDI name by calling `Context.bind()`. In this case the JNDI name identifies a subcontext, “jdbc”, of the root naming context and a logical name, “EmployeeDB”, within the jdbc subcontext. This is all of the code required to deploy a data source object within JNDI.

Note that the example above is provided mainly for illustrative purposes. We expect that developers or system administrators will normally use a GUI tool to deploy a data source object.

5.5.2 Connecting to a data source

Once a data source has been registered with JNDI, it can then be used by a JDBC application, as is shown in the following example.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup(“jdbc/EmployeeDB”);
Connection con = ds.getConnection(“seth”, “teabag”);
```

The first line in the example creates a Java object that references the initial JNDI naming context. Next, the initial naming context is used to do a lookup operation using the logical name of the data source. The `Context.lookup()` method returns a reference to a Java `Object`, which is narrowed to a `javax.sql.DataSource` object. In the last line, the `DataSource.getConnection()` method is called to produce a database connection.

5.6 Logging and tracing

The `DataSource.setLogWriter()` method provides a way to register a character stream to which tracing and error logging information will be written by a JDBC implementation. This allows for `DataSource` specific tracing. If one wants all `DataSources` to use the same log stream, one must register the stream with each `DataSource` object individually. Log messages written to a `DataSource` specific log stream are not written to the log stream maintained by the `DriverManager`. When a `DataSource` object is created the log writer is initially null, in other words, logging is disabled.

5.7 Relationship to the driver manager

The `DataSource` facility provides an alternative to the JDBC `DriverManager`—essentially duplicating all of the driver manager's useful functionality. Although, both mechanisms may be used by the same application if desired, we encourage developers to regard the `DriverManager` as a legacy feature of the JDBC API. Applications should use the `DataSource` API whenever possible.

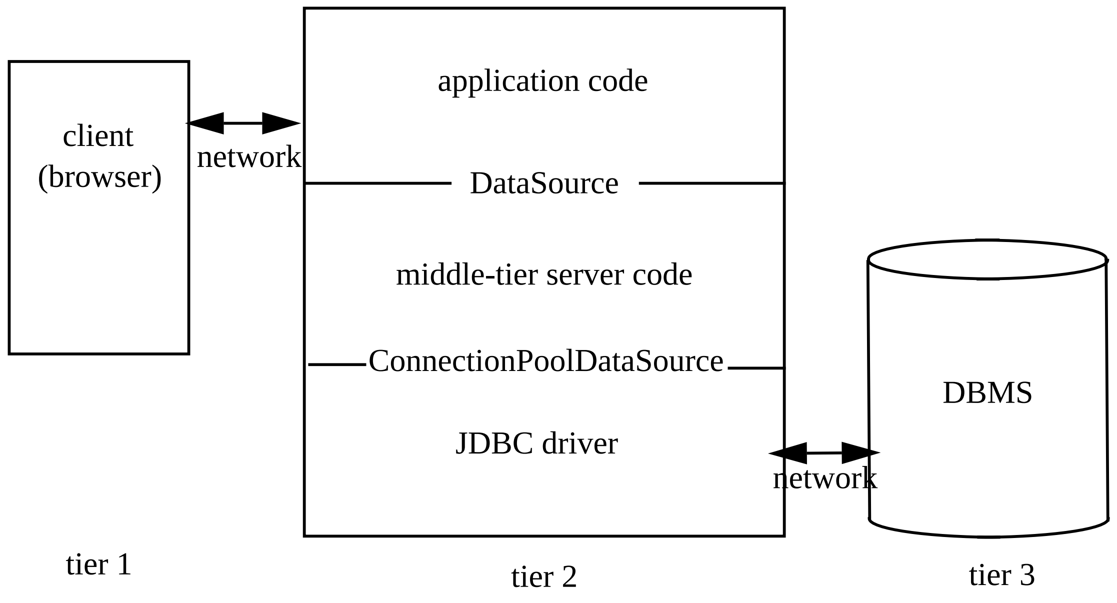
A JDBC implementation that is accessed via the `DataSource` API should not automatically register itself with the `DriverManager`. The `DriverManager`, `Driver`, and `DriverPropertyInfo` interfaces may be deprecated in the future.

6 Connection Pooling

A connection pool is a cache of database connections that is maintained in memory, so that the connections may be reused. Connection pooling is important for increasing performance, especially when the JDBC API is used in a middle-tier server environment, such as a Java-enabled web server.

6.1 Overview

The figure below gives an example of the basic components that might comprise a distributed, 3-tier application that uses the JDBC 2.0 API's connection pooling facility. The role-defining interface supported by each JDBC component is also shown.



In the figure, the client portion of the application is shown invoking some application code—a Java servlet, for example—that is executing in the middle-tier. The middle-tier application code uses the JDBC API to access a database. The application developer uses the `DataSource` interface in the standard way to obtain a `Connection` object. The `DataSource` implementation performs connection pooling and is implemented, in this case, by the middle-tier server vendor. The connection pooling implementation uses the facilities provided by the JDBC 2.0 driver vendor to implement the connection caching algorithm that it chooses. Lets take a closer look at the roles of each of the parties mentioned above: the JDBC vendor, the middle-tier server vendor, and the application developer.

In order to support connection pooling, a JDBC driver vendor must provide an implementation of the `javax.sql.ConnectionPoolDataSource` interface. This interface and the closely related `javax.sql.PooledConnection` interface (not shown above) provide methods—or "hooks"—that enable a third party such as a middle-tier server vendor to implement connection pooling as a layer on top of JDBC. The JDBC 2.0 API takes the approach of providing hooks to support connection pooling instead of mandating a particular connection pooling implementation because there are a large number of possible connection pooling algorithms that JDBC user's may want to use. Of course, JDBC driver vendors may also provide connection pooling implementations if they wish.

A party, such as a middle-tier server vendor, who wishes to implement connection pooling should provide a data source class—an implementation of the `javax.sql.DataSource` interface—that interacts with their particular connection pooling implementation. Instances of this data source class can be retrieved from JNDI by application code executing in the middle-tier and used to obtain a database connection in the usual manner.

Connection pooling doesn't impact application code. The application simply accesses a JDBC `DataSource` and uses it in the standard way. The `DataSource` implements connection pooling transparently to the application using the `PooledConnection` and `ConnectionPoolDataSource` facilities provided by a JDBC 2.0 driver. When an application is finished with a `Connection` object, it calls `Connection.close()`, as usual. This signals to the underlying connection pooling implementation that the physical database connection can be reused. The example below illustrates a typical sequence of JDBC API operations that an application developer would write in a middle-tier environment where connection pooling is being done. The example assumes that `ds` is of type `DataSource`.

```
// First, get a Connection. Connection pooling is done
// internally by the DataSource object.
Connection con = ds.getConnection("jdbc/webDatabase",
        "seth", "teacup");

// Do all the work as a single transaction (optional).
con.setAutoCommit(false);

// The actual work (queries and updates) would go here.
// Work is done using standard JDBC code as defined in the
// rest of the JDBC API.

// Commit the transaction.
con.commit();

// Close the connection. This returns the underlying physical
// database connection to the pool.
con.close();
```

This concludes the overview of the JDBC 2.0 API's connection pooling facility. We've seen that connection pooling actually has no impact on the code that application developers will write. The remainder of this chapter is intended mainly for JDBC driver implementors and those wishing to develop a connection pooling implementation. You may skip the rest of this chapter if you are interested only in developing applications using the JDBC 2.0 API.

6.2 Connection pool data sources

This section describes the `ConnectionPoolDataSource` and `PooledConnection` interfaces. Implementations of these interfaces are provided by JDBC driver vendors as part of their JDBC 2.0 products. The `ConnectionPoolDataSource` and `PooledConnection` interfaces are used by systems developers who are implementing a connection pooling module. These interfaces should not be used directly by developers writing JDBC applications. Appendix B outlines an example connection pooling implementation.

6.2.1 Basic description

A `ConnectionPoolDataSource` object is a factory for `PooledConnection` objects. The `ConnectionPoolDataSource.getPooledConnection()` method is called to create a `PooledConnection`. `ConnectionPoolDataSource` objects are similar in many ways to the `DataSource` objects discussed in Chapter 5. For example, a `ConnectionPoolDataSource` instance is normally registered in JNDI following the same naming conventions as those specified for `DataSource` objects. In addition, `ConnectionPoolDataSource` objects may generally support any of the properties that `DataSource` objects support.

```
public interface ConnectionPoolDataSource {

        PooledConnection getPooledConnection()
                throws SQLException;
        PooledConnection getPooledConnection(String user,
                String password) throws SQLException;

         ...

}
```

A `PooledConnection` object represents a physical connection to a data source. `PooledConnection` objects are cached by a connection pooling implementation, so that the physical connections they encapsulate can be reused.

```
public interface PooledConnection {

        Connection getConnection() throws SQLException;

        void close() throws SQLException;

        void addConnectionEventListener(
                ConnectionEventListener listener) ...;
        void removeConnectionEventListener(
                                ConnectionEventListener listener);
    }
```

A `PooledConnection` is a factory for the `Connection` objects that are actually used by an application. (Note the `PooledConnection.getConnection()` method above.) The reason for this will become clear in the next section. Although their APIs are identical, there are some interesting internal differences between the `Connection` objects returned by `PooledConnection.getConnection()` and traditional JDBC `Connection` objects, such as those returned by the JDBC `DriverManager`.

6.2.2 Fundamental concepts

When connection pooling is not being done, a `java.sql.Connection` object essentially encapsulates a physical database connection. Even though some JDBC drivers may contain logic to multiplex multiple JDBC `Connection` objects onto a single physical database connection, this logic is all internal to the driver, and code that uses JDBC can always assume that a `Connection` represents a physical database connection. Hence, we see that a `PooledConnection` is in many ways like a ‘traditional’ JDBC `Connection`, in that creating a `PooledConnection` object implies creating a physical database connection and calling `PooledConnection.close()` implies closing a physical database connection.

A `Connection` object produced by calling `PooledConnection.getConnection()` is somewhat different. Although a new `Connection` object instance is produced each time `PooledConnection.getConnection()` is called, this does not imply that a new physical database connection is created. Similarly, calling `Connection.close()` on a `Connection` object produced by calling `PooledConnection.getConnection()` never closes a physical database connection. A `Connection` object produced by calling `PooledConnection.getConnection()` is a temporary handle that an application can use to access an underlying physical database connection that is being pooled, and that is represented in the JDBC API by a `PooledConnection` object. A new `Connection` instance must be manufactured each time `PooledConnection.getConnection()` is called in order to mimic the standard behavior of the `DriverManager` and the `DataSource` interface, both of which always manufacture a new instance when their `getConnection()` methods are called.

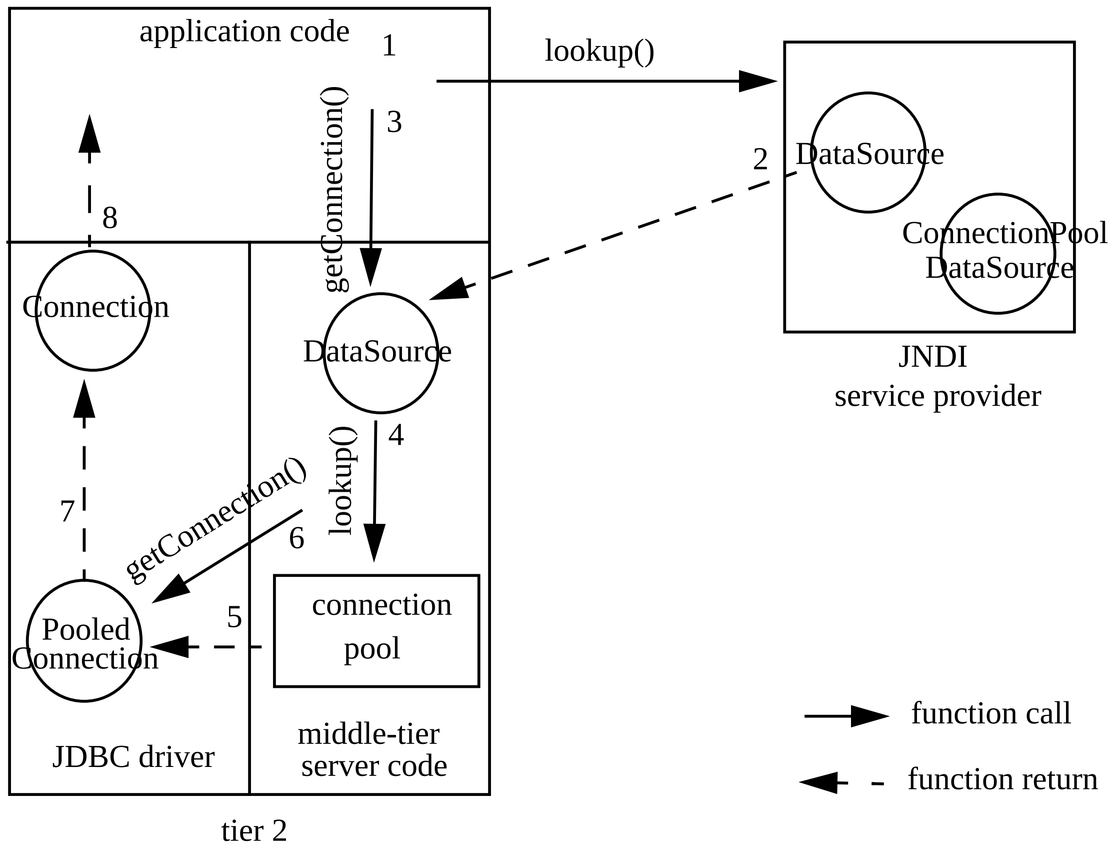
A JDBC application can be completely unaware of the relationships discussed in this section. The discussion above is of interest primarily to JDBC driver implementors, and persons implementing a connection pooling module, since they must understand the real relationship of `PooledConnection` objects to the `Connection` objects that they produce. Now, if you’re still a little confused—don’t worry! The next section should make things absolutely clear.

6.2.3 Retrieving a Connection

We proceed with a step-by-step illustration of the usage of the `ConnectionPoolDataSource` and `PooledConnection` interfaces in a middle-tier server environment. The figure below shows a typical sequence of steps that might be taken to satisfy a request for a database connection when connection pooling is being done in the middle-tier.

The steps are numbered 1-8 in the order they are performed. The middle-tier server is composed of three basic components that share the same address space: the application code, the middle-tier server's own code, and a JDBC driver. For simplicity, the client and database tiers of the application—tiers 1 and 3—are not shown.

1. First, the application code obtains the initial JNDI context and calls `Context.lookup()` to retrieve a `DataSource` object from a JNDI service provider.
2. The JNDI provider returns a `DataSource` object to the application. The `DataSource` object returned is implemented by the middle-tier server vendor in this case, and interacts with the server vendor's connection pooling runtime.
3. The application invokes `DataSource.getConnection()`. Note that the application expects a regular JDBC `Connection` object to be returned.



4. When `DataSource.getConnection()`, is called the middle-tier server performs a `lookup()` operation in the connection pool to see if there is a `PooledConnection` instance—a physical database connection—that can be

reused. Note that the JDBC API does not specify the interface between the `DataSource` object and the connection pooling module. The exact interaction of these components can be determined by the application server vendor.

5. If there is a hit in the connection cache, the connection pool simply updates its internal data structures and returns an existing `PooledConnection` object. Otherwise, a `ConnectionPoolDataSource` object is used to produce a new `PooledConnection` (not shown). In either case, the connection pooling module returns a `PooledConnection` object that is ready for use. The `PooledConnection` object is implemented by the JDBC driver.
6. The middle-tier server calls `PooledConnection.getConnection()` to obtain a `Connection` object for the application to use.
7. The JDBC driver creates a `Connection` object and returns it to the middle-tier server. Remember, this `Connection` object is really just a handle object that delegates most of its work to the underlying physical connection represented by the `PooledConnection` object that produced it.
8. The middle-tier server returns the `Connection` object to the application. The application uses the returned `Connection` object as though it were a normal JDBC `Connection`. The application code is completely unaware of the internal gymnastics that the middle-tier server has performed as a result of its `getConnection()` request.

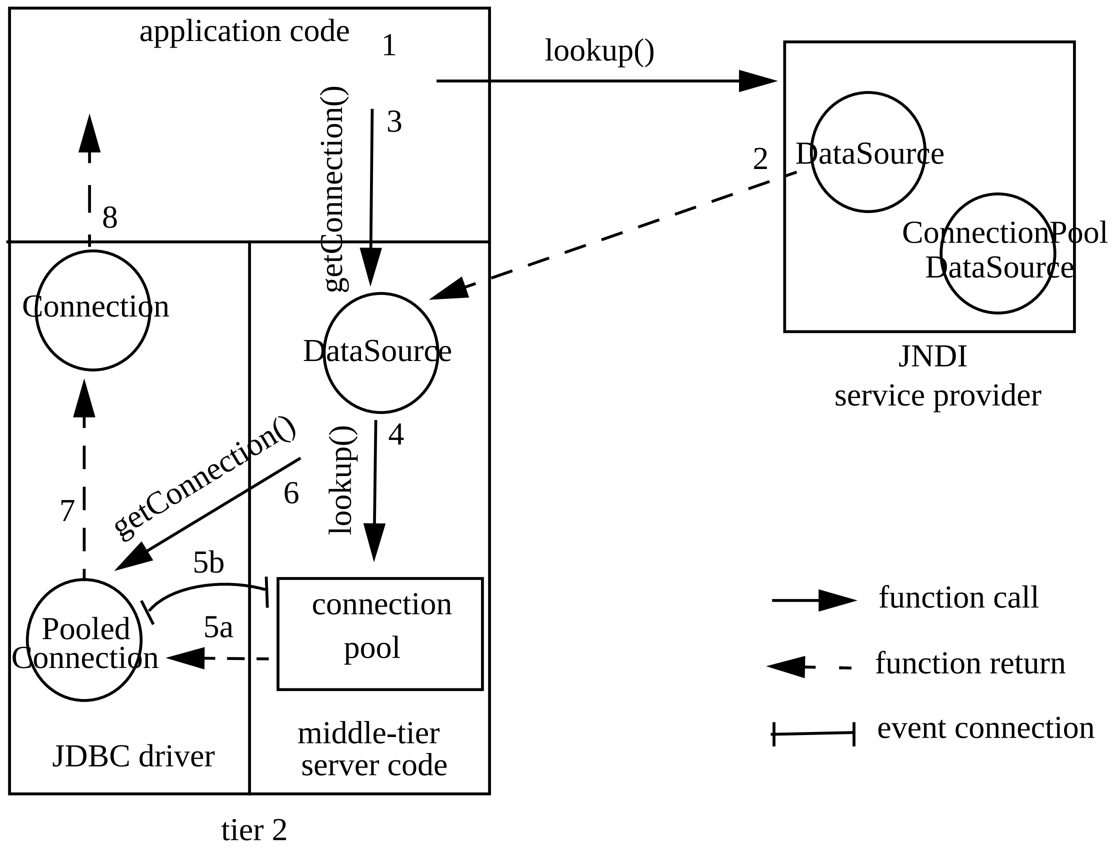
When the application is finished using the `Connection` object, it calls the `Connection.close()` method. As usual in JDBC, the application should not attempt to use its `Connection` after calling `Connection.close()`—else it will receive an `SQLException`. Note that the call to `Connection.close()` does not close the underlying physical connection represented by the associated `PooledConnection` object.

An individual `PooledConnection` object will usually be asked to produce a series of `Connection` objects during its lifetime. Only one of these `Connection` objects may be open at a particular point in time. The connection pooling implementation must call `PooledConnection.getConnection()` each time a new reference to a `PooledConnection` is handed out to the JDBC application layer. Each call to `PooledConnection.getConnection()` must return a newly constructed `Connection` object that exhibits the default `Connection` behavior. Only the most recent `Connection` object produced from a particular `PooledConnection` is open. An existing `Connection` object is automatically closed, if the `getConnection()` method of its associated `PooledConnection` is called again, before it has been explicitly closed by the application. This gives the application server a way to 'take away' a `Connection` from the application if it wishes, and give it out to someone else. This capability will not likely be used frequently in practice.

6.3 Handling events

Recall that in the previous section we said that the physical database connection wasn't actually closed when the application called `Connection.close()` on its `Connection` object. So, what actually did happen when `Connection.close()` was called? And, how was the physical database connection ever reused? The JDBC API uses JavaBeans style events to notify a connection pooling implementation when a `PooledConnection` can be recycled. When the application calls `Connection.close()`, an event is triggered that tells the connection pool it can recycle the physical database connection. In other words, the event signals the connection pool that the `PooledConnection` object which originally produced the `Connection` object generating the event can be put back in the connection pool.

`PooledConnection` objects provide a method—`addConnectionEventListener()` that allow a connection pooling module to register as a `ConnectionEventListener` and thereby receive *close* and *error* events relevant to a `PooledConnection`. Again, a `ConnectionEventListener` is notified by the JDBC driver when a JDBC application closes its `Connection`. The `PooledConnection` can then be put back into the cache of available `PooledConnection` objects and potentially reused. A `ConnectionEventListener` will also be notified when a fatal error occurs, so that it can make a note not to put a bad `PooledConnection` object back in the cache when the application finishes using it. When an error occurs, the `ConnectionEventListener` is notified by the JDBC driver, just before the driver throws an `SQLException` to the application to notify it of the same error. Note that automatic closing of a `Connection` object as discussed in the previous section does not generate a connection close event.



The connection pooling module typically registers itself as a `ConnectionEventListener` between Steps 5 and 6 of the figure in the previous section, before returning a `Connection` object to the application. In the figure above, step 5a corresponds to step 5 in the diagram in the previous section and step 5b has been added to illustrate the stage at which the connection pool module may register itself as a `ConnectionEventListener` with the `PooledConnection` object. Once the connection pool module registers itself as a listener it will be notified when the application closes the `Connection` object, or when an error on the pooled connection occurs.

6.3.1 Closing a `PooledConnection`

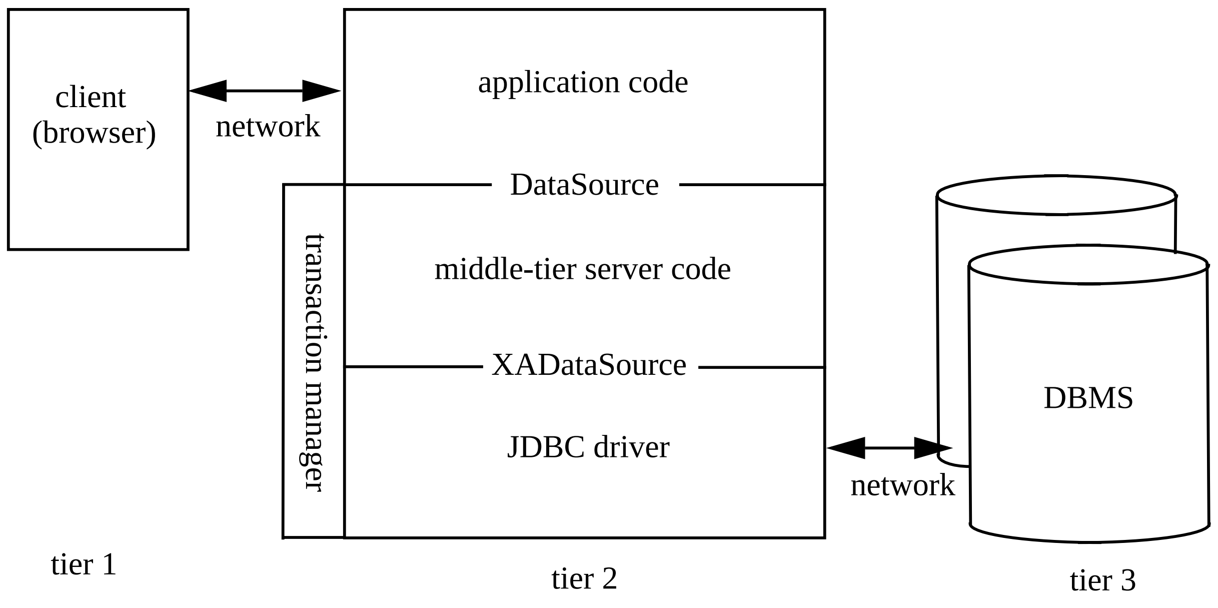
The `PooledConnection.close()` method is called by the connection pool module itself to tell the JDBC driver that it is finished using a physical database connection. This method will typically be called during an orderly shutdown of the middle-tier server or when the connection cache is re-initialized. Calling `PooledConnection.close()` closes a physical database connection.

7 Distributed Transactions

This chapter discusses the support that the JDBC 2.0 Standard Extension API provides for performing distributed database transactions. This chapter builds on the material presented in Chapter 6 concerning connection pooling. We begin with an overview and a discussion of those aspects of distributed transactions that are important for application developers. In this chapter the term *application developer* includes Enterprise JavaBeans providers as defined in the Enterprise JavaBeans specification, version 1.0. The remainder of the chapter then discusses topics of interest to JDBC vendors and application server vendors, including Enterprise JavaBeans server providers. The Java Transaction API (JTA), Java Transaction Service (JTS), and Enterprise JavaBeans specifications contain additional information related to the material in this chapter.

7.1 Overview

The figure below gives an example of the basic components that might comprise a distributed, 3-tier application that uses the JDBC 2.0 API's distributed transaction features. The role-defining interfaces supported by each JDBC component are also shown. Note that the figure below is very similar to the corresponding figure in Section 6.1.



While some details have been omitted for clarity, the figure basically shows a web client running in a browser that invokes some bit of application code executing in a middle-tier server environment. The middle-tier server could be a Java-enabled web server, or an Enterprise JavaBeans server, or some other type of application server that supports the Java programming language. The middle-tier application code uses the JDBC API to access a pair of databases in the context of a global (distributed) transaction. The fact that distributed transactions are used doesn't greatly affect the code that the application developer must write. The application code uses the `DataSource` interface in the standard way to obtain a `Connection` object. The `DataSource` implementation, which is provided by the middle-tier server vendor, interacts with the transaction manager to set up the transactional environment for the `Connection` returned to the application, and typically maintains a pool of database connections to achieve maximum performance. The `DataSource` implementation uses the facilities provided by the JDBC 2.0 driver vendor to implement connection caching and to enable distributed transactions. Lets take a closer look at the roles of each of the parties mentioned above: the JDBC vendor, the middle-tier server vendor, and the application developer.

In order to support distributed transactions, a JDBC driver vendor must provide implementations of the `javax.sql.XADataSource` and `javax.sql.XAConnection` interfaces. These interfaces are very similar to the `ConnectionPoolDataSource` and `PooledConnection` interfaces, respectively, that were introduced in Chapter 6. They provide the same hooks to enable connection pooling, and add the ability to do distributed transactions, as well. The details of how this is accomplished are discussed later in this chapter.

A middle-tier server vendor who wishes to support distributed transactions should provide a data source class—an implementation of the `javax.sql.DataSource` interface—that interacts with their middle-tier server's particular transaction infrastructure. The transaction infrastructure includes a transaction manager such as an implementation of the Java Transaction Service API, and a JDBC driver that supports the JDBC 2.0 API. Instances of this data source class can be retrieved from JNDI by application code executing in the middle-tier and used to obtain a database connection in the usual manner.

Distributed transactions don't impact application code significantly. However, there are a few coding differences that application developers should be aware of. The reason for the differences is that the JDBC API assumes that distributed transaction boundaries are controlled by either the middle-tier server, or another API such as the user transaction portion of the Java Transaction API. The JDBC API is intended to be used by component-based transactional applications that operate in a modern application server environment, such as an Enterprise JavaBeans server where declarative transactions are used.

Because transaction boundaries are controlled by the middle-tier server, the application code may not call the `Connection.commit()` or `Connection.rollback()` methods. These methods will throw an `SQLException` if they are called. In addition, the `Connection` object that is returned by a `DataSource` when distributed transactions are being done has autoCommit turned off by default. Attempting to enable autoCommit by calling `Connection.setAutoCommit()` with a value of `true` will throw an `SQLException`. The example below illustrates a typical sequence of JDBC API operations that an application developer would write in a middle-tier environment where distributed transactions are being used. The example assumes that `ds` is of type `DataSource`.

```
    // First, obtain a JDBC Connection. Distributed transactions
    // as well as connection pooling are handled internally by the
```

```
// DataSource object. All work done using the Connection will
// be part of the same global transaction.
Connection con = ds.getConnection("jdbc/webDatabase",
        "seth", "teacup");

// The actual work (queries and updates) would go here.
// Work is done using standard JDBC code as defined in the
// rest of the JDBC API.

// Close the connection. This returns the underlying physical
// database connection to the pool if connection pooling is done.
con.close();
```

The example shows that the application just has to create a `Connection`, do some work, and then close the `Connection`. The server takes care of committing the work done by the application, so the application just has to remember not to call commit itself.

This concludes the overview of the JDBC 2.0 API's distributed transaction features. We've seen that distributed transactions have very little impact on the code that application developers write. The remainder of this chapter is intended mainly for JDBC driver implementors and application server implementors. You may skip the rest of this chapter if you are interested only in developing applications using the JDBC 2.0 API.

7.2 New interfaces

Two new interfaces have been introduced in the JDBC 2.0 API to support distributed transactions. These are `javax.sql.XADataSource` and `javax.sql.XAConnection`. JDBC 2.0 drivers which support distributed transactions must implement the `javax.sql.XADataSource` and `javax.sql.XAConnection` interfaces.

7.2.1 The XADataSource interface

`XADataSource` objects are similar to `ConnectionPoolDataSource` and `DataSource` objects:

- `XADataSource` objects generally support the same properties as other data source objects.
- `XADataSource` objects are registered in JNDI using the same conventions as other data source objects.

An `XADataSource` object is a factory for `XAConnection` objects.

```
public interface XADataSource {
        XAConnection getXAConnection() throws SQLException;
        XAConnection getXAConnection(String user,
                    String password) throws SQLException;
        ...
}
```

An application server—like an EJB server, for example—uses an `XADataSource` to implement support for distributed transactions. The `XADataSource` interface is not visible to application code. The application server can assume that all of the `XAConnection` objects produced by a particular `XADataSource` object refer to the same underlying resource manager instance. However, unless the application server has some special knowledge, it may not generally assume that different `XADataSource` objects refer to different resource managers. In other words, it is possible that different `XADataSource` objects will refer to the same resource manager. In the case of the JDBC API, the term *resource manager* (RM) refers to a particular DBMS server that participates in a distributed transaction.

The mapping of `XADataSource` objects to resource managers is important to keep in mind when implementing a distributed transaction processing (DTP) system that is based on the X/Open XA architecture. XA requires that an individual resource manager receive only a single prepare/commit flow for a particular transaction branch of a global transaction. Thus, different `XADataSource` objects that are part of the same global transaction will need to be assigned to different transaction branches when the application server is uncertain about whether or not they refer to different underlying resource managers.

7.2.2 The XAConnection interface

Like `PooledConnection`, an `XAConnection` object represents a physical database connection and is itself a factory for `Connection` objects. The `XAConnection` object is used by the application server to implement distributed transactions and is not visible to application code. The `XAConnection` interface extends `PooledConnection` with a single method, `getXAResource()`, that is used to obtain an `XAResource` object for the `XAConnection`.

```
public interface XAConnection extends PooledConnection {
        javax.jta.xa.XAResource getXAResource()
                  throws SQLException;

}
```

A `javax.transaction.xa.XAResource` object is used by a transaction manager to associate and disassociate a global transaction with an `XAConnection` (See `XAResource.start()` and `XAResource.end()`) and to perform the 2-phase commit transaction protocol (See `XAResource.commit()`, `XAResource.rollback()`, and so on.). A transaction manager never deals directly with the `XAConnection` interface.

The `XAResource` interface is a Java mapping of the industry standard X/Open XA interface. A JDBC driver must maintain a 1-to-1 relationship between `XAConnection` and `XAResource` objects. In other words, each time that `XAConnection.getXAResource()` is called for a particular `XAConnection`, the same `XAResource` object instance is returned. The `XAResource` object is implicitly opened by the JDBC driver before it is returned to the application server, and is closed when its corresponding `XAConnection` object is closed. The methods on the `XAResource` interface may be called from any thread in the application server process.

```
public interface XAResource {
        void commit(Xid xid, boolean onePhase)
                throws XAException;
        void end(Xid xid, int flags) throws XAException;
        void forget(Xid xid) throws XAException;
        int prepare(Xid xid) throws XAException;
        Xid[] recover(int flag) throws XAException;
        void rollback(Xid xid) throws XAException;
        void start(Xid xid, int flags) throws XAException;
}
```

The `XAResource` object can be used to associate an `XAConnection` with a single global transaction at a time. Also note that any `XAResource` object that refers to the proper underlying resource manager can be used during 2-phase commit to commit the global transaction. The `XAResource` object used during commit processing need not have been enlisted to do work for the transaction. A consequence of this is that a TM may simply want to remember the `XADataSource` objects that were involved in a global transaction and use any `XAResource` object associated with a particular `XADataSource` object to commit the transaction branch associated with the `XADataSource` object.

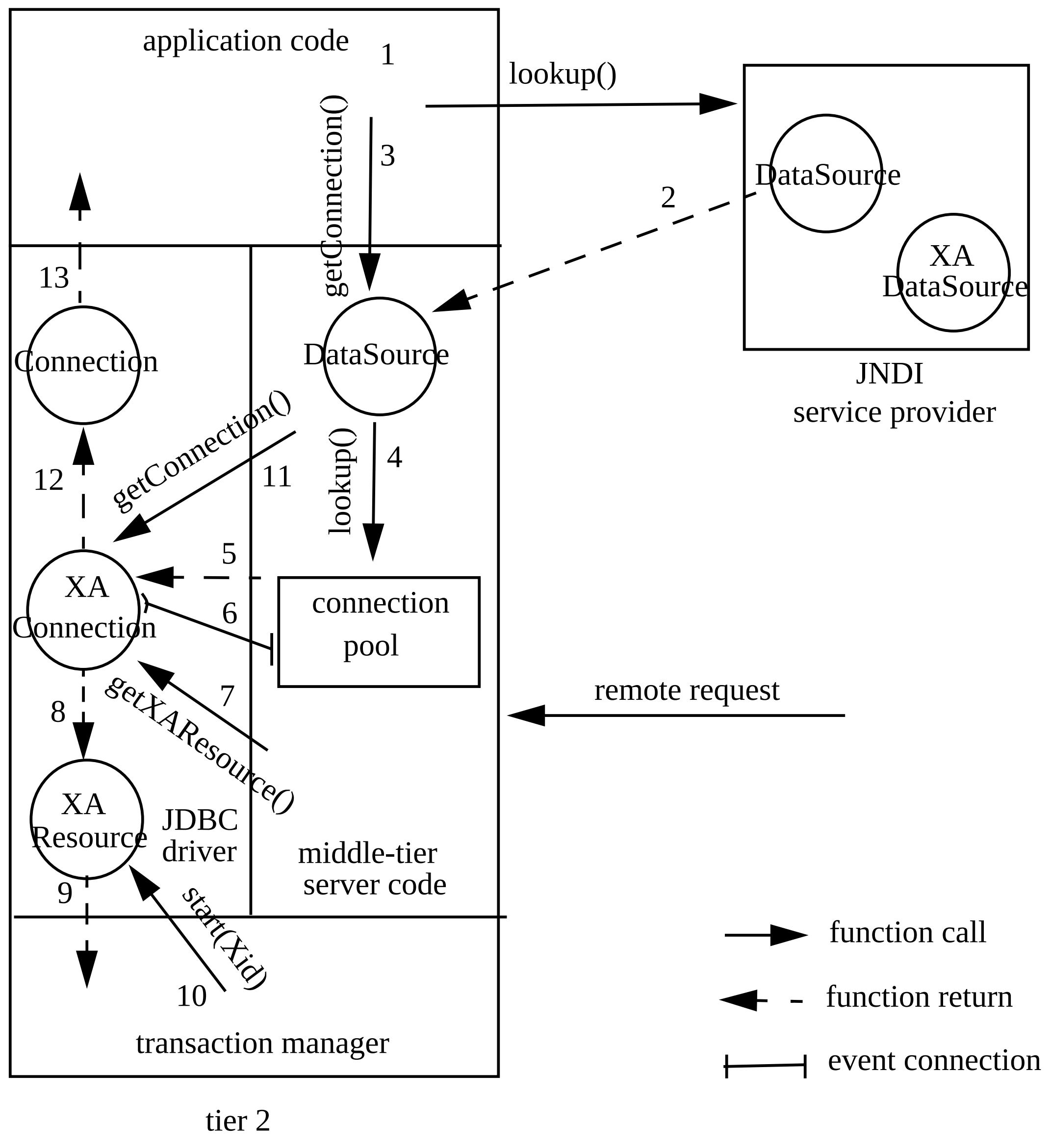
We have briefly covered the important aspects of the `XAResource` interface in this section. The `XAResource` interface is formally defined in the Java Transaction API (JTA) specification which is a separate document. Finally we would like to reemphasize that, as with the `ConnectionPoolDataSource` interface, JDBC applications never deal directly with an `XADataSource`, `XAConnection`, or `XAResource` object. Connection pooling for `XAConnection` objects can be done using the same hooks as were provided for `PooledConnection` objects (see Chapter 6).

7.3 Examples

The examples in this section illustrate usage of the new interfaces in a middle-tier server environment where both distributed transactions and connection pooling are supported.

7.3.1 Retrieving a `Connection`

The figure below illustrates the process by which application code obtains a database connection when pooling of XA connections is being done in a middle-tier server. We begin with the arrival of a remote client request. The request may bring with it a global transaction id, `Xid`, if work is already being done in the scope of a distributed transaction. Alternatively, the middle-tier server may begin a global transaction before passing the remote request on to the application code. Once control is passed to the application code, the application eventually requests a JDBC `Connection` with which to access an underlying DBMS. At this point, the sequence of steps that are illustrated in the figure commences:

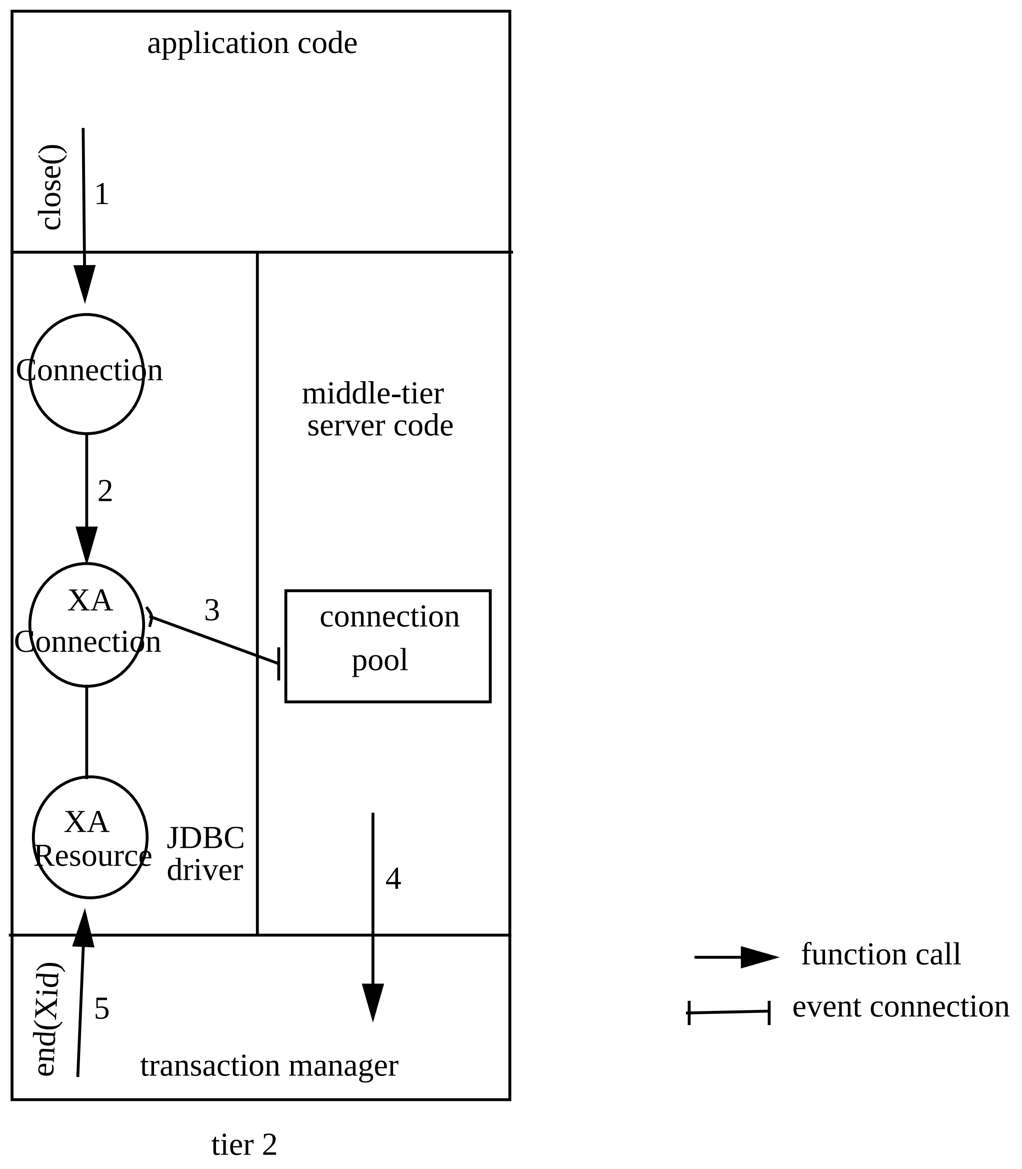


1. First, the application obtains the initial JNDI context and calls `Context.lookup()` to retrieve a `DataSource` object from a JNDI service provider.
2. The JNDI provider returns a `DataSource` object to the application. The `DataSource` object returned is implemented by the middle-tier server vendor and it interacts with the server vendor's connection pooling runtime.
3. The application invokes `DataSource.getConnection()`.

4. As a result of calling `DataSource.getConnection()`, a `lookup()` is done in the connection pool to see if there is an `XAConnection` instance that can be reused. Note that the JDBC API does not specify the interface between the `DataSource` object and the connection pool. The exact interaction of these components is determined by the middle-tier server vendor, in this case.
5. If there is a hit in the connection cache, the connection pool simply updates its internal data structures and returns an existing `XAConnection` object that is ready for use. Otherwise, an `XADataSource` object would be used to produce a new `XAConnection` object and create a new physical database connection (not shown). In either case, an `XAConnection` object that is ready for use is returned.
6. The connection pool module registers itself as a `ConnectionEventListener` with the `XAConnection` object. Once the connection pool module registers itself as a listener it will be notified when the application closes its `Connection` object, or when an error on the `XAConnection` occurs.
7. The middle-tier server calls `XAConnection.getXAResource()` to obtain an XA resource object that can be used to control distributed transactions on the connection.
8. The JDBC driver returns an `XAResource` object to the middle-tier server.
9. The middle-tier server passes the `XAResource` object to the transaction manager.
10. The transaction manager calls `XAResource.start(Xid)` to associate actions performed on the `XAConnection` with the global transaction denoted by `Xid`.
11. The middle-tier server calls `XAConnection.getConnection()` to obtain a `Connection` object for the application to use.
12. The JDBC driver creates a `Connection` object and returns it to the middle-tier server. Conceptually, the `Connection` object returned is just a handle that delegates most of its work to the underlying physical database connection represented by the `XAConnection` object.
13. The connection pool module returns the `Connection` object to the application. The application uses the `Connection` to do some work, but doesn't call commit or rollback since these operations are controlled by the middle-tier server.

7.3.2 Closing a `Connection`

The figure below illustrates a typical sequence of steps that occur when the application code finishes using its JDBC `Connection`.



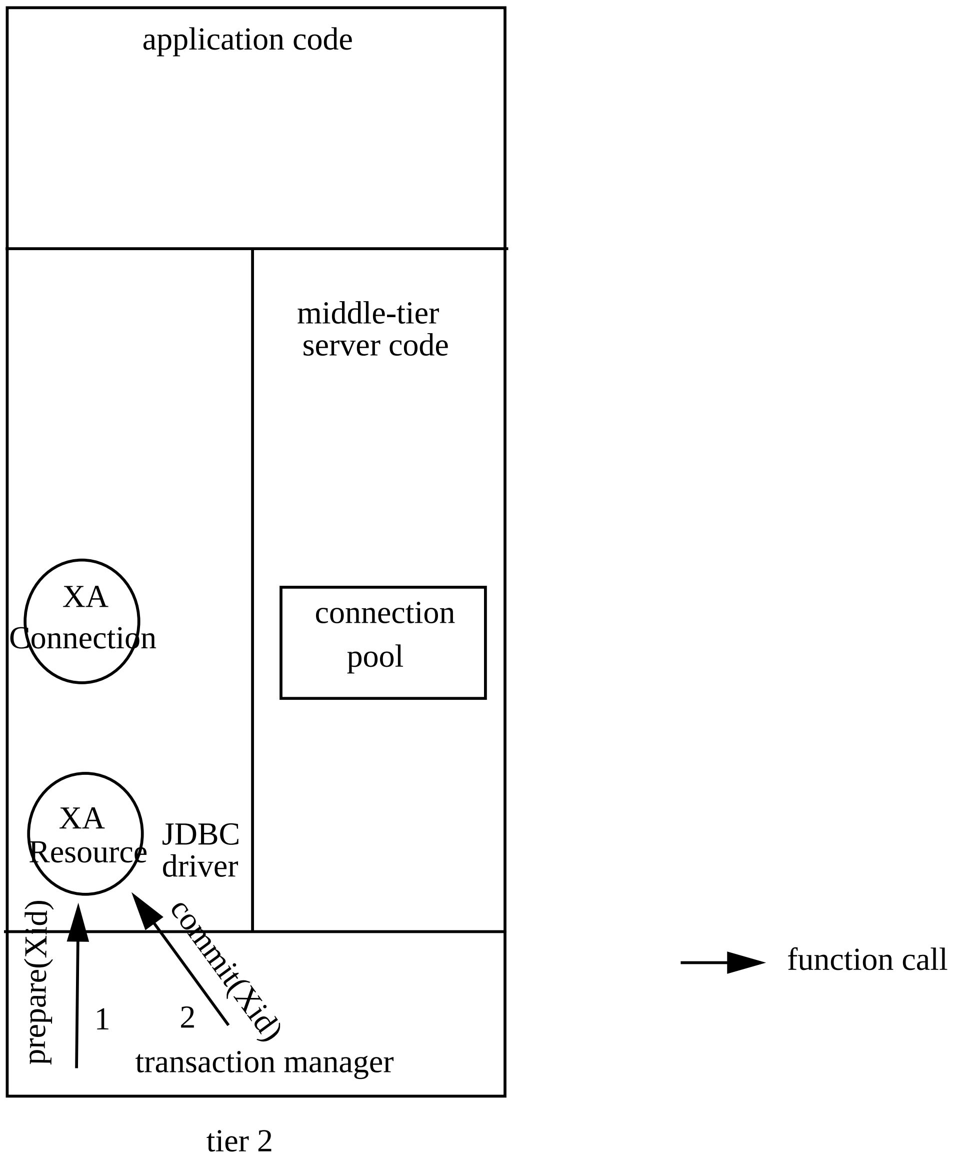
1. The application calls `Connection.close()` to signal that it is done using the `Connection` object.
2. The `Connection` object delegates the close request to the underlying `XAConnection` object.
3. The `XAConnection` object notifies all registered listeners that the application has called `close()` and that the connection is available.
4. The middle-tier server receives the event notification and notifies the transaction manager that the application is done using the `XAConnection`.

5. The transaction manager calls `XAResource.end(Xid)` to tell the resource manager that further operations on the `XAConnection` are no longer associated with the global transaction denoted by `Xid`.

Once Step 5 completes, the connection pooling module can update its internal data structures so that the `XAConnection` can be reused by the next request.

7.3.3 Committing a transaction

This section illustrates the steps taken to commit a distributed transaction. These steps are executed at some point in time after `XAResource.end(Xid)` is called. They may be initiated by the middle-tier server itself if the server initiated the distributed transaction, or a remote process if the transaction is being controlled by a remote entity.



1. First, the transaction manager calls `XAResource.prepare(Xid)` to begin the first phase of the commit process. The transaction manager can use any `XAResource` object that is associated with the proper underlying resource manager, and not necessarily one that was directly involved with the global transaction previously. A middle-tier server may assume that all of the `XAResource` objects produced by a particular `XADataSource` refer to the same underlying resource manager.
2. Assuming that all resource managers involved in the distributed transaction agree to commit, the transaction manager calls `XAResource.commit(Xid)` to commit the transaction. Otherwise, `XAResource.rollback(Xid)` is called. The transaction manager may use any `XAResource` object that is associated with the proper underlying resource manager for this step.

7.4 More on `Connection` objects

Up to this point, our description of the behavior of a `Connection` object produced by an `XAConnection` has been somewhat simplified, for ease of explanation. The purpose of this section is to give a complete description of this behavior. Previously, it has always been assumed that a `Connection` object produced by an `XAConnection` object was used in the scope of a global transaction. When this is the case, a JDBC application may not call the operations `Connection.commit()` or `Connection.rollback()` on the `Connection` object as these operations will throw an `SQLException.` In addition, we have said that the `Connection` returned by `XAConnection.getConnection()` is not in autocommit mode, which is the normal default and that attempting to set autocommit to true will throw an `SQLException`.

The above statements are all true, however, for completeness we note that it is also possible for an application to use a `Connection` object produced by calling `XAConnection.getConnection()` outside of the scope of a global transaction—in a local transaction mode. In this case, the `Connection` behaves like a ‘normal’ JDBC connection, and all of its methods have their usual behavior.

7.5 Enterprise JavaBeans

The support provided by the JDBC 2.0 API for distributed transactions is an integral part of the distributed transaction architecture of the Enterprise JavaBeans (EJB) API. The EJB specification defines several different transaction attributes that an EJB provider may use to declaratively specify the transactional behavior of an Enterprise Bean. They are:

- TX_NOT_SUPPORTED
- TX_BEAN_MANAGED
- TX_REQUIRED
- TX_SUPPORTS
- TX_REQUIRES_NEW

- TX_MANDATORY

See the Enterprise JavaBeans Specification, version 1.0 for a detailed description of the transaction attributes listed above. Note that the transactional behavior of an EJB is always specified at development time, and may not be changed later on in the EJB workflow.

Any EJB implementation can use the JDBC API to do its work, no matter what value its transaction attribute has been set to. An EJB implementation is expected to know whether or not it will execute in the context of a global transaction and use the JDBC API accordingly. For example, an EJB that runs in the context of a global transaction should not call `Connection.commit()`, or attempt to enable autoCommit, while an EJB that runs outside of a global transaction is allowed to make full use of all of the methods on the `Connection` interface. A TX_BEAN_MANAGED bean may even use the same JDBC `Connection` both inside and outside of a global transaction. In this case, the behavior of the `Connection.commit()`, `Connection.setAutoCommit()`, etc. methods will differ depending on whether the `Connection` is in the scope of a global transaction or not.

In order for an EJB server implementation to support EJBs that use the JDBC API, it should provide an implementation of the `javax.sql.DataSource` interface that interacts with its own distributed transaction infrastructure. This makes it possible to deploy an EJB that uses the JDBC API on that particular EJB server. In order for Enterprise Beans that use the JDBC API to be portable across different EJB server products, all EJB server implementations will need to provide such a `DataSource` implementation.

8 Rowsets

This chapter describes the rowset facility provided by the JDBC Standard Extension API.

8.1 What is a rowset?

As its name implies, a rowset is an object that encapsulates a set of rows that have been retrieved from some tabular data source. A rowset object must implement the `javax.sql.RowSet` interface which adds support to the JDBC API for the JavaBeans^{TM} component model. For example, a rowset can be used as a Bean in a visual Bean development environment—a `RowSet` instance can be created and configured at design time and its methods executed at runtime.

8.2 Rowsets at design time

8.2.1 Properties

The `RowSet` interface provides a set of JavaBeans properties that allow a `RowSet` instance to be configured to connect to a data source and retrieve a set of rows. For example,

```
rset.setDataSourceName("jdbc/SomeDataSourceName");
rset.setTransactionIsolation(
        Connection.TRANSACTION_READ_COMMITTED);
rset.setCommand("SELECT NAME, BREED, AGE FROM CANINE");
```

The example above uses a variable `rset` of type `RowSet`. The example sets three `RowSet` properties: data source name, transaction isolation, and command. The data source name property is used by a `RowSet` to lookup a JDBC `DataSource` object in a JNDI naming service. (JDBC data sources are the most common type of data source used by rowsets.) The `DataSource` object is used internally to create a connection to the underlying data source. The transaction isolation property specifies that only data that was produced by committed transactions may be read by the rowset. Lastly, the command property specifies a command that will be executed to retrieve a set of rows. In this case the NAME, BREED, and AGE columns for all rows in the CANINE table are retrieved.

8.2.2 Events

`RowSet` components support JavaBeans events which allows other JavaBeans components in an application to be notified when an important event on a rowset occurs. A component that wishes to register for `RowSet` events must implement the `RowSetListener` interface. Event listeners are registered with a rowset by calling the `addRowSetListener` method as shown below. Any number of listeners may be registered with an individual `RowSet` object.

```
RowSetListener listener ...;

rset.addRowSetListener(listener);
```

Rowsets can generate three different types of events. Cursor movement events indicate that the rowset's cursor has moved. Row change events indicate that a particular row has been inserted, updated or deleted. Finally, rowset change events indicate that the entire contents of a rowset changed which may happen when `RowSet.execute()` is called, for example. When an event occurs, an appropriate listener method is called to notify a listener. If a listener is not interested in a particular event type, it may implement an event handling method with an empty method body. An event listener is passed a `RowSetEvent` object which identifies the source of the event when an event occurs.

8.3 Rowsets at run-time

8.3.1 Parameters

In Section 8.2.1, a simple SQL command that takes no input parameters was used, but a rowset may also use a command that accepts input parameters. A group of `setXXX()` methods provide a way to pass input parameters to a rowset. The example below shows a command that takes a `String` input parameter. The `RowSet.setString()` method is used to pass the input parameter value to the `RowSet`. Typically, the command property is specified at design time, while parameters aren't set until run-time when their values are known.

```
rset.setCommand(
        "SELECT NAME, BREED, AGE FROM CANINE WHERE NAME = ?");
rset.setString(1, "spot");
```

8.3.2 Traversing a rowset

The `javax.sql.RowSet` interface extends the `java.sql.ResultSet` interface, so in many ways a rowset behaves just like a result set. In fact, most components that make use of a `RowSet` component will likely treat it as a `ResultSet` object. A `RowSet` is simply a `ResultSet` that can function as a JavaBeans component. The code below shows how to iterate forward through a rowset. Notice that since a rowset is a result set, this code is identical to the code that would be used to iterate forward through a result set.

```
// iterate forward through the rowset
rset.beforeFirst();
while (rset.next()) {
        System.out.println(rset.getString(1) + " " +
                     rset.getFloat(2));
}
```

Iterating backward through the rowset, positioning the cursor to a specific row, and so on, are also done identically as for result sets.

8.3.3 Command execution

A rowset may be filled with data by calling the `RowSet.execute()` method. The `execute()` method uses the appropriate property values that have been set to connect to a data source, and retrieve some data. The exact properties that must be set, may vary between `RowSet` implementations. Developers will need to check the documentation for the particular rowset they're using to find out what properties are required. The `RowSet` interface contains the properties that are needed to connect to a JDBC data source. The current contents of the rowset, if any, are lost when `execute()` is called.

9 Rowset implementations

The `RowSet` interface can be implemented using the rest of the JDBC API. In other words, a `RowSet` is a layer of software that can function on top of a JDBC driver. Implementations of the `RowSet` interface can be provided by anyone, including JDBC driver vendors who want to provide a `RowSet` implementation as part of their JDBC products.

This chapter contains concrete descriptions of three possible `RowSet` implementations: the `CachedRowSet`, `JDBCRowSet`, and `WebRowSet` classes. These classes are not part of the JDBC Standard Extension release that is described in this document, but will likely be included in a future release. They are discussed here to illustrate the power of rowsets, and to give a preview of the types of rowset implementations that will likely be available in the future. This chapter also describes the use of the `RowSetReader`, `RowSetWriter`, and `RowSetInternal` interfaces which are used by the rowset implementation classes. The `RowSetReader`, `RowSetWriter`, and `RowSetInternal` interfaces are included in this release of the JDBC 2.0 Standard Extension API.

9.1 The CachedRowSet class

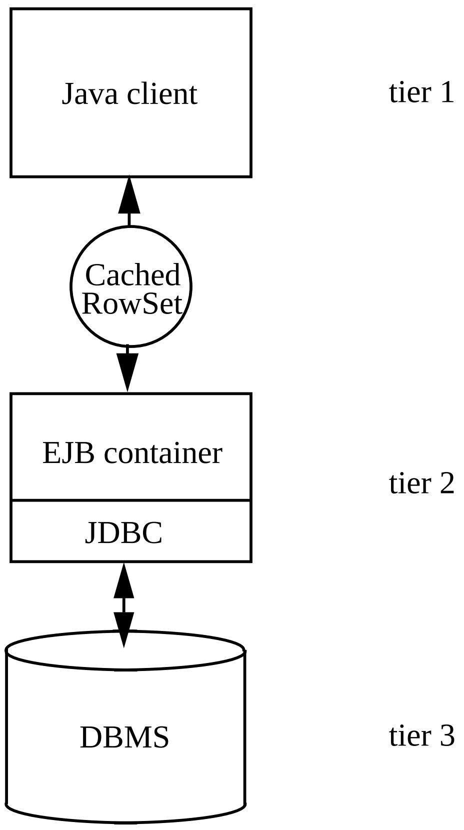
9.1.1 Overview

The `CachedRowSet` class is representative of rowset implementations which, in a nutshell, provide a disconnected, serializable, scrollable container for tabular data. A `CachedRowSet` object can simply be thought of as a disconnected set of rows that are being cached outside of a data source. Since all data is cached in memory, `CachedRowSets` are not appropriate for extremely large data sets.

An important intended use of the `CachedRowSet` style of rowset implementation is as a container for tabular data that can be passed between different components of a distributed application. For example, an Enterprise JavaBeans component running in an application server may use JDBC to retrieve a set of rows from a database. Then a `CachedRowSet` object (since it is serializable) can be used to send the data over the network from the EJB server to a client running in a web browser. This scenario is illustrated in the figure below.

The figure below shows that a `CachedRowSet` can be used to provide Java clients with access to tabular data in situations where it would be inappropriate to use a JDBC driver directly in the client due to resource limitations or security considerations. This includes PersonalJava clients such as a Personal Digital Assistant (PDA), as well as thin clients like a Network Computer (NC). Put another way, a `CachedRowSet` object provides a means to "get rows in" and—since it is updatable—"get changed rows out" without the need for a thin client to host a full implementation of the JDBC API which requires database connectivity, database metadata access, and command execution, among other things.

A second use for `CachedRowSets` is to provide scrolling and updating for `ResultSets` that don't provide these capabilities themselves. A `CachedRowSet` can be used to augment the capabilities of a JDBC driver that doesn't have full support for scrolling and updating.



9.1.2 Creating a CachedRowSet

A `CachedRowSet` is a JavaBean, so `CachedRowSet` instances will often be created by a visual Beans development tool when an application is being assembled. Applications may also create instances at run-time using any public constructor that the `CachedRowSet` class provides.

9.1.3 Retrieving data

A `CachedRowSet` object can contain data retrieved via a JDBC driver or data from some other source, such as a file. We expect that `CachedRowSet` objects will contain data that was fetched from an SQL database using JDBC most of the time. One way to get data into a `CachedRowSet` is to call the `CachedRowSet.populate()` method (defined on the `CachedRowSet` class), as shown below.

```
ResultSet rs = stmt.executeQuery(
        “SELECT NAME, SALARY FROM EMPLOYEE”);
CachedRowSet crset = new CachedRowSet();
crset.populate(rs);
```

The example above uses a variable `stmt` of type `Statement`. The `populate()` method takes a parameter that is a `ResultSet` containing some data. The `populate()` method reads the contents of the `ResultSet`, and caches the contents in the `CachedRowSet` object. Recall that we use the term *disconnected* to describe a `CachedRowSet`. Disconnected means that a `CachedRowSet` object caches its data outside of a data source. Once the `populate()` method returns, the `CachedRowSet` object won't maintain a connection to the underlying data source that produced the initial `ResultSet`. Note that the `populate()` method doesn't require that any properties be set ahead of time to enable data source connectivity.

Another way to get data into a `CachedRowSet` object will be to call `execute()`. There are two forms of `execute()`: one which takes a `Connection` object as a parameter and one that does not. If a `Connection` is passed to `execute()` then there is no need to set properties ahead of time to enable data source connectivity. However, `execute()` does always require that a command be specified via the command property. The `CachedRowSet` only makes use of the JDBC connection briefly while data is being read from the database and used to populate it with rows. To use `execute()` one simply calls it once the appropriate properties have been set.

```
    crset.execute();
```

Here, since a `Connection` is not passed to execute(), the `CachedRowSet` must create a `Connection` internally and use it to retrieve a set of rows, assuming that a JDBC data source is being accessed.

9.1.4 Accessing data

The contents of a `CachedRowSet` are accessed using methods inherited from the `ResultSet` interface. A `CachedRowSet` is always scrollable, and has type `ResultSet.TYPE_SCROLL_INSENSITIVE` since the rows it contains are cached outside of a data source.

9.1.5 Modifying data

A `CachedRowSet` object keeps track of both its *original value* and its *current value.* The original value is set by methods `CachedRowSet.execute()` and `populate()`. For example, when `execute()` is called, the original value is typically assigned the set of rows returned by executing the rowset's command. After calling `execute()` (or `populate()`) the original value and the current value are the same.

The `CachedRowSet.updateXXX()` methods (which are inherited from `ResultSet`) can be called to update the current value of a `CachedRowSet`. Calling an `updateXXX()` method does not affect the original value, or the values stored in the underlying data source. Only the current value of the `CachedRowSet`, which is cached in memory, is changed.

The example below illustrates the concepts mentioned above. When `CachedRowSet.execute()` is called in the first line of the example, the original and current value of the `CachedRowSet`, which are maintained in main memory, are initialized. The cursor is then positioned to the first and second rows, in turn, and `updateString()` and `updateFloat()` are called to modify the current value of the first two rows in the rowset, which contain an employee name and salary. The `updateRow()` method is called to signal completion of each row update. Once `updateRow()` is called, `cancelRowUpdates()` can no longer be called to undo updates to the current value of the row. `UpdateRow()` does not affect the original value of the row or the underlying values in the data source. Deleting and inserting of rows is done exactly as for `ResultSets`, so this is not shown in the example.

```
// initialize the original and current values
crset.execute();

// update the current value of the first row
crset.first();
crset.updateString(1, "John Jacobs");
crset.updateFloat(2, 50000f);
crset.updateRow();

// update the current value of the second row
crset.relative(1);
crset.updateString(1, "Harold Hill");
crset.updateFloat(2, 38000f);
crset.updateRow();

// update the original value and the database
crset.acceptChanges();
```

After making some changes, `acceptChanges()` can be called to propagate any updates, inserts and deletes back to the underlying data source. The `acceptChanges()` method invokes a *writer* component internally to actually update the data source. Writers are discussed in more detail in Section 9.1.6, but typically a writer compares the original value of each updated row with the values stored in the database to be sure that the underlying data hasn't been changed by someone else. If nothing has been changed, the rowset's updates are written to the database. The original value is set equal to the current value before `acceptChanges()` returns.

Alternatively, an application may call the method `restoreOriginal()` if it decides to discard the updates that it has made on a `CachedRowSet`. Calling `restoreOriginal()` simply replaces the current value of the rowset with the original value—there is no interaction with the underlying data source.

9.1.6 Reading and writing data

The rowset framework provides an extensible reader/writer facility that allows data retrieval and updating to be customized. This facility is used by the `CachedRowSet` class.

Custom readers and writers are useful because they can enable a `RowSet` to read its data from a regular file data source or from some other non-SQL data source like a spreadsheet. A `CachedRowSet` style implementation that has been configured with a custom reader/writer can be made available as a normal JavaBeans component. Thus, developers writing applications can generally ignore the details of the customization process and focus on more important aspects of using `RowSets`.

When the `CachedRowSet.execute()` method is called by an application, the `CachedRowSet` actually invokes an object that implements the `RowSetReader` interface internally to handle the task of reading data from a data source. The `RowSetReader` interface provides a single method, `readData()`, which is called to read the new contents of the rowset. A `CachedRowSet` object exposes itself as an object of type `RowSetInternal` to the `RowSetReader`. The `RowSetInternal` interface contains some additional methods that are needed by the `RowSetReader` to do its work. Appendix C contains an example implementation of the `RowSetReader` interface.

Similarly, when the `CachedRowSet.acceptChanges()` method is called, the `CachedRowSet` invokes an object that implements the `RowSetWriter` interface internally to handle the task of writing changed rows back to the data source. The `RowSetWriter.writeData()` method is invoked by the rowset to write any data that has been changed. This method may detect a conflict with the underlying data and signal this by returning a value of `true` to the caller.

Readers and writers are registered with a `CachedRowSet` object by calling the methods `CachedRowSet.setReader()` and `CachedRowSet.setWriter()`. A `CachedRowSet` style implementation should also provide corresponding getter methods.

9.1.7 Miscellaneous methods

The `CachedRowSet.clone()` and `CachedRowSet.createCopy()` methods both create an exact copy of a rowset that is independent from the original. By contrast the `CachedRowSet.createShared()` method creates a rowset that shares its state with the original rowset, i.e. both the new and the original rowset share the same physical, in-memory copy of their original and current values. Updates made by calling an `updateXXX()` method on one shared rowset are reflected in the other. The `createShared()` method, in effect, allows an application to create multiple cursors over a single set of rows.

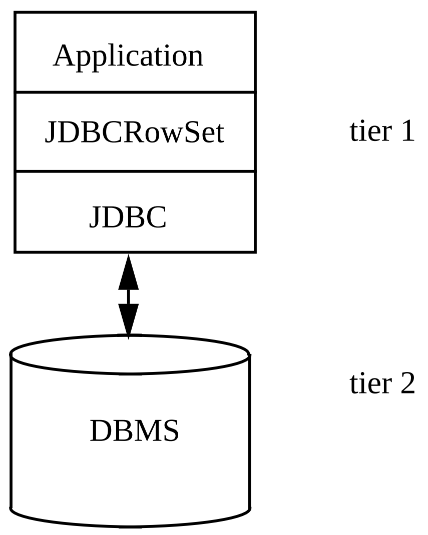
`CachedRowSets` also provide methods for easy conversion to a Java collection, for convenience.

9.2 The JDBCRowSet class

9.2.1 Overview

The `JDBCRowSet` class represents a style of rowset that we term *connected* since it always maintains an open database connection when it is in use. The purpose of the `JDBCRowSet` class is to provide a thin layer that wraps around a JDBC `ResultSet` at runtime, in effect, making a JDBC driver look like a JavaBeans component. The figure below shows a 2-tier application architecture that uses the `JDBCRowSet` class. Note that the `JDBCRowSet` requires the presence of a JDBC driver. This is different from a

`CachedRowSet` style implementation (see Section 9.1) which may be used in environments where a JDBC driver is not present. A `JDBCRowSet` style implementation may provide additional functionality, such as scrolling, that isn’t provided by the underlying JDBC driver.



9.2.2 Creating a JDBCRowSet

The `JDBCRowSet` class provides a single no-args public constructor. Since a `JDBCRowSet` object is a JavaBeans component, instances will often be created by a visual Beans development tool when an application is being assembled.

9.2.3 Retrieving data

Rows are retrieved by calling the `execute()` method which may take an existing `Connection` object as an argument. When `execute()` is called a `JDBCRowSet` object opens a JDBC `Connection` internally, if needed, which remains open until `close()` is called.

9.2.4 Accessing and modifying data

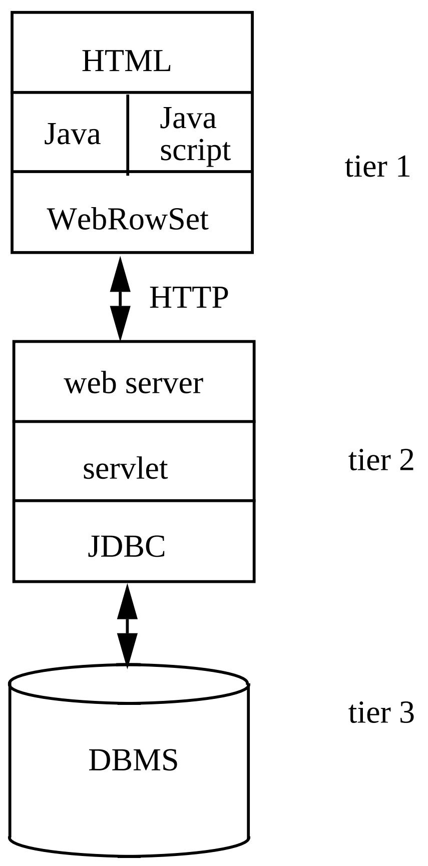
Operations such as cursor movement and updating are performed using methods inherited from the `ResultSet` interface. These methods simply delegate to an underlying JDBC `ResultSet` object which is maintained internally.

9.3 The WebRowSet class

9.3.1 Overview

The `WebRowSet` class represents a distributed client/server implementation of the `RowSet` interface. Conceptually, a `WebRowSet` object is connected, however, the HTTP protocol is used internally to talk to a Java servlet that provides data access. The purpose of the `WebRowSet` class is to provide data access to thin web clients that don't need full JDBC, but just need to get access to a set of rows and possibly update some of them. The `WebRowSet` class will provide incremental caching and uncaching of rows, unlike the `CachedRowSet` class (see Section 9.1) which maintains all data in memory.

The figure below shows an example 3-tier application architecture that uses the `WebRowSet` class. The client, or first tier, is a web client that uses a combination of HTML and Java or Java script to do its work. The `WebRowSet` class is used by the client for accessing rows of tabular data. The middle-tier consists of a webserver that supports Java servlets. The server portion of the `WebRowSet` is a servlet that runs in the middle tier and uses a standard JDBC driver to access an underlying DBMS. The `WebRowSet` implementation includes the `WebRowSet` class, the servlet, and a protocol for transmitting tabular data (possibly based on XML).



9.3.2 Creating a WebRowSet

The `WebRowSet` class provides a single no-args public constructor. Since a `WebRowSet` object is a JavaBeans component, instances will often be created by a visual Beans development tool when an application is being assembled.

9.3.3 Retrieving data

Rows are retrieved by calling the `execute()` method which is inherited from the `RowSet` interface. When `execute()` is called a servlet is invoked to read some initial data for the `WebRowSet` object.

9.3.4 Accessing and modifying data

Operations such as cursor movement and updating are performed using methods inherited from the `ResultSet` interface. Internally, these methods may invoke a servlet to retrieve additional data, or propagate updates to the underlying data source.

Appendix A: Implementing a JDBC Data Source

This appendix describes an example JDBC data source implementation. This appendix is intended mainly for JDBC driver vendors who will provide data source implementations as part of their JDBC 2.0 products.

A.1 A Sample Data Source

The `SampleDataSource` class below implements the `javax.sql.DataSource` interface. The `javax.sql.DataSource` interface provides a pair of `getConnection()` methods that can be invoked directly by user-level application code.

The `SampleDataSource` class also implements the `javax.naming.Referenceable` and `java.io.Serializable` interfaces. These interfaces make it possible for instances of the `SampleDataSource` class to be bound in a JNDI namespace. JNDI supports two separate mechanisms for storing objects that are bound in a namespace. The first is based on the `javax.naming.Referenceable` interface. The second approach is simply to store a serializable Java object as a sequence of bytes. For maximum portability a data source implementation should support both styles.

```
package com.SampleCompanyName.Jdbc;

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

public class SampleDataSource implements javax.sql.DataSource,
                           javax.naming.Referenceable,
                           java.io.Serializable {

  /*
   * Constructors
   */
  public SampleDataSource() {
    // This constructor is needed by the object factory
  }


  /*
   * Properties
   */
  public String getServerName() { return serverName; }
  public void setServerName(String serverName) {
    this.serverName = serverName;
  }

  public String getDatabaseName() { return databaseName; }
  public void setDatabaseName(String databaseName) {
    this.databaseName = databaseName;
  }
```

```
    /*
     * Methods inherited from DataSource
     */
    public Connection getConnection() throws SQLException {

      //vendor specific code to create a JDBC Connection goes here

    }

    public Connection getConnection(String username,
          String password) throws SQLException {

       //vendor specific code to create a JDBC Connection goes here

    }

    public java.io.PrintWriter getLogWriter() throws SQLException {
          //vendor specific code goes here
    }

    public void setLogWriter(java.io.PrintWriter out)
          throws SQLException {
          //vendor specific code goes here
    }

    public void setLoginTimeout(int seconds) throws SQLException {
          //vendor specific code goes here
    }

    public int getLoginTimeout() throws SQLException {
          //vendor specific code goes here
    }

    /*
     * Methods inherited from referenceable
     */
    public Reference getReference() throws NamingException {
      Reference ref = new Reference(this.getClass().getName(),
          "com.SampleCompanyName.Jdbc.SampleObjectFactory",
          null);

      ref.add( new StringRefAddr("serverName", getServerName()) );
      ref.add( new StringRefAddr("databaseName",
          getDatabaseName()) );

      return ref;
    }

    private String serverName = null;
    private String databaseName = null;
  }
```

The `javax.naming.Referenceable` interface contains a single method, `getReference()`, which is called by JNDI to obtain a `Reference` for a data source object when that object is bound in a JNDI naming service. The `Reference` object contains all of the information needed to reconstruct the data source object when it is later retrieved from JNDI. The component that actually reconstructs a data source object when it is retrieved from JNDI is called an *object factory*. References are needed since many naming services don't have the ability to store Java objects in their serialized form. When a data source object is bound in this type of naming service the `Reference` for that object is actually stored by the JNDI implementation, not the data source object itself.

The `SampleDataSource.getReference()` method begins by creating a new `Reference` object to represent the data source. The class name of the data source object is saved in the `Reference`, so that an object factory will know that it should create an instance of that class when a lookup operation is performed. In this example, the class name of the object factory, `com.SampleCompanyName.Jdbc.SampleObjectFactory`, is also stored in the reference. This is not required by JNDI, but we recommend this practice. JNDI will always use the object factory class specified in the reference when reconstructing an object, if a class name has been specified. See the JNDI SPI documentation for further details on this topic, and for a complete description of the `Reference` and `StringRefAddr` classes.

The `SampleDataSource` class provides two standard JDBC properties: serverName and databaseName. The names and values of the data source properties are also stored in the reference using the `StringRefAddr` class. This is all the information needed to reconstruct a `SampleDataSource` object.

The other methods on the `SampleDataSource` class should be self-explanatory. There is a single no-args public constructor which is used by the object factory. There are getter and setter methods for the two properties that `SampleDataSource` supports. The `getConnection()` methods are inherited from `javax.sql.DataSource`, and in a real implementation would contain code for creating a database connection to the particular type of database supported by this JDBC driver.

A.2 Sample Object Factory

An object factory implements the `javax.naming.spi.ObjectFactory` interface. This interface contains a single method, `getObjectInstance`, which is called by a JNDI service provider to reconstruct an object when that object is retrieved from JNDI. A JDBC driver vendor should provide an object factory as part of their JDBC 2.0 product. A JNDI administrator is responsible for making sure that both the object factory and data source implementation classes provided by a JDBC driver vendor are accessible to the JNDI service provider at runtime.

```
package com.SampleCompanyName.Jdbc;

import java.util.*;
import javax.naming.*;
import javax.naming.spi.*;
```

```
public class SampleObjectFactory implements ObjectFactory {

  public Object getObjectInstance(Object refObj,
                                     Name name,
                                     Context nameCtx,
                                     Hashtable env)
    throws Exception {

      Reference ref = (Reference)refObj;

      if (ref.getClassName().equals(
          "com.SampleCompanyName.Jdbc.SampleDataSource")) {
        SampleDataSource sds = new SampleDataSource();

        sds.setServerName((String)ref.get("serverName").
                  getContent());
        sds.setDatabaseName((String)ref.get("databaseName").
                  getContent());

        return sds;
      }
      else {
        return null;
      }
  }
}
```

The `getObjectInstance()` method is passed a reference that corresponds to the object being retrieved as its first parameter. The other parameters are optional in the case of JDBC data source objects. The object factory should use the information contained in the reference to reconstruct the data source. If for some reason, a data source object cannot be reconstructed from the reference, a value of null may be returned. This allows other object factories that may be registered in JNDI to be tried. If an exception is thrown then no other object factories are tried.

See the document “JNDI SPI: Java Naming and Directory Service Provider Interface” for further information regarding the JNDI concepts mentioned in this appendix”.

Appendix B: Implementing Connection Pooling

This appendix describes an example connection pooling implementation. This appendix is intended mainly for system vendors who will provide JDBC connection pooling implementations as part of their products.

9.4 Connection pool example code

The example below sketches how a connection pooling implementation might actually be structured. The example shows one possible approach. There are many alternatives.

```
class ConnectionPoolModule implements javax.sql.DataSource,
                    javax.naming.Referenceable,
                    java.io.Serializable{

  public Connection getConnection(String user,
    String password) throws SQLException {
    ...

    PooledConnection pc = cache.lookup();
    if (pc == null) {
      // There was a miss in the cache, so do a JNDI lookup to
      // get a connection pool data source.
      Context ctx = new InitialContext();
      cpds = ctx.lookup(getDataSourceName());

      pc = cpds.getPooledConnection(user,password);
    }
    pc.addConnectionEventListener(cache);
      ...
    return pc.getConnection();
  }

  private static ConnectionCache cache = null;
  private ConnectionPoolDataSource cpds = null;
  private String dataSourceName = null;

  public String getDataSourceName() { return dataSourceName;}
  public void setDataSourceName(String s) { ... }

  static {
    // Code to initialize the connection cache goes here.
    ...
  }
  ...
}

class ConnectionCache implements ConnectionEventListener {
  ...
}
```

The `ConnectionPoolModule` class implements the `javax.sql.DataSource` interface so that an instance of `ConnectionPoolModule` can be registered with JNDI as a JDBC `DataSource`. Instances of the `ConnectionPoolModule` class will be used directly by applications to create database connections. The `ConnectionCache` class implements the `javax.sql.ConnectionEventListener` interface so that it can be notified when *close* and *error* events occur on a `PooledConnection` that is in use.

`ConnectionPoolModule` supports one standard data source property, DataSource-Name, which is used to locate a connection pool data source when one is needed. A `ConnectionPoolDataSource` is only used when there is a miss in the connection cache. In this case, a second JNDI `lookup()` operation is done to produce a `ConnectionPoolDataSource` object. The `ConnectionPoolDataSource` object is used to create a new `PooledConnection` object.

The private `ConnectionCache` field is declared static so that connections created with different `DataSource` objects can be cached in the same physical connection pool.

Appendix C: Implementing RowSetReader

This appendix contains an example implementation of the `RowSetReader` interface. The example uses JDBC to read data from an SQL database. A rowset that supports the reader/writer paradigm should implement the `RowSetInternal` interface and pass a reference to itself when invoking `readData()`. The example `readData()` method uses the calling rowset's properties to create a `Connection` to a data source. A `Prepared-Statement` is created using the command property and the `PreparedStatement` is passed any properties that were set on the rowset. The example is designed to be used with the `CachedRowSet` rowset implementation. The `CachedRowSet.populate()` method is used to read the rows from a `ResultSet` object into the rowset. The example assumes the caller is of type `CachedRowSet`. An `SQLException` is thrown is anything goes wrong.

```
public class RowSetReaderImpl implements RowSetReader {

  /**
   * This method is called by the rowset internally when
   * the application invokes execute() to read a new set of rows.
   */
  public void readData(RowSetInternal caller) throws SQLException
  {
    Connection con = null;
    PreparedStatement pstmt = null;
    ResultSet rs = null;

    try {

      CachedRowSet crs = (CachedRowSet)caller;

      // Get rid of the current contents of the rowset.
      crs.close();
      writerCalls = 0;

      // Get a connection.  This reader assumes that the necessary
      // properties have been set on the caller to let it supply a
      // connection.
      con = this.connect(caller);

      // Check our assumptions.
      if ( con == null || crs.getCommand() == null )
        throw new SQLException();

      con.setTransactionIsolation(crs.getTransactionIsolation());
      // Use JDBC to read the data.
      pstmt = con.prepareStatement(crs.getCommand());
      // Pass any input parameters to JDBC.
      Object[] params = caller.getParams();
      for (int i=0; i < params.length; i++ )
        pstmt.setObject(i+1, params[i]);
      pstmt.setMaxRows(crs.getMaxRows());
      pstmt.setMaxFieldSize(crs.getMaxFieldSize());
      pstmt.setEscapeProcessing(crs.getEscapeProcessing());
      pstmt.setQueryTimeout(crs.getQueryTimeout());
      rs = pstmt.executeQuery();

      // Get the data.
      crs.populate(rs);

    }
    catch (SQLException ex) {
      // Throw an exception if reading fails for any reason.
      throw ex;
    }
    finally {
        if (rs != null)
          rs.close();
        if (pstmt != null)
          pstmt.close();
        if (con != null && caller.getConnection() == null)
          con.close();
    }
  }

  /**
   * This method is called by the writer to see if it needs to
   * reset its state. The reader is registered with the writer
   * when components are being wired together.
   */
  boolean reset() throws SQLException {
    writerCalls++;
    return writerCalls == 1;
  }

  /**
   * This method is used internally by the reader and writer to
   * make a Connection.
   */
  Connection connect(RowSetInternal caller) throws SQLException {

    // Get a JDBC connection.

    if ( caller.getConnection() != null ) {
      // A connection was passed to execute(), so use it.
      return caller.getConnection();
    }
    else if (((RowSet)caller).getDataSourceName() != null) {
      // Connect using JNDI.
      try {
        Context ctx = new InitialContext();
        DataSource ds = (DataSource)ctx.lookup(
                  ((RowSet)caller).getDataSourceName());
        return ds.getConnection(
                  ((RowSet)caller).getUser(),
                  ((RowSet)caller).getPassword());
      }
      catch (javax.naming.NamingException ex) {
        throw new SQLException();
      }
    }
    else if (((RowSet)caller).getUrl() != null) {
      // Connect using the driver manager.
      return DriverManager.getConnection(
           ((RowSet)caller).getUrl(),
           ((RowSet)caller).getUser(),
           ((RowSet)caller).getPassword());
    }
    else
      return null;
  }

  private int writerCalls = 0;
}
```

An alternative reader implementation could read data directly from a regular file. In this way a rowset can be used to provide access to non-SQL data. a reader of this type can use the `RowSet.insertRow()` method to insert new rows into the rowset. `InsertRow()` updates the original value of the rowset when invoked by a reader. The `RowSetMeta` interface can also be used by such a reader to communicate the format of the data being read to the rowset implementation.

Appendix D: Implementing RowSetWriter

This appendix contains an example implementation of the `RowSetWriter` interface. This writer works in conjunction with the reader shown in the previous appendix. The writer enforces an optimistic concurrency control policy that requires a row in the database to be unchanged for it to be updated. For simplicity, some details are not shown.

The writer calls the reader to create a `Connection`, and to find out if it needs to reset itself. This may be necessary if new data has been read since the command used to read the data may have changed. The writer iterates through each row of the rowset and updates the underlying database when necessary.

```
public class RowSetWriterImpl implements RowSetWriter {

  /**
   * This method is called by the rowset internally when
   * the application invokes acceptUpdates().
   */
  public boolean writeData(RowSetInternal caller)
        throws SQLException {

    int i = 0, j = 0;
    PreparedStatement pstmt = null;
    boolean conflict = false;
    boolean showDel = false;

    // We assume caller is a CachedRowSet
    CachedRowSet crs = (CachedRowSet)caller;

    // The reader is registered with the writer at design time.
    // This is not required, in general.  The reader has logic
    // to get a JDBC connection, so call it.
    con = reader.connect(caller);
    if ( con == null )
      throw new SQLException();
    con.setAutoCommit(false);
    con.setTransactionIsolation(crs.getTransactionIsolation());

    // Ask the reader if the writer needs to initialize it's state.
    // This may be necessary if new contents have been read since
    // the writer was called last.
    if ( reader.reset() )
      initialize(crs);

    if ( callerColumnCount < 1 ) {
      // No data, so return success.
      con.close();
      return true;
    }

    // We need to see rows marked for deletion.
```

```
showDel = crs.getShowDeleted();
crs.setShowDeleted(true);
// Look at all the rows.
crs.beforeFirst();
while (crs.next()) {
  if (crs.rowUpdated()) {
    // The row has been updated.

    // Select the row from the database.
    pstmt = con.prepareStatement(selectCmd);
    for (i=0; i < keyColumnNums.length; i++)
      pstmt.setObject(i+1, crs.getObject(keyColumnNums[i]));
    pstmt.setMaxRows(crs.getMaxRows());
    pstmt.setMaxFieldSize(crs.getMaxFieldSize());
    pstmt.setEscapeProcessing(crs.getEscapeProcessing());
    pstmt.setQueryTimeout(crs.getQueryTimeout());
    ResultSet rs = pstmt.executeQuery();
    rs.next();

    // Check for a conflict.  There is a conflict if any
    // column has changed.  Other writers may have a
    // different policy.
    ResultSet origVals = caller.getOriginalRow();
    conflict = false;
    for (i=1; i <= callerColumnCount; i++) {
      if (!(rs.getObject(i).equals(origVals.getObject(i)))) {
        conflict = true;
        break;
      }
    }

    // Make sure there was only one row.
    if ( rs.next())
      throw new SQLException();
    rs.close();
    pstmt.close();

    if ( !conflict ) {
      // Do the update.

      pstmt = con.prepareStatement(updateCmd);
      // Pass the new data values.
      for (i=1; i <= callerColumnCount; i++) {
        pstmt.setObject(i, crs.getObject(i));
      }
      // Tell which row to update.
      for (i=callerColumnCount+1,j=0;
           i <= callerColumnCount+keyColumnNums.length;
           i++, j++) {
        pstmt.setObject(i,
              origVals.getObject(keyColumnNums[j]));
      }

      pstmt.setMaxRows(crs.getMaxRows());
```

```
            pstmt.setMaxFieldSize(crs.getMaxFieldSize());
            pstmt.setEscapeProcessing(crs.getEscapeProcessing());
            pstmt.setQueryTimeout(crs.getQueryTimeout());
            pstmt.executeUpdate();
            pstmt.close();
          }
        }
        else if (crs.rowInserted()) {
          // The row has been inserted.
        }
        else if (crs.rowDeleted()) {
          // The row has been deleted.
        }
      }
      con.close();
      crs.setShowDeleted(showDel);

      return conflict;
    }

    /**
     * Sets the reader.
     */
    public void setReader(RowSetReaderImpl reader)
          throws SQLException {
      this.reader = reader;
    }

    /**
     * Gets the reader.
     */
    public RowSetReaderImpl getReader() throws SQLException {
      return reader;
    }

    /**
     * Do some setup work.
     *    1.  Compose a SELECT command.
     *    2.  Compose an UPDATE command.  (not shown)
     *    3.  Compose an INSERT command.  (not shown)
     *    4.  Compose a DELETE command.  (not shown)
     */
    void initialize(CachedRowSet caller) throws SQLException {

      int i = 0;

      ResultSetMetaData callerMd = caller.getMetaData();
      int callerColumnCount = callerMd.getColumnCount();
      if ( callerColumnCount < 1 )
        // No data, so return.
        return;

      // The writer can assume that all columns come from the same
      // table.  Note: The policy for determining updatability is
```

```
// built into the reader.
String catalog = callerMd.getCatalogName(1);
String schema = callerMd.getSchemaName(1);
String table = callerMd.getTableName(1);

//
// Compose a SELECT statement.  There are three parts.
//

// Compose the SELECT clause.
String selectCmd = "SELECT ";
for (i=1; i <= callerColumnCount; i++) {
  selectCmd += callerMd.getColumnName(i);
  if ( i <  callerMd.getColumnCount() )
    selectCmd += ", ";
  else
    selectCmd += " ";
}

// Compose the FROM clause.
// con is set by the calling method
DatabaseMetaData dbmd = con.getMetaData();
if ( dbmd.isCatalogAtStart() ) {
  selectCmd += "FROM " +
    catalog + dbmd.getCatalogSeparator() +
    schema + "." +
    table + " ";
}
else {
  selectCmd += "FROM " +
    schema + "." +
    table + dbmd.getCatalogSeparator() +
    catalog + " ";
}

// Compose the WHERE clause.
selectCmd += "WHERE ";
ResultSet keyRs =
  dbmd.getBestRowIdentifier(catalog, schema, table,
                         DatabaseMetaData.bestRowTransaction,
                         false);
int keyColCount = 0;
Vector keyVec = new Vector();
while (keyRs.next()) {
  // This check may not be needed.
  if ( keyRs.getShort(1) ==
    DatabaseMetaData.bestRowTransaction ) {
    keyColCount++;
    if ( keyColCount > 1 )
      selectCmd += "AND ";
    selectCmd += keyRs.getString(2) + " = ? ";

    // Need to get the column from the rowset
    keyVec.add( new Integer(
                  caller.findColumn(keyRs.getString(2))));
      }
    }
    Enumeration enum = keyVec.elements();
    keyColumnNums = new int[keyVec.size()];
    i = 0;
    while (enum.hasMoreElements()) {
      keyColumnNums[i++] = (
        (Integer)enum.nextElement()).intValue();
    }

    //
    // Compose an UPDATE statement.  (not shown)
    //

    //
    // Compose an INSERT statement.  (not shown)
    //

    //
    // Compose a DELETE statement.  (not shown)
    //
  }

  private Connection con = null;
  private String selectCmd = null;
  private String updateCmd = null;
  private int callerColumnCount = 0;
  private int[] keyColumnNums = null;
  private RowSetReaderImpl reader = null;
}
```

Appendix E: RowSet Implementation Classes

This appendix contains a listing of the `CachedRowSet`, `JDBCRowSet`, and `WebRowSet` classes which are discussed in Chapter 9, and their methods. This reference is provided to make the descriptions in Chapter 9 more concrete. The classes and methods listed in this appendix are not part of this release of the JDBC 2.0 Standard Extension API, and are likely to change.

```
public class CachedRowSet implements RowSet, Serializable,
                                     Cloneable {

  public CachedRowSet() throws SQLException

  public boolean getShowDeleted() throws SQLException
  public void setShowDeleted(boolean value) throws SQLException
  public void populate(ResultSet data) throws SQLException
  public void execute(Connection connection)
  public void acceptChanges() throws SQLException
  public void acceptChanges(Connection con) throws SQLException
  public void restoreOriginal() throws SQLException
  public void release() throws SQLException
  public void cancelRowDelete () throws SQLException
  public void cancelRowInsert () throws SQLException
  public RowSet createShared() throws SQLException
  public Object clone()
  public RowSet createCopy() throws SQLException
  public Collection toCollection() throws SQLException
  public Collection toCollection(int column) throws SQLException
  public RowSetReader getReader() throws SQLException
  public void setReader(RowSetReader reader) throws SQLException
  public RowSetWriter getWriter() throws SQLException
  public void setWriter(RowSetWriter writer) throws SQLException
}

public class JDBCRowSet implements RowSet {

  public JDBCRowSet() throws SQLException {}
  void execute(Connection con) throws SQLException {}
}


public class WebRowSet implements RowSet {
}
```

Appendix F: Change History

F.1 Changes between 0.10 and 0.20

- added CachedRowSet, WebRowSet, and JDBCRowSet types
- added RowSetReader, RowSetWriter, and RowSetInternal types
- added Appendices describing implementation strategies
- some other miscellaneous changes

F.2 Changes between 0.20 and 0.70

- added logWriter and loginTimeout properties to the DataSource interface
- added Section 5.6
- Section 5.7: changed description of the relationship between the DataSource and DriverManager facilities
- Chapter 4: added figures
- Section 5.3.1: removed the standard DriverName property
- Chapter 6: added additional descriptions
- Chapter 7: added additional descriptions

F.3 Changes between 0.70 and 1.0

- Chapter 5: data source property names now begin with lower-case
- Chapter 5: changed the *userName* property to *user* for compatibility with existing `DriverManager` properties
- Chapter 9: Made the CachedRowSet, JDBCRowSet, and WebRowSet classes part of a future release, but retained their descriptions.