Oracle9*i* JDBC Developer's Guide and Reference
Release 2 (9.2)
Part Number A96654-01

16 Connection Pooling and Caching

This chapter covers the Oracle JDBC implementations of (1) data sources, a standard facility for specifying resources to use, including databases; (2) connection pooling, which is a framework for caches of database connections; and (3) connection caching, including documentation of a sample Oracle implementation. You will also find related discussion of Oracle JDBC support for the standard Java Naming and Directory Interface (JNDI).

The following topics, which apply to all Oracle JDBC drivers, are described in this chapter:

Notes:

This chapter describes features of the Sun Microsystems JDBC 2.0 Standard Extension API, which are available through the `javax` packages from Sun Microsystems. These packages are not part of the standard JDK, but relevant packages are included with the `classes111.zip` and `classes12.zip` files.

For further information on listed topics, refer to the Sun Microsystems specification for the JDBC 2.0 Standard Extension API. For information about additional connection pooling functionality specific to the OCI driver, see "OCI Driver Connection Pooling".

Data Sources

The JDBC 2.0 extension API introduced the concept of *data sources*, which are standard, general-use objects for specifying databases or other resources to use. Data sources can optionally be bound to Java Naming and Directory Interface (JNDI) entities so that you can access databases by logical names, for convenience and portability.

This functionality is a more standard and versatile alternative to the connection functionality described under "Open a Connection to a Database". The data source facility provides a complete replacement for the previous JDBC `DriverManager` facility.

You can use both facilities in the same application, but ultimately developers will be encouraged to use data sources for their connections, regardless of whether connection pooling or distributed transactions are required. Eventually, Sun Microsystems will probably deprecate `DriverManager` and related classes and functionality.

For further introductory and general information about data sources and JNDI, refer to the Sun Microsystems specification for the JDBC 2.0 Optional Package.

A Brief Overview of Oracle Data Source Support for JNDI

The standard Java Naming and Directory Interface, or JNDI, provides a way for applications to find and access remote services and resources. These services can be any enterprise services, but for a JDBC application would include database connections and services.

JNDI allows an application to use logical names in accessing these services, removing vendor-specific syntax from application code. JNDI has the functionality to associate a logical name with a particular source for a desired service.

All Oracle JDBC data sources are JNDI-referenceable. The developer is not required to use this functionality, but accessing databases through JNDI logical names makes the code more portable.

Note:

Using JNDI functionality requires the file `jndi.jar` to be in the `CLASSPATH`. This file is included with the Java products on the Oracle9*i* CD, but is not included in the `classes12.zip` and `classes111.zip` files. You must add it to the `CLASSPATH` separately. (You can also obtain it from the Sun Microsystems Web site, but it is advisable to use the version from Oracle, because that has been tested with the Oracle drivers.)

Data Source Features and Properties

"First Steps in JDBC" includes sections on how to use the JDBC `DriverManager` class to register driver classes and open database connections. The problem with this model is that it requires your code to include vendor-specific class names, database URLs, and possibly other properties, such as machine names and port numbers.

With data source functionality, using JNDI, you do not need to register the vendor-specific JDBC driver class name, and you can use logical names for URLs and other properties. This allows your application code for opening database connections to be portable to other environments.

Data Source Interface and Oracle Implementation

A JDBC data source is an instance of a class that implements the standard `javax.sql.DataSource` interface:

```
public interface DataSource
{
   Connection getConnection() throws SQLException;
   Connection getConnection(String username, String password)
      throws SQLException;
   ...
}
```

Oracle implements this interface with the `OracleDataSource` class in the `oracle.jdbc.pool` package. The overloaded `getConnection()` method returns an `OracleConnection` instance, optionally taking a user name and password as input.

To use other values, you can set properties using appropriate setter methods discussed in the next section. For alternative user names and passwords, you can also use the `getConnection()` signature that takes these as input--this would take priority over the property settings.

Note:

The `OracleDataSource` class and all subclasses implement the `java.io.Serializable` and `javax.naming.Referenceable` interfaces.

Data Source Properties

The `OracleDataSource` class, as with any class that implements the `DataSource` interface, provides a set of properties that can be used to specify a database to connect to. These properties follow the JavaBeans design pattern.

Table 16-1 and Table 16-2 document `OracleDataSource` properties. The properties in Table 16-1 are standard properties according to the Sun Microsystems specification. (Be aware, however, that Oracle does not implement the standard `roleName` property.) The properties in Table 16-2 are Oracle extensions.

Table 16-1 Standard Data Source Properties

Name	Type	Description
`databaseName`	`String`	name of the particular database on the server; also known as the "SID" in Oracle terminology
`dataSourceName`	`String`	name of the underlying data source class (for connection pooling, this is an underlying pooled connection data source class; for distributed transactions, this is an underlying XA data source class)
`description`	`String`	description of the data source
`networkProtocol`	`String`	network protocol for communicating with the server; for Oracle, this applies only to the OCI drivers and defaults to tcp (Other possible settings include ipc. See the *Oracle Net Services Administrator's Guide* for more information.)
`password`	`String`	login password for the user name
`portNumber`	`int`	number of the port where the server listens for requests
`serverName`	`String`	name of the database server
`user`	`String`	name for the login account

The `OracleDataSource` class implements the following setter and getter methods for the standard properties:

- `public synchronized void setDatabaseName(String dbname)`
- `public synchronized String getDatabaseName()`
- `public synchronized void setDataSourceName(String dsname)`
- `public synchronized String getDataSourceName()`
- `public synchronized void setDescription(String desc)`
- `public synchronized String getDescription()`
- `public synchronized void setNetworkProtocol(String np)`
- `public synchronized String getNetworkProtocol()`
- `public synchronized void setPassword(String pwd)`
- `public synchronized void setPortNumber(int pn)`

- `public synchronized int getPortNumber()`
- `public synchronized void setServerName(String sn)`
- `public synchronized String getServerName()`
- `public synchronized void setUser(String user)`
- `public synchronized String getUser()`

Note that there is no `getPassword()` method, for security reasons.

Table 16-2 Oracle Extended Data Source Properties

Name	Type	Description
`driverType`	`String`	This designates the Oracle JDBC driver type as either oci, thin, or kprb (server-side internal).
`tnsEntry`	`String`	This is the TNS entry name, relevant only for the OCI driver. It assumes an Oracle client installation with a TNS_ADMIN environment variable that is set appropriately. Enable this OracleXADataSource property when using the HeteroRM feature with the OCI driver, to access Oracle pre-8.1.6 databases and higher. The HeteroRM XA feature is described in "OCI HeteroRM XA". If the tnsEntry property is not set when using the HeteroRM XA feature, an SQLException with error code ORA-17207 is thrown.
`url`	`String`	This is the URL of the database connect string. Provided as a convenience, it can help you migrate from an older Oracle database. You can use this property in place of the Oracle tnsEntry and driverType properties and the standard portNumber, networkProtocol, serverName, and databaseName properties.
nativeXA	boolean	Enable this OracleXADataSource property when using the HeteroRM feature with the OCI driver, to access Oracle pre-8.1.6 databases and higher. The HeteroRM XA feature is described in "OCI HeteroRM XA". If the nativeXA property is enabled, be sure to set the tnsEntry property as well. This DataSource property defaults to false.

Note:

Since nativeXA performs better than JavaXA, use nativeXA whenever possible.

The `OracleDataSource` class implements the following `setXXX()` and `getXXX()` methods for the Oracle extended properties:

- `public synchronized void setDriverType(String dt)`
- `public synchronized String getDriverType()`
- `public synchronized void setURL(String url)`
- `public synchronized String getURL()`
- `public synchronized void setTNSEntryName(String tns)`
- `public synchronized String getTNSEntryName()`
- `public synchronized void setNativeXA(boolean nativeXA)`
- `public synchronized boolean getNativeXA()`

If you are using the server-side internal driver--`driverType` property is set to `kprb`--then any other property settings are ignored.

If you are using a Thin or OCI driver, note the following:

- A URL setting can include settings for `user` and `password`, as in the following example, in which case this takes precedence over individual `user` and `password` property settings:

  ```
  jdbc:oracle:thin:scott/tiger@localhost:1521:orcl
  ```

- Settings for `user` and `password` are required, either directly, through the URL setting, or through the `getConnection()` call. The `user` and `password` settings in a `getConnection()` call take precedence over any property settings.
- If the `url` property is set, then any `tnsEntry`, `driverType`, `portNumber`, `networkProtocol`, `serverName`, and `databaseName` property settings are ignored.
- If the `tnsEntry` property is set (which presumes the `url` property is not set), then any `databaseName`, `serverName`, `portNumber`, and `networkProtocol` settings are ignored.
- If you are using an OCI driver (which presumes the `driverType` property is set to `oci`) and the `networkProtocol` is set to `ipc`, then any other property settings are ignored.

Creating a Data Source Instance and Connecting (without JNDI)

This section shows an example of the most basic use of a data source to connect to a database, without using JNDI functionality. Note that this requires vendor-specific, hard-coded property settings.

Create an `OracleDataSource` instance, initialize its connection properties as appropriate, and get a connection instance as in the following example:

```
...
OracleDataSource ods = new OracleDataSource();

ods.setDriverType("oci");
ods.setServerName("dlsun999");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");

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

Or optionally override the user name and password:

```
...
Connection conn = ods.getConnection("bill", "lion");
...
```

Creating a Data Source Instance, Registering with JNDI, and Connecting

This section exhibits JNDI functionality in using data sources to connect to a database. Vendor-specific, hard-coded property settings are required only in the portion of code that binds a data source instance to a JNDI logical name. From that point onward, you can create portable code by using the logical name in creating data sources from which you will get your connection instances.

Note:

Creating and registering data sources is typically handled by a JNDI administrator, not in a JDBC application.

Initialize Connection Properties

Create an `OracleDataSource` instance, and then initialize its connection properties as appropriate, as in the following example:

```
...
OracleDataSource ods = new OracleDataSource();

ods.setDriverType("oci");
ods.setServerName("dlsun999");
ods.setNetworkProtocol("tcp");
ods.setDatabaseName("816");
ods.setPortNumber(1521);
ods.setUser("scott");
ods.setPassword("tiger");
...
```

Register the Data Source

Once you have initialized the connection properties of the `OracleDataSource` instance `ods`, as shown in the preceding example, you can register this data source instance with JNDI, as in the following example:

```
...
Context ctx = new InitialContext();
ctx.bind("jdbc/sampledb", ods);
...
```

Calling the JNDI `InitialContext()` constructor creates a Java object that references the initial JNDI naming context. System properties that are not shown instruct JNDI which service provider to use.

The `ctx.bind()` call binds the `OracleDataSource` instance to a logical JNDI name. This means that anytime after the `ctx.bind()` call, you can use the logical name `jdbc/sampledb` in opening a connection to the database described by the properties of the `OracleDataSource` instance `ods`. The logical name `jdbc/sampledb` is logically bound to this database.

The JNDI name space has a hierarchy similar to that of a file system. In this example, the JNDI name specifies the subcontext `jdbc` under the root naming context and specifies the logical name `sampledb` within the `jdbc` subcontext.

The `Context` interface and `InitialContext` class are in the standard `javax.naming` package.

Notes:

The JDBC 2.0 Specification requires that all JDBC data sources be registered in the `jdbc` naming subcontext of a JNDI namespace or in a child subcontext of the `jdbc` subcontext.

Open a Connection

To perform a lookup and open a connection to the database logically bound to the JNDI name, use the logical JNDI name. Doing this requires casting the lookup result (which is otherwise simply a Java `Object`) to a new `OracleDataSource` instance and then using its `getConnection()` method to open the connection.

Here is an example:

```
...
OracleDataSource odsconn = (OracleDataSource)ctx.lookup("jdbc/sampledb");
Connection conn = odsconn.getConnection();
...
```

Logging and Tracing

The data source facility offers a way to register a character stream for JDBC to use as output for error logging and tracing information. This facility allows tracing specific to a particular data source instance. If you want all data source instances to use the same character stream, then you must register the stream with each data source instance individually.

The `OracleDataSource` class implements the following standard data source methods for logging and tracing:

- `public synchronized void setLogWriter(PrintWriter pw)`
- `public synchronized PrintWriter getLogWriter()`

The `PrintWriter` class is in the standard `java.io` package.

Notes:

- When a data source instance is created, logging is disabled by default (the log stream name is initially null).
- Messages written to a log stream registered to a data source instance are not written to the log stream normally maintained by `DriverManager`.
- An `OracleDataSource` instance obtained from a JNDI name lookup will not have its `PrinterWriter` set, even if the `PrintWriter` was set when a data source instance was first bound to this JNDI name.

Connection Pooling

10.5

Connection pooling in the JDBC 2.0 extension API is a framework for caching database connections. This allows reuse of physical connections and reduced overhead for your application. Connection pooling functionality minimizes expensive operations in the creation and closing of sessions.

The following are central concepts:

- *Connection pool data sources*--similar in concept and functionality to the data sources described previously, but with methods to return *pooled connection* instances, instead of normal connection instances.
- *Pooled connections*--a pooled connection instance represents a single physical connection to a database, remaining open during use by a series of *logical connection instances*.

  A logical connection instance is a simple connection instance (such as a standard `Connection` instance or an `OracleConnection` instance) returned by a pooled connection instance. Each logical connection instance acts as a temporary handle to the physical connection represented by the pooled connection instance.

For connection pooling information specific to OCI drivers, see "OCI Driver Connection Pooling". For further introductory and general information about connection pooling, refer to the Sun Microsystems specification for the

JDBC 2.0 Optional Package.

Note:

The concept of connection pooling is not relevant to the server-side internal driver, where you are simply using the default connection, and is only relevant to the server-side Thin driver within a single session.

Connection Pooling Concepts

If you do not use connection pooling, each connection instance (`java.sql.Connection` or `oracle.jdbc.OracleConnection` instance) encapsulates its own physical database connection. When you call the `close()` method of the connection instance, the physical connection itself is closed. This is true whether you obtain the connection instance through the JDBC 2.0 data source facility described under "Data Sources", or through the `DriverManager` facility described under "Open a Connection to a Database".

With connection pooling, an additional step allows physical database connections to be reused by multiple logical connection instances, which are temporary handles to the physical connection. Use a connection pool data source to return a pooled connection, which is what encapsulates the physical database connection. Then use the pooled connection to return JDBC connection instances (one at a time) that each act as a temporary handle.

Closing a connection instance that was obtained from a pooled connection does *not* close the physical database connection. It does, however, free the resources of the connection instance, clear the state, close statement objects created from the connection instance, and restore the defaults for the next connection instance that will be created.

To actually close the physical connection, you must invoke the `close()` method of the pooled connection. This would typically be performed in the middle tier.

Connection Pool Data Source Interface and Oracle Implementation

The `javax.sql.ConnectionPoolDataSource` interface outlines standard functionality of connection pool data sources, which are factories for pooled connections. The overloaded `getPooledConnection()` method returns a pooled connection instance and optionally takes a user name and password as input:

```
public interface ConnectionPoolDataSource
{
   PooledConnection getPooledConnection() throws SQLException;
   PooledConnection getPooledConnection(String user, String password)
      throws SQLException;
   ...
}
```

Oracle JDBC implements the `ConnectionPoolDataSource` interface with the `oracle.jdbc.pool.OracleConnectionPoolDataSource` class. This class also extends the `OracleDataSource` class, so it includes all the connection properties and getter and setter methods described in "Data Source Properties".

The `OracleConnectionPoolDataSource` class `getPooledConnection()` methods return the Oracle implementation of pooled connection instances, which are `OraclePooledConnection` instances (as discussed in the next section).

Note:

You can register connection pool data sources in JNDI using the same naming conventions as discussed for non-pooling data sources in "Register the Data Source".

Pooled Connection Interface and Oracle Implementation

A pooled connection instance encapsulates a physical connection to a database. This database would be the one specified in the connection properties of the connection pool data source instance used to produce the pooled connection instance.

A pooled connection instance is an instance of a class that implements the standard `javax.sql.PooledConnection` interface. The `getConnection()` method specified by this interface returns a logical connection instance that acts as a temporary handle to the physical connection, as opposed to encapsulating the physical connection, as does a non-pooling connection instance:

```
public interface PooledConnection
{
   Connection getConnection() throws SQLException;
   void close() throws SQLException;
   void addConnectionEventListener(ConnectionEventListener listener) ... ;
   void removeConnectionEventListener(ConnectionEventListener listener);
   void setStmtCacheSize(int size);
   void setStmtCacheSize(int size, boolean clearMetaData);
   int getStmtCacheSize();
}
```

(Event listeners are used in connection caching and are discussed in "Typical Steps in Using a Connection Cache".)

Oracle JDBC implements the `PooledConnection` interface with the `oracle.jdbc.pool.OraclePooledConnection` class. The `getConnection()` method returns an `OracleConnection` instance.

A pooled connection instance will typically be asked to produce a series of connection instances during its existence, but only one of these connection instances can be open at any particular time.

Each time a pooled connection instance `getConnection()` method is called, it returns a new connection instance that exhibits the default behavior, and it closes any previous connection instance that still exists and has been returned by the same pooled connection instance. You should explicitly close any previous connection instance before opening a new one, however.

Calling the `close()` method of a pooled connection instance closes the physical connection to the database. The middle-tier layer typically performs this.

The `OraclePooledConnection` class includes methods to enable statement caching for a pooled connection. The cache for statements is maintained for the pooled connection as a whole, and all logical connections obtained from the pooled connection share it. Therefore, when statement caching is enabled, a statement you create on one logical connection can be re-used on another logical connection. For the same reason, you cannot enable or disable statement caching on individual logical connections. This function applies to both implicit and explicit statement caching.

The following are `OraclePooledConnection` method definitions for statement caching:

```
public void setStmtCacheSize (int size)
   throws SQLException

public void setStmtCacheSize (int size, boolean clearMetaData)
   throws SQLException

public int getStmtCacheSize()
```

See Chapter 14, "Statement Caching", for more details on **statement caching.**

Creating a Connection Pool Data Source and Connecting

This section contains an example of the most basic use of a connection pool data source to connect to a database without using JNDI functionality. You could optionally use JNDI, binding the connection pool data source instance to a JNDI logical name, in the same way that you would for a generic data source instance (as "Register the Data Source" illustrates).

Summary of Imports for Oracle Connection Pooling

You must import the following for Oracle connection pooling functionality:

```
import oracle.jdbc.pool.*;
```

This package contains the `OracleDataSource`, `OracleConnectionPoolDataSource`, and `OraclePooledConnection` classes, in addition to classes for connection caching and event-handling, which "Connection Caching" discusses.

Oracle Connection Pooling Code Sample

This example first creates an `OracleConnectionPoolDataSource` instance, next initializes its connection properties, then gets a pooled connection instance from the connection pool data source instance, and finally gets a connection instance from the pooled connection instance. (The `getPooledConnection()` method actually returns an `OraclePooledConnection` instance, but in this case only generic `PooledConnection` functionality is required.)

```
...
OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();

ocpds.setDriverType("oci");
ocpds.setServerName("dlsun999");
ocpds.setNetworkProtocol("tcp");
ocpds.setDatabaseName("816");
ocpds.setPortNumber(1521);
ocpds.setUser("scott");
ocpds.setPassword("tiger");

PooledConnection pc = ocpds.getPooledConnection();

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

Connection Caching

Connection caching, generally implemented in a middle tier, is a means of keeping and using caches of physical database connections.

Connection caching uses the connection pooling framework--such as connection pool data sources and pooled connections--in much of its operations. "Connection Pooling", starting, describes this framework.

The JDBC 2.0 specification does not mandate a connection caching implementation, but Oracle provides a simple implementation to serve at least as an example.

This section is divided into the following topics:

- Overview of Connection Caching
- Typical Steps in Using a Connection Cache
- Oracle Connection Cache Specification: OracleConnectionCache Interface

---

Note:

The concept of connection caching is not relevant to the server-side internal driver, where you are simply using the default connection, and is only relevant to the server-side Thin driver within a single session.

Overview of Connection Caching

Each connection cache is represented by an instance of a *connection cache class* and has an associated group of pooled connection instances. For a single connection cache instance, the associated pooled connection instances must all represent physical connections to the same database and schema. Pooled connection instances are created as needed, which is whenever a connection is requested and the connection cache does not have any free pooled connection instances. A "free" pooled connection instance is one that currently has no logical connection instance associated with it; in other words, a pooled connection instance whose physical connection is not being used.

Basics of Setting Up a Connection Cache

The middle tier, in setting up a connection cache, will create an instance of a connection cache class and set its data source connection properties as appropriate--for example, `serverName`, `databaseName`, or `URL`. Recall that a connection cache class extends a data source class. For information about data source properties, see "Data Source Properties".

An example of a connection cache class is `OracleConnectionCacheImpl`. How to instantiate this class and set its connection properties is described in "Instantiating OracleConnectionCacheImpl and Setting Properties". This class extends the `OracleDataSource` class and so includes the setter methods to set connection properties to specify the database to connect to. All the pooled connection instances in the cache would represent physical connections to this same database, and in fact to the same schema.

Once the middle tier has created a connection cache instance, it can optionally bind this instance to JNDI as with any data source instance, which is described in "Register the Data Source".

Basics of Accessing the Connection Cache

A JDBC application must retrieve a connection cache instance to use the cache. This is typically accomplished through the middle tier, often using a JNDI lookup. In a connection caching scenario, a JNDI lookup would return a connection cache instance instead of a generic data source instance. Because a connection cache class extends a data source class, connection cache instances include data source functionality.

Executing a JNDI lookup is described in "Open a Connection".

If JNDI is not used, the middle tier will typically have some vendor-specific API through which a connection cache instance is retrieved for the application.

Basics of Opening Connections

A connection cache class, as with a pooled connection class, has a `getConnection()` method. The `getConnection()` method of a connection cache instance returns a logical connection to the database and schema associated with the cache. This association is through the connection properties of the connection cache instance, as typically set by the middle tier.

Whenever a JDBC application wants a connection to a database in a connection caching scenario, it will call the `getConnection()` method of the connection cache instance associated with the database.

This `getConnection()` method checks if there are any free pooled connection instances in the cache. If not, one is created. Then a logical connection instance will be retrieved from a previously existing or newly created pooled connection instance, and this logical connection instance will be supplied to the application.

Basics of Closing Connections: Use of Connection Events

JDBC uses JavaBeans-style events to keep track of when a physical connection (pooled connection instance) can be returned to the cache or when it should be closed due to a fatal error. When a JDBC application calls the `close()` method of a logical connection instance, an event is triggered and communicated to the event listener or listeners associated with the pooled connection instance that produced the logical connection instance. This triggers a connection-closed event and informs the pooled connection instance that its physical connection can be reused. Essentially, this puts the pooled connection instance and its physical connection back into the cache.

The point at which a connection event listener is created and registered with a pooled connection instance is implementation-specific. This could happen, for example, when the pooled connection instance is first created or each time the logical connection associated with it is closed.

It is also possible for the cache class to implement the connection event listener class. In this case, the connection event listener is part of the connection cache instance. (This is not the case in the Oracle sample implementation.) Even in this case, however, an explicit association must be made between the connection event listener and each pooled connection instance.

Basics of Connection Timeout

Some application developers prefer to have connections released automatically after a certain timespan has passed. This prevents slow resource leaks when an application fails to close connections.

An application can specify any of the following timeout periods for a connection:

- **Wait timeout (**`FIXED_WAIT_SCHEME` only) -- the maximum period that a connection will wait for a physical connection to be returned to the cache. This wait occurs only when all connections are in use and a new connection is requested. When the timeout expires, a timeout exception, `EOJ_FIXED_WAIT_TIMEOUT`, is thrown.
- **Inactivity timeout (physical connections only)** --the maximum period a **physical** connection can be unused. When the period expires, the connection is closed and its resources are freed.
- **Time-to-Live timeout (logical connections only)**--the maximum period a logical connection can be active. After this time expires, whether or not the connection is still in use, the connection is closed and its resources are freed.

Note::

Under the `FIXED_WAIT_SCHEME`, if every connection in the cache is in use, additional connection requests may wait forever. You can avoid this problem by using the `CacheFixedWaitIdleTime` property. This property specifies how long a connection request will wait before it times out. The `ThreadWakeUpInterval` property controls how often the cache thread checks whether a physical connection has become available. `ThreadWakeUpInterval` defaults to 30 seconds.

To set or unset a timeout, you use the `OracleConnectionCacheImpl` properties `CacheInactivityTimeout`, `CacheFixedWaitTimeout`, and `CacheTimeToLiveTimeout`, each of which has public get and set methods.

Implementation Scenarios

Middle-tier developers have the option of implementing their own connection cache class and connection event listener class.

For convenience, however, Oracle provides the following, all in the `oracle.jdbc.pool` package:

- a connection cache interface: `OracleConnectionCache`
- a connection cache class: `OracleConnectionCacheImpl`
- a connection event listener class: `OracleConnectionEventListener`

The `OracleConnectionCacheImpl` class is a simple connection cache class implementation that Oracle supplies as an example, providing sufficient but minimal functionality. It implements the `OracleConnectionCache` interface and uses instances of the `OracleConnectionEventListener` class for connection events.

If you want more functionality than `OracleConnectionCacheImpl` has to offer but still want to use `OracleConnectionEventListener` for connection events, then you can create your own class that implements `OracleConnectionCache`.

Or you can create your own connection cache class and connection event listener class from scratch.

Typical Steps in Using a Connection Cache

This section lists the general steps in how a JDBC application and middle-tier will use a connection cache in opening and closing a logical connection.

Preliminary Steps in Connection Caching

Presume the following has already been accomplished:

1. The middle tier has created a connection cache instance, as described in "Basics of Setting Up a Connection Cache".
2. The middle tier has provided connection information to the connection cache instance for the database and schema that will be used. This can be accomplished when constructing the connection cache instance.
3. The application has retrieved the connection cache instance, as described in "Basics of Accessing the Connection Cache".

General Steps in Opening a Connection

Once the JDBC application has access to the connection cache instance, the application and middle tier perform the following steps to produce a logical connection instance for use by the application:

1. The application requests a connection through the `getConnection()` method of the connection cache instance. No input is necessary, because a connection cache instance is already associated with a particular database and schema.
2. The connection cache instance examines its cache as follows: a) to see if there are any pooled connection instances in the cache yet; and b) if so, if any are free--that is, to see if there is at least one pooled connection instance that currently has no logical connection instance associated with it.

3. The connection cache instance chooses an available pooled connection instance or, if none is available, might create a new one (this is implementation-specific). In creating a pooled connection instance, the connection cache instance can specify connection properties according to its own connection properties, because the pooled connection instance will be associated with the same database and schema.

Note:

Exactly what happens in a situation where no pooled connection instances are available depends on the implementation schemes and whether the cache is limited to a maximum number of pooled connections. For the Oracle sample implementation, this is discussed in "Schemes for Creating New Pooled Connections in the Oracle Implementation".

---

4. Depending on the situation and implementation, the connection cache instance creates a connection event listener (a process that associates the listener with the connection cache instance) and associates the listener with the chosen or newly created pooled connection instance. The association with the pooled connection instance is accomplished by calling the standard `addConnectionEventListener()` method specified by the `PooledConnection` interface. This method takes the connection event listener instance as input. If the connection cache class implements the connection event listener class, then the argument to the `addConnectionEventListener()` method would be the `this` object.

   In some implementations, the creation and association of the connection event listener can occur only when the pooled connection instance is first created. In the Oracle sample implementation, this also occurs each time a pooled connection instance is reused.

   Note that in being associated with both the connection cache instance and a pooled connection instance, the connection event listener becomes the bridge between the two.

5. The connection cache instance gets a logical connection instance from the chosen or newly created pooled connection instance, using the pooled connection `getConnection()` method.

   No input is necessary to `getConnection()`, because a pooled connection instance is already associated with a particular database and schema.

6. The connection cache instance passes the logical connection instance to the application.

The JDBC application uses this logical connection instance as it would any other connection instance.

General Steps in Closing a Connection

Once the JDBC application has finished using the logical connection instance, its associated pooled connection instance can be returned to the connection cache (or closed, as appropriate, if a fatal error occurred). The application and middle tier perform the following steps to accomplish this:

1. The application calls the `close()` method on the logical connection instance (as it would with any connection instance).
2. The pooled connection instance that produced the logical connection instance triggers an event to the connection event listener or listeners associated with it (associated with it through previous calls by the connection cache instance to the pooled connection instance `addConnectionEventListener()` method).
3. The connection event listener performs one of the following:
   - It puts the pooled connection instance back into the cache and flags it as available (typical).

or:

- It closes the pooled connection instance (if a fatal error occurred during use of its physical connection).

The connection event listener will typically perform these steps by calling methods of the connection cache instance, which is implementation-specific. For the Oracle sample implementation, these functions are performed by methods specified in the `OracleConnectionCache` interface, as discussed in"Oracle Connection Cache Specification: OracleConnectionCache Interface".

4. Depending on the situation and implementation, the connection cache instance disassociates the connection event listener from the pooled connection instance. This is accomplished by calling the standard `removeConnectionEventListener()` method specified by the `PooledConnection` interface.

   In some implementations, this step can be performed only when a pooled connection instance is closed, either because of a fatal error or because the application is finished with the physical connection. In the Oracle sample implementation, however, the connection event listener is disassociated with the pooled connection instance each time the pooled connection is returned to the available cache (because in the Oracle implementation, a connection event listener is associated with the pooled connection instance whenever it is reused).

Oracle Connection Cache Specification: OracleConnectionCache Interface

Middle-tier developers are free to implement their own connection caching scheme as desired, but Oracle offers the `OracleConnectionCache` interface, which you can implement in a connection cache class and which uses instances of the `OracleConnectionEventListener` class for its listener functionality.

In addition, Oracle offers a class that implements this interface, `OracleConnectionCacheImpl`, which can be used as is. This class also extends the `OracleDataSource` class and, therefore, includes a `getConnection()` method. For more information about this class, see "Oracle Connection Cache Implementation: OracleConnectionCacheImpl Class".

These Oracle classes and interfaces are all in the `oracle.jdbc.pool` package.

The `OracleConnectionCache` interface specifies the following methods (in addition to data source methods that it inherits), to be implemented in a connection cache class:

- `reusePooledConnection()`: Takes a pooled connection instance as input and returns it to the cache of available pooled connections (essentially, the available physical connections).

  This method would be invoked by a connection event listener after a JDBC application has finished using the logical connection instance provided by the pooled connection instance (through previous use of the pooled connection `getConnection()` method).

- `closePooledConnection()`: Takes a pooled connection instance as input and closes it.

  A connection event listener would invoke this method after a fatal error has occurred through the logical connection instance provided by the pooled connection instance. The listener would call `closePooledConnection()`, for example, if it notices a server crash.

- `close()`: Closes the connection cache instance, after the application has finished using connection caching with the associated database.

The functionality of the `reusePooledConnection()` and `closePooledConnection()` methods is an implementation of some of the steps described generally in "General Steps in Closing a Connection".

Oracle Connection Cache Implementation: OracleConnectionCacheImpl Class

Oracle offers a sample implementation of connection caching and connection event listeners, providing the `OracleConnectionCacheImpl` class. This class implements the `OracleConnectionCache` interface (which you can optionally implement yourself in some other connection cache class) and uses instances of the `OracleConnectionEventListener` class for listener functionality.

These Oracle classes and interfaces are all in the `oracle.jdbc.pool` package.

If you use the `OracleConnectionCacheImpl` class for your connection caching functionality, you should be familiar with the following topics, discussed immediately below:

- Instantiating OracleConnectionCacheImpl and Setting Properties
- Setting a Maximum Number of Pooled Connections
- Setting a Minimum Number of Pooled Connections
- Schemes for Creating New Pooled Connections in the Oracle Implementation
- Additional OracleConnectionCacheImpl Methods

Instantiating OracleConnectionCacheImpl and Setting Properties

A middle tier that uses the Oracle implementation of connection caching can construct an `OracleConnectionCacheImpl` instance and set its connection properties in one of three ways:

- It can use the `OracleConnectionCacheImpl` constructor that takes an existing connection pool data source as input. This is convenient if the middle tier has already created a connection pool data source instance and set its connection properties. For example, where `cpds` is a connection pool data source instance:

  ```
  OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl(cpds);
  ```

or:

- It can use the default `OracleConnectionCacheImpl` constructor (which takes no input) and then the `setConnectionPoolDataSource()` method, which takes an existing connection pool data source instance as input. Again, this is convenient if the middle tier already has a connection pool data source instance with its connection properties set. For example, where `cpds` is a connection pool data source instance:

  ```
  OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl();
  ocacheimpl.setConnectionPoolDataSource(cpds);
  ```

Notes:

- You can also use the `setConnectionPoolDataSource()` method to override a previously set pooled connection data source or previously set connection properties.
- If you call `setConnectionPoolDataSource()` when there is already a connection pool data source with associated logical connections in use, then an exception will be thrown if the new connection pool data source specifies a different database schema than the old connection pool data source.

or:

- It can use the default `OracleConnectionCacheImpl` constructor and then set the properties individually, using setter methods. For example:

```
OracleConnectionCacheImpl ocacheimpl = new OracleConnectionCacheImpl();

ocacheimpl.setDriverType("oci");
ocacheimpl.setServerName("dlsun999");
ocacheimpl.setNetworkProtocol("tcp");
ocacheimpl.setDatabaseName("816");
ocacheimpl.setPortNumber(1521);
ocacheimpl.setUser("scott");
ocacheimpl.setPassword("tiger");
```

This is equivalent to setting properties in any generic data source or connection pool data source, as discussed in "Initialize Connection Properties".

Setting a Maximum Number of Pooled Connections

In any connection caching implementation, the middle-tier developer must decide whether there should be a maximum number of pooled connections in the cache, and how to handle situations where no pooled connections are available and the maximum number has been reached.

The `OracleConnectionCacheImpl` class includes a maximum cache size that you can set using the `setMaxLimit()` method (taking an `int` as input). The default value is 1.

The following is an example that presumes `ocacheimpl` is an `OracleConnectionCacheImpl` instance:

```
ocacheimpl.setMaxLimit(10);
```

This example limits the cache to a maximum size of ten pooled-connection instances.

Setting a Minimum Number of Pooled Connections

Just as the middle-tier developer can set the maximum number of pooled connections, you can also determine if there should be a minimum number of pre-spawned pooled connections in the cache. The minimum number is passed as an argument to the `setMinLimit()` method. If the cache doesn't have the specified number of pooled connections instances, the cache will create the new spooled-connection instances, not exceeding the specified minimum limit. The cache always keeps the minimum number of pooled connections open whether the connections are active or idle.

The following is an example that presumes `ocacheimpl` is an `OracleConnectionCacheImpl` instance:

```
ocacheimpl.setMinLimit(3);
```

The cache, in this example, always has a minimum of three pooled-connection instances.

Schemes for Creating New Pooled Connections in the Oracle Implementation

The `OracleConnectionCacheImpl` class supports three *connection cache schemes*. Use these schemes in situations where (1) the application has requested a connection, (2) all existing pooled connections are in use, and (3) the maximum number of pooled connections in the cache have been reached.

- dynamic

  In this default scheme, you can create new pooled connections above and beyond the maximum limit, but each one is automatically closed and freed as soon as the logical connection instance that it provided is no longer in use. (As opposed to the normal scenario when a pooled connection instance is finished being used, where it is returned to the available cache.)

fixed with no wait

In this scheme, the maximum limit cannot be exceeded. Requests for connections when the maximum has already been reached will return `null`.

- fixed wait

  Same as the "fixed with no wait" scheme except that a request for a new connection will wait if the limit for the number of connections has been reached. In this case, the connection request waits until another client releases a connection.

Set the cache scheme by invoking the `setCacheScheme()` method of the `OracleConnectionCacheImpl` instance.

There are two versions of `setCacheScheme()`, one that takes a string and one that takes an integer.

- The string version is case-insensitive and accepts "`dynamic_scheme`", "fixed_return_null_scheme", or "fixed_wait_scheme" .
- The integer version accepts the class static constants `DYNAMIC_SCHEME`, `FIXED_RETURN_NULL_SCHEME`, or `FIXED_WAIT_SCHEME`.

For example, if `ocacheimpl` is an `OracleConnectionCacheImpl` instance, you could set the cached scheme to fixed with no wait using either the integer version of `setCacheScheme()`:

```
ocacheimpl.setCacheScheme(OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME);
```

or the string version:

```
setCacheScheme("fixed_return_null_scheme")
```

Additional OracleConnectionCacheImpl Methods

In addition to the key methods already discussed in "Oracle Connection Cache Specification: OracleConnectionCache Interface", the following `OracleConnectionCacheImpl` methods may be useful:

- `getActiveSize()`: Returns the number of currently active pooled connections in the cache (pooled connection instances with an associated logical connection instance being used by the JDBC application).
- `getCacheSize()`: Returns the total number of pooled connections in the cache, both active and inactive.

Oracle Connection Event Listener: OracleConnectionEventListener Class

This section discusses `OracleConnectionEventListener` functionality by summarizing its constructors and methods.

Instantiating an Oracle Connection Event Listener

In the Oracle implementation of connection caching, an `OracleConnectionCacheImpl` instance constructs an Oracle connection event listener, specifying the connection cache instance itself (its `this` instance) as the constructor argument. This instance associates the connection event listener with the connection cache instance.

In general, however, the `OracleConnectionEventListener` constructor can take any data source instance as input. For example, where `ds` is a generic data source:

```
OracleConnectionEventListener ocel = new OracleConnectionEventListener(ds);
```

There is also a default constructor that takes no input and can be used in conjunction with the `OracleConnectionEventListener` class `setDataSource()` method:

```
OracleConnectionEventListener ocel = new OracleConnectionEventListener();
...
ocel.setDataSource(ds);
```

The input can be any kind of data source, including an `OracleConnectionCacheImpl` instance (because that class extends `OracleDataSource`).

Oracle Connection Event Listener Methods

This section summarizes the methods of the `OracleConnectionEventListener` class:

- `setDataSource()` (previously discussed): Used to input a data source to the connection event listener, in case one was not provided when constructing the listener. This can take any type of data source as input.
- `connectionClosed()`: Invoked when the JDBC application calls `close()` on its representation of the connection.
- `connectionErrorOccurred()`: Invoked when a fatal connection error occurs, just before a `SQLException` is issued to the application.