There are two performance advantages to this approach:

- Since we maintain a pool of connections (instead of creating and destroying connections with each request), the overhead of establishing database connections goes down dramatically for the case of a large number of users.
- Since the application "reserves" the connection only for the duration of the request, we don't waste connection resources when the application is idle (i.e., between requests). This again translates directly to improved performance and scalability.

Note that this solution scales only if you can map the large number of end users to a much smaller number of actual database users. On a popular website such as `http://www.amazon.com`, the number of users can run into the millions; creating a connection pool for these many different users is not feasible.

For example, imagine our website has 1,000,000 users, and for each user we have to connect as a separate database account. Say we have a connection pool of 100 connections. The connections in this pool are good for use only for the 100 users whose connections exist in the pool currently. The moment a new user tries to connect, the connection pool cannot serve that user, since each user requires connection to a different account. Thus, one of the connections will have to be aged out and a new connection established for this user. Clearly, the purpose of connection pool is defeated if you need to tear down and create a connection every time a new user whose connection is not currently in the pool tries to connect. Contrast this with a scenario where the 1,000,000 users can be "mapped" into 10 database users based on different privileges each user requires. Now the likelihood of a new user getting a "hit" in the connection pool is very high, thus allowing us to share the 100 connections among a much larger end-user population. So how do we map these many users to a manageable number of database users? We will look at this aspect in detail in the next chapter.

Let's now take a brief look at what the term "connection caching" means before we discuss Oracle9*i* connection pooling and caching, followed by Oracle 10*g* implicit connection caching.

What Is Connection Caching?

At the core, connection caching and connection pooling refer to the same concept: pooling physical database connections to be shared across multiple client sessions. *Connection caching*, usually implemented in the middle tier, refers to the concept of creating a cache of physical connections using the connection pooling framework provided by the JDBC driver. In Oracle9*i*, we need to create our own cache on top of the connection pooling framework provided by Oracle JDBC drivers. We can use a sample connection cache provided by Oracle as well. In Oracle 10*g*, the connection cache can be enabled implicitly at the data source level itself, thus obviating the need to maintain or manage our own cache.

The next section discusses in detail the Oracle9*i* implementation of connection pooling and caching.

Oracle9*i* Connection Pooling Framework

The Oracle9*i* connection pooling framework depends on the following key concepts:

- *Connection pool data source*: A connection pool data source is similar in concept and functionality to the data sources described in Chapter 3, but with methods to return *pooled connection* objects, instead of *normal connection* objects.
- *Pooled connection (or physical connection)*: A pooled connection instance represents a single connection to a database that remains open during use by a series of *logical connection* instances.
- *Logical connection*: A logical connection is a connection instance (such as a standard `Connection` instance or an `OracleConnection` instance) returned by a pooled connection instance. It is the pooled connection *checked out* by an application at a given point in time.

When we use connection pooling, we essentially introduce an intermediate step to enable reuse of a physical connection. The connection pool data source returns a pooled connection, which encapsulates the physical database connection. We then use the pooled connection to return JDBC connection instances (one at a time) that each act as a temporary handle (the logical connection).

When an application closes the logical connection, it does not result in the closing of the physical database connection. It does, however, clear the connection state, restore the defaults (e.g., it resets autocommit to `true` if you had set it to `false`), and mark the underlying physical connection (or pooled connection) as "available" for creating the next logical instance from the connection pool.

To actually close the physical connection, you must invoke the `close()` method of the pooled connection. This action is typically performed in the middle tier that manages the connection pool, and not by the application.

Related JDBC Standard and Oracle Interfaces

Figure 14-3 shows the JDBC interfaces related to connection pooling and their Oracle counterparts. Note that as an application developer, you will typically not deal with the `PooledConnection` and `ConnectionPoolDataSource` interfaces described in this section. These interfaces are implemented for you by the connection cache (either the sample Oracle9*i* connection cache or the implicit connection cache in 10*g*). So feel free to skip the section "Oracle9*i* Connection Caching" if you are not interested in this topic.

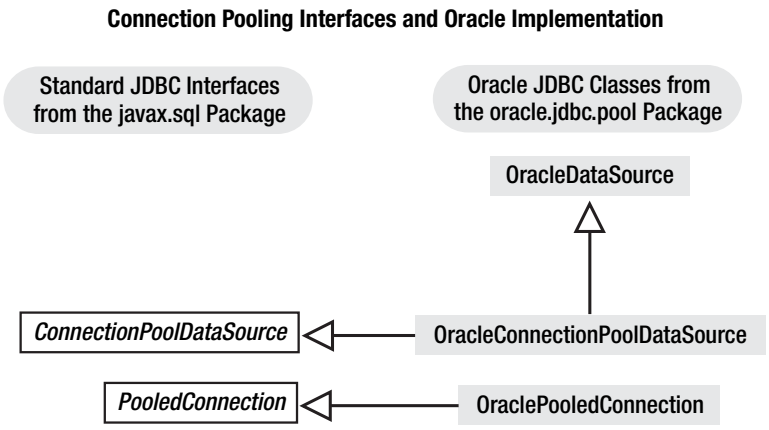

Figure 14-3. *JDBC interfaces that define connection pooling and the Oracle classes that implement them*

The `javax.sql.ConnectionPoolDataSource` interface defines the standard functionality of connection pool data sources. The `getPooledConnection()` method of this interface returns a pooled connection instance and optionally takes a username and password as input in its overloaded version:

```
public javax.sql.PooledConnection getPooledConnection() throws SQLException;
public javax.sql.PooledConnection getPooledConnection(
  String userName, String password ) throws SQLException;
```

As Figure 14-3 indicates, the Oracle class `oracle.jdbc.pool.OracleConnectionPoolDataSource` implements the `ConnectionPoolDataSource` interface. This class also extends the `OracleDataSource` class, so it includes all the connection properties and getter and setter methods described in the section "Connecting to a Database" of Chapter 3. The `getPooledConnection()` method of this class returns an `OraclePooledConnection` class instance, which implements the `PooledConnection` interface.

A pooled connection instance encapsulates a physical connection to the database specified in the connection properties of the connection pool data source instance with which it was created. It implements the following standard `javax.sql.PooledConnection` interface:

```
public interface javax.sql.PooledConnection
{
  public void close() throws SQLException
  public java.sql.Connection getConnection() throws SQLException
  public abstract void addConnectionEventListener(
                       javax.sql.ConnectionEventListener);
  public abstract void removeConnectionEventListener(
                       javax.sql.ConnectionEventListener);
}
```

The `getConnection()` method of this interface returns a logical connection instance to the application. Calling the `close()` method on a pooled connection object closes the physical connection—remember, this is performed by the middle-tier code that manages the connection pool. The connection event listeners are used to handle events that arise when an associated logical connection is closed, for example.

The `OraclePooledConnection` class has methods that enable statement caching (both implicit and explicit) for a pooled connection (see Chapter 13 for details on this feature). All logical connections obtained from a pooled connection share the same cache, since the underlying physical connection is where the caching happens. This implies that when statement caching is enabled, a statement you create on one logical connection can be reused by another logical connection. It follows that you cannot enable or disable statement caching on individual logical connections.

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

```
public boolean getExplicitCachingEnabled();
public boolean getImplicitCachingEnabled();
public int getStatementCacheSize();
public void setExplicitCachingEnabled(boolean);
public void setImplicitCachingEnabled(boolean);
public void setStatementCacheSize(int);
```

Let's move on to look at an example of creating a connection pool data source and getting a pooled connection object.

Example of Creating a Pooled Connection and Obtaining a Logical Connection

In this example, we demonstrate the following concepts:

- Creating a pooled connection
- Getting a logical connection from a pooled connection
- Closing the logical connection
- Closing the pooled connection

To peek behind the scenes, we will run the query that we discussed in the section "Connections and Sessions in Oracle" earlier to list the current physical connections actually made to the database. Note that in this example, there is a one-to-one correspondence between a physical connection and a session.

We need a way to "pause" after each of the four steps in the program, in order to run the query in the database and watch how many connections are created. For this, I wrote a class called `InputUtil` whose `waitTillUserHitsEnter()` method pauses until you press the Enter key. Please see the section "A Utility to Pause in a Java Program" of Chapter 1 for more details on this method.

The following program, DemoConnectionPooling, illustrates how to create a pooled connection and retrieve a logical connection from it. First, the necessary import statements and the declaration of the main() method are shown:

```
/* This program demonstrates how to create a pooled connection
* and obtain a logical connection from it.
* COMPATIBLITY NOTE: runs successfully against 9.2.0.1.0 and 10.1.0.2.0
*/
import java.sql.Connection;
import javax.sql.PooledConnection;
import oracle.jdbc.pool.OracleConnectionPoolDataSource;
import book.util.InputUtil;
class DemoConnectionPooling
{
  public static void main(String args[]) throws Exception
  {
```

To create a pooled connection, we first create an instance of OracleConnectionPoolDataSource and set its connection properties:

```
    OracleConnectionPoolDataSource ocpds = new OracleConnectionPoolDataSource();
    ocpds.setURL ( "jdbc:oracle:thin:@usunrat24.us.oracle.com:1521:ora92i" );
    ocpds.setUser("scott");           // username
    ocpds.setPassword("tiger");       // password
```

■ **Note** Instead of using the setURL() method and so on, you can use the individual setter methods such as setServerName() to set the same properties.

The next step is to obtain the pooled connection from the OracleConnectionPoolDataSource instance as follows (note the pause we give right after, using the InputUtil.waitTillUser➥HitsEnter() method):

```
    PooledConnection pooledConnection = ocpds.getPooledConnection();
    InputUtil.waitTillUserHitsEnter(
      "Done creating pooled connection.");
```

We then obtain the logical connection from the pooled connection, followed by another pause:

```
    Connection connection = pooledConnection.getConnection();
    InputUtil.waitTillUserHitsEnter(
      "Done getting connection from pooled connection object.");
```

After using the logical connection to execute statements, the application can close it:

```
    connection.close();
    InputUtil.waitTillUserHitsEnter(
      "Done closing logical connection");
```

And finally, to close the pooled connection (thus releasing the actual physical connection), we can use the `close()` method on the pooled connection:

```
    pooledConnection.close();
    InputUtil.waitTillUserHitsEnter("Done closing pooled connection");
  }// end of main
}// end of program
```

Before running the program, I made sure that there was no one connected to my test database. In one session, I connected to my database as `SYS` user. Then I ran the preceding program and ran the query discussed earlier to list connections as `SYS` after each of the pauses we introduced. Table 14-1 lists the steps and the query results.

Table 14-1. *Results of Running a Query That Detects Connections After Each "Pause" in the Program* `DemoConnectionPooling`

Step	Query Results	Notes
After creating a pooled connection	PROGRAM SERVER SERVER_PID USERNAME ---------- ------ ---------- -------- JDBC Thin Client DEDICATED 22326 SCOTT	A physical connection is created.
After obtaining a logical connection	PROGRAM SERVER SERVER_PID USERNAME ---------- ------ ---------- -------- JDBC Thin Client DEDICATED 22326 SCOTT	The physical connection is checked out as a logical connection.
After closing the logical connection	PROGRAM SERVER SERVER_PID USERNAME ---------- ------ ---------- -------- JDBC Thin Client DEDICATED 22326 SCOTT	The physical connection remains.
After closing the pooled connection	No rows selected	The physical connection is closed.

As you can see, the creation of a pooled connection results in an actual physical connection being created. But the retrieval of a logical connection does not result in any new physical connection. Similarly, even after closing the logical connection, the physical connection created is retained for use across other sessions. Finally, when we close the pooled connection, the physical connection is also closed.

In the next section, we'll look at a simple Oracle implementation of a connection cache using the connection pooling framework.

Oracle9*i* Connection Caching

As discussed earlier, a connection cache is a cache of physical connections maintained by the middle tier using the connection pooling framework just discussed. JDBC 2.0 does not specify any API specific to connection cache; it only specifies an API for the underlying connection pooling framework. Thus, the middle tier is free to implement the connection cache in any way. JDBC 3.0 does specify an API along with some connection cache–related properties for implementation of a connection cache at the data source level. As discussed later in the section "Oracle 10*g* Connection Caching," the new Oracle 10*g* caching is compliant with JDBC 3.0 requirements.

■ **Note** The concept of connection caching is not relevant to the server-side internal driver, where you use the default connection.

A connection cache is typically represented by an instance of a connection cache class that caches a group of pooled connection instances (remember, a pooled connection cache is associated with an actual physical database connection). The connection cache class extends the data source API. In Oracle9*i*, for a single connection cache instance, all the associated pooled connections must be physical connections to the same database and schema. In Oracle 10*g*, this restriction has been lifted in the new cache architecture.

■ **Tip** In 10*g*, the restriction that all pooled connections associated with a cache must belong to same schema and database has been removed.

Oracle's Implementation of Connection Cache

Figure 14-4 shows the JDBC standard and Oracle classes related to connection caching.

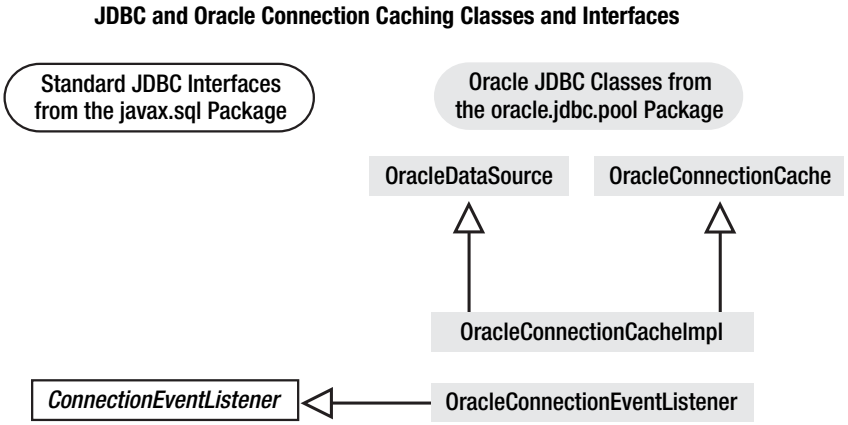

Figure 14-4. *JDBC standard and Oracle-specific connection caching related classes and interfaces*

Oracle provides you with the following classes and interfaces in the `oracle.jdbc.pool` package that implement connection cache functionality for you. In Oracle9*i*, you can use this implementation, or you can code your own using some or all of these classes and interfaces. In Oracle 10*g*, these classes have been deprecated and you should use the implicit connection caching.

- `OracleConnectionCache`: An interface you need to implement if you want to implement your own version of connection caching
- `OracleConnectionCacheImpl`: A class that implements the `OracleConnectionCache` interface
- `OracleConnectionEventListener`: A connection event listener class

As shown in Figure 14-4, the `OracleConnectionCacheImpl` class implements the `OracleConnectionCache` interface and extends `OracleDataSource`. It employs `OracleConnectionEventListener` class instances for connection events pertaining to cache management scenarios, such as when the application closes the logical connection.

Interaction of the Application and Middle Tier When Using Connection Cache

The following steps occur during a typical interaction of a JDBC application and a middle-tier connection cache. The Oracle implementation is used in our examples.

1. The middle tier creates an instance of a connection cache class with its own data source properties that define the physical connections it will cache.
2. The middle tier may optionally bind this instance to a JNDI source.
3. The JDBC application retrieves a connection cache instance (instead of a generic data source) from the middle tier either by using a JNDI lookup or through a vendor-specific API.
4. The JDBC application retrieves the connection from the connection cache through its `getConnection()` method. This results in a logical connection being returned to the JDBC application.
5. 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 the JDBC application is done using the connection, it invokes the `close()` method on the connection. This triggers a connection-closed event and informs the pooled connection instance that its physical connection can be reused.

Steps in Using OracleConnectionCacheImpl

The following sections describe the steps required to instantiate and use the `OracleConnectionCacheImpl` class.

Instantiating OracleConnectionCacheImpl

You instantiate an `OracleConnectionCacheImpl` instance and set its connection properties in one of three ways:

- Use the `OracleConnectionCacheImpl` constructor, which takes an existing connection pool data source as input:

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

- Use the `setConnectionPoolDataSource()` method on an existing `OracleConnectionCacheImpl` instance, which takes a connection pool data source instance as input:

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

- Use the default `OracleConnectionCacheImpl` constructor and set the properties using the setter methods inherited from the `OracleDataSource` class:

```
OracleConnectionCacheImpl occi = new
  OracleConnectionCacheImpl();
occi.setServerName("myserver");
occi.setNetworkProtocol("tcp");
```

Setting Pooled Connection Limit Parameters

The examples in this section assume that `occi` is an initialized `OracleConnectionCacheImpl` variable.

You can set the minimum number of pooled connections by invoking the `setMinLimit()` method as follows:

```
occi.setMinLimit( 3 );
```

The cache will keep three pooled connections open and ready for use at all times.

You can set the maximum number of pooled connections by invoking the `setMaxLimit()` method as follows:

```
occi.setMaxLimit(  10 );
```

The cache will have a maximum of ten pooled connections. What happens when you reach the limit and need another connection? That depends on the cache scheme you set, as discussed next.

Setting Cache Schemes for Creating New Pooled Connections

The `OracleConnectionCacheImpl` class supports three connection cache schemes that come into effect when all three of the following conditions are true:

- The application has requested a connection.
- All existing pooled connections are in use.
- The maximum limit of pooled connections in the cache has been reached.

The three cache schemes are

- *Dynamic*: This is the default scheme. In this scheme, the cache would automatically create new pooled connections, though each of these new connections is automatically closed and freed as soon as the logical connection instance that it provided is closed. You can set this scheme using one of the two overloaded versions of the method `setCacheScheme()`:

```
occi.setCacheScheme( "dynamic" );
occi.setCacheScheme( OracleConnectionCacheImpl.DYNAMIC_SCHEME );
```

- *Fixed return null*: In this scheme, the requests after the maximum limit is exceeded get a null value returned. You can set this scheme using one of the two overloaded versions of the method `setCacheScheme()`:

```
occi.setCacheScheme( "fixed_return_null_scheme" );
occi.setCacheScheme( OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME );
```

- *Fixed wait*: In this case, when the maximum limit of pooled connections is reached, the next request would wait forever. You can set this scheme using one of the two overloaded versions of the method `setCacheScheme()`:

```
occi.setCacheScheme( "fixed_wait_scheme" );
occi.setCacheScheme( OracleConnectionCacheImpl.FIXED_WAIT_SCHEME );
```

Setting Oracle Connection Cache Timeouts

Applications can also time out the physical and logical connections. Oracle JDBC drivers provide following three types of timeout periods for this purpose:

- *Wait timeout*: The maximum period after which a physical connection is returned to the cache. This wait triggers only when all connections are in use and a new connection is requested. A timeout exception, `EOJ_FIXED_WAIT_TIMEOUT`, is thrown when the timeout expires. You use the getter and setter methods on the property `CacheFixedWait➥Timeout` to get and set this timeout.

- *Inactivity timeout*: The maximum period a physical connection can remain unused. When the period expires, the connection is closed and its resources are freed. You use the getter and setter methods on the property `CacheInactivityTimeout` to get and set this timeout.

- *Time to live timeout*: 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. You use the getter and setter methods on the property `CacheTimeToLiveTimeout` to get and set this timeout.

Example of Using OracleConnectionCacheImpl

The following `DemoOracleConnectionCache` class illustrates how to use Oracle connection caching by using the dynamic and "fixed return null" cache schemes. We begin with the imports and the class declaration, followed by the `main()` method:

```
/* This program demonstrates how to use Oracle connection cache.
* COMPATIBLITY NOTE: runs successfully against 9.2.0.1.0 and 10.1.0.2.0
*/
import java.sql.Connection;
import java.sql.SQLException;
import oracle.jdbc.pool.OracleConnectionCacheImpl;
import book.ch03.JDBCUtil;
```

```
class DemoOracleConnectionCache
{
  public static void main(String args[]) throws Exception
  {
```

We instantiate the cache object and set the properties that define the limits and attributes of the cached connections. We also print the default cache scheme.

```
    OracleConnectionCacheImpl occi = new OracleConnectionCacheImpl();
    occi.setURL ( "jdbc:oracle:thin:@rmenon-lap:1522:ora92" );
    occi.setUser("scott");     // username
    occi.setPassword("tiger"); // password
    occi.setMaxLimit( 3 );     // max # of connections in pool
    occi.setMinLimit( 1 );     // min # of connections in pool
    System.out.println( "By default, the cache scheme is: " +
      occi.getCacheScheme() );
```

We then set the cache scheme to "dynamic" and invoke the method `getOneMoreThanMax➥Connections()`. We will see the definition of this method soon, but as the method name suggests, it attempts to get one connection more than the maximum limit of three set previously. This is to see how different cache schemes behave when the limit is exceeded.

```
    occi.setCacheScheme( OracleConnectionCacheImpl.DYNAMIC_SCHEME );
    int maxLimit = occi.getMaxLimit();
    System.out.println( "Max Limit: " + maxLimit );
    System.out.println( "Demo of dynamic cache scheme - the default" );
    _getOneMoreThanMaxConnections( occi, maxLimit );
```

We do the same for the cache scheme "fixed return null":

```
    System.out.println( "\nDemo of fixed return null cache scheme" );
    occi.setCacheScheme( OracleConnectionCacheImpl.FIXED_RETURN_NULL_SCHEME );
    _getOneMoreThanMaxConnections( occi , maxLimit);
  }// end of main
```

The method `getOneMoreThanMaxConnections()` is defined at the end of the program. It simply loops through and tries to create one more than the maximum limit passed to it as a parameter.

```
  private static void _getOneMoreThanMaxConnections(
    OracleConnectionCacheImpl occi , int maxLimit) throws SQLException
  {
    //Create an array of connections 1 more than max limit
    Connection[] connections = new Connection[ maxLimit + 1 ];

    for( int i=0; i < connections.length; i++ )
    {
      System.out.print( "Getting connection no " + (i+1) + " ..." );
```

```
      connections[i] = occi.getConnection();
      if( connections[i] != null )
        System.out.println( " Successful." );
      else
        System.out.println( " Failed." );
    }
    // close all connections
    for( int i=0; i < connections.length; i++ )
    {
      JDBCUtil.close( connections[i] );
    }
  }// end of getOneMoreThanMaxConnections
}// end of program
```

The following is the output of the program when I ran it on my machine:

```
B:\>java DemoOracleConnectionCache
By default, the cache scheme is: 1
Max Limit: 3
Demo of dynamic cache scheme - the default
Getting connection no 1 ... Successful.
Getting connection no 2 ... Successful.
Getting connection no 3 ... Successful.
Getting connection no 4 ... Successful.

Demo of fixed return null cache scheme
Getting connection no 1 ... Successful.
Getting connection no 2 ... Successful.
Getting connection no 3 ... Successful.
Getting connection no 4 ... Failed.
```

As shown, even though we hit the maximum limit, we still got a connection successfully when the cache scheme was dynamic (which is the default). When the cache scheme was "fixed return null," we got a `null` object when we tried to get a fourth connection. The "fixed wait" cache scheme isn't shown, but if you modify the program to use it, the program will wait forever when you try to get the fourth connection.

This concludes our discussion of Oracle9*i* connection pooling and caching. It's time now to look at the implicit connection caching of Oracle 10*g*.

Oracle 10*g* Implicit Connection Caching

As mentioned earlier, starting with Oracle 10*g*, the cache architecture just discussed was deprecated. It has been replaced by a more powerful, JDBC 3.0–compliant implicit connection caching. The highlights of implicit connection caching are

- Transparent access to a connection cache at the data source level.
- Support for connections with different usernames and passwords in the same cache.
- Ability to control cache behavior by defining a set of cache properties. The supported properties include ones that set timeouts, the maximum number of physical connections, and so on.
- Ability to retrieve a connection based on user-defined connection attributes (a feature known as *connection striping*).
- Ability to use callbacks to control cache behavior
  - When a connection is returned to the cache.
  - When a connection has been abandoned.
  - When an application requests a connection that does not exist in the cache.
- The new class `OracleConnectionCacheManager` is provided for administering the connection cache.

With the new cache architecture, you can turn on connection caching simply by invoking `setConnectionCachingEnabled( true )` on an `OracleDataSource` object. After caching is turned on, the first connection request to `OracleDataSource` implicitly creates a connection cache. There is a one-to-one mapping between the `OracleDataSource` object and its implicit connection cache.

Using the Oracle 10*g* Implicit Connection Cache

The following sections discuss the various steps involved in using the implicit connection cache.

Instantiating OracleDataSource

This step should be familiar to you by now:

```
OracleDataSource ods = new OracleDataSource();
ods.setURL ( "jdbc:oracle:thin:@rmenon-pc:1521:ora10g" );
ods.setUser("scott");           // username
ods.setPassword("tiger");       // password
```

Turning the Connection Cache On

You turn on the connection cache by simply invoking `setConnectionCachingEnabled()` on the `OracleDataSource` object:

```
ods.setConnectionCachingEnabled( true );
```

Setting Connection Cache Properties

You can optionally set connection properties listed later in this section by either using the method `setConnectionCacheProperties()` of the `OracleDataSource` object, or using the `OracleConnectionCacheManager` API to create or reinitialize the connection cache as discussed later. For example, the following code sets three properties of the connection cache using the `setConnectionCacheProperties()` method of the `OracleDataSource` object:

```
Properties cacheProperties = new Properties();
cacheProperties.setProperty( "InitialLimit", "2" );
cacheProperties.setProperty( "MinLimit", "3" );
cacheProperties.setProperty( "MaxLimit", "15" );
ods.setConnectionCacheProperties(cacheProperties);
```

By setting connection cache properties, you can control the characteristics of the connection cache.

■ **Caution** In my tests with 10.1.0.2.0, I found that the JDBC driver silently ignores an invalid (or misspelled) property. Thus, you need to be extra careful in spelling these properties while setting them up. Another problem is that, unfortunately, the property names that Oracle chose in many cases are not the same as the ones mentioned in JDBC 3.0 standard, though their meanings may be the same. For example, the JDBC property `InitialPoolSize` means the same thing as the Oracle property `InitialLimit`. This can be confusing.

Let's look at these cache properties in more detail.

Limit Properties

Limit properties control the size of the cache and the number of statements that are cached (see Chapter 13 for details on statement caching), among other things. Table 14-2 lists these properties along with their equivalent JDBC 3.0 standard property (if available), default value, and a brief description.

Table 14-2. *Cache Properties Related to Various Cache Limits Supported by Oracle 10*g *Implicit Connection Cache*

Property	Equivalent JDBC 3.0 Property	Default Value	Description
`InitialLimit`	`initialPoolSize`	0	Determines how many connections are created in the cache when it is created or reinitialized.
`MaxLimit`	`maxPoolSize`	No limit	Sets the maximum number of connections the cache can hold.
`MaxStatementsLimit`	`maxStatements`	0	Sets the maximum number of statements cached by a connection.
`MinLimit`	`minPoolSize`	0	Sets the minimum number of connections the cache is guaranteed to have at all times.
`LowerThresholdLimit`		20% of `maxLimit`	Sets the lower threshold limit on the cache. It is used when a `releaseConnection()` callback is registered with a cached connection. When the number of connections in the cache reaches this limit (`LowerThresholdLimit`), and a request is pending, the cache manager calls this method on the cache connections (instead of waiting for the connection to be freed).

Timeout and Time Interval Properties

These properties determine when the connections in the cache time out or at what interval Oracle checks and enforces the specified cache properties. Table 14-3 lists each of these properties with its default value and the type of connection it impacts (physical or logical). Only the property `PropertyCheckInterval` has a JDBC 3.0–equivalent property (`propertyCycle`).

Table 14-3. *Timeout and Interval Properties of Implicit Connection Cache*

Property	Default Value	Type of Connection Impacted	Description
`InactivityTimeout`	0 (no timeout)	Physical	Sets the maximum time in seconds a physical connection can remain idle in a connection cache. An *idle* connection is one that is not active and does not have a logical handle associated with it.
`TimeToLiveTimeout`	0 (no timeout)	Logical	Sets the maximum time in seconds that a logical connection can remain open (or checked out), after which it is returned to the cache.
`AbandonedConnectionTimeout`	0 (no timeout)	Logical	Sets the maximum time in seconds that a logical connection can remain open (or checked out) without any SQL activity on that connection, after which the logical connection is returned to the cache.
`ConnectionWaitTimeout`	0 (no timeout)	Logical	Comes into play when there is a request for a logical connection, the cache has reached the `MaxLimit`, and all physical connections are in use. This is the number of seconds the cache will wait for one of the physical connections currently in use to become free so that the request can be satisfied. After this timeout expires, the cache returns `null`.
`PropertyCheckInterval`	900 seconds		Sets the time interval in seconds at which the cache manager inspects and enforces all specified cache properties.

Attribute Weight Properties

Attribute weight properties allow you to set weights on certain attributes of a connection in the connection cache. If the property `ClosestConnectionMatch` is set to `true`, then these weights are used to get a "closest" match to the connection you request. We will look at these properties along with this feature in more detail in the section "Using Connection Attributes and Attribute Weights (10*g* Only)."

The ValidateConnection Property

If you set this property to `true`, the cache manager tests for validity each connection to be retrieved from the database. The default value is `false`.

Closing a Connection

Once an application is done using a connection, the application closes it using the `close()` method on the connection. There is another variant of this method that we will discuss in the section on "Using Connection Attributes and Attribute Weights (10*g* Only)."

An Example of Using Implicit Connection Caching

The following `DemoImplicitConnectionCaching` class illustrates how implicit connection caching works. First, we declare the class and the `main()` method after importing the required classes:

```
/* This program demonstrates implicit connection caching.
* COMPATIBLITY NOTE: runs successfully 10.1.0.2.0
*/
import java.sql.Connection;
import java.util.Properties;
import oracle.jdbc.pool.OracleDataSource;
import book.util.InputUtil;
class DemoImplicitConnectionCaching
{
  public static void main(String args[]) throws Exception
  {
```

Next, in the `main()` method, we instantiate the `OracleDataSource` object, which will hold our implicit cache:

```
    OracleDataSource ods = new OracleDataSource();
    ods.setURL ( "jdbc:oracle:thin:@rmenon-lap:1521:ora10g" );
    ods.setUser("scott");           // username
    ods.setPassword("tiger");       // password
```

We then enable the implicit caching:

```
    // enable implicit caching
    ods.setConnectionCachingEnabled( true );
```

We set the cache properties with an initial and minimum limit of three connections and a maximum limit of fifteen connections. The cache, when first set up, should pre-establish three connections, and it should never shrink below three connections later. Note that in production code, you should use a properties file to set these properties instead of using the `setProperty()` method in your Java code. This makes it easier to change them during runtime.

```
// set cache properties
Properties cacheProperties = new Properties();
cacheProperties.setProperty( "InitialLimit", "3" );
cacheProperties.setProperty( "MinLimit", "3" );
cacheProperties.setProperty( "MaxLimit", "15" );
ods.setConnectionCacheProperties(cacheProperties);
```

We first establish two connections to the `SCOTT` user, followed by one connection to the `BENCHMARK` user. We calculate and print the time it took to establish each of these connections. We also interject pauses using the `InputUtil.waitTillUserHitsEnter()` method (explained earlier in this chapter) after each connection establishment step. During these pauses, we will examine the database to see the number of physical connections using the query we covered in the section "Connections and Sessions in Oracle."

```
// time the process of establishing first connection
long startTime = System.currentTimeMillis();
Connection conn1 = ods.getConnection("scott", "tiger");
long endTime = System.currentTimeMillis();
System.out.println("It took " + (endTime-startTime) +
  " ms to establish the 1st connection (scott)." );
InputUtil.waitTillUserHitsEnter();
// time the process of establishing second connection
startTime = System.currentTimeMillis();
Connection conn2 = ods.getConnection("scott", "tiger");
endTime = System.currentTimeMillis();
System.out.println("It took " + (endTime-startTime) +
  " ms to establish the 2nd connection (scott)." );
InputUtil.waitTillUserHitsEnter();
// time the process of establishing third connection
startTime = System.currentTimeMillis();
Connection conn3 = ods.getConnection("benchmark", "benchmark");
endTime = System.currentTimeMillis();
System.out.println("It took " + (endTime-startTime) +
  " ms to establish the 3rd connection (benchmark)." );
InputUtil.waitTillUserHitsEnter();
```

At the end of the program, we close all connections, putting a pause after the first `close()` statement:

```
// close all connections
conn1.close();
InputUtil.waitTillUserHitsEnter("After closing the first connection.");
conn2.close();
```

```
    conn3.close();
  }// end of main
}// end of program
```

I ran this program on my machine, and after every pause, I ran the query to detect connections. Let's look at the output and the query results side by side:

```
B:\> java DemoImplicitConnectionCaching
It took 1015 ms to establish the 1st connection (scott).
Press Enter to continue...
```

The query results executed as the SYS user right after the pause are as follows:

```
sys@ORA10G> select s.server, p.spid server_pid, s.username
  2  from v$session s, v$process p
  3  where s.type = 'USER'
  4    and s.username != 'SYS'
  5    and p.addr(+) = s.paddr;
SERVER          SERVER_PID   USERNAME
--------------- ------------ ----------
DEDICATED       2460         SCOTT
DEDICATED       3528         SCOTT
DEDICATED       3288         SCOTT
```

This tells us that the very first time we establish a connection, the implicit connection caching results in the creation of three connections (equal to the parameter InitialLimit that we set earlier). That also explains why it took 1,015 milliseconds to establish the "first" connection. The program output after I pressed Enter follows:

```
It took 0 ms to establish the 2nd connection (scott).
Press Enter to continue...
```

This shows the main benefit of connection caching. Since we already have three connections established, this call gets one of these from the cache and completes really fast. At this time, if we run the query again, we will see the same output as before, since no new connections have been established. Pressing Enter again results in the following output:

```
It took 266 ms to establish the 3rd connection (benchmark).
Press Enter to continue...
```

Even though we have one more physical connection left in the cache, we cannot use it, since this time the request is to establish a connection to the user BENCHMARK. Hence the connection cache has to create a new connection, which took 266 milliseconds. At this time, the cache has four physical connections (three to SCOTT and one to BENCHMARK) established, as shown by our query results:

```
sys@ORA10G> select s.server, p.spid server_pid, s.username
  2  from v$session s, v$process p
  3  where s.type = 'USER'
  4    and s.username != 'SYS'
  5    and p.addr(+) = s.paddr;
```

```
SERVER          SERVER_PID   USERNAME
--------------- ------------ ----------
DEDICATED       2460         SCOTT
DEDICATED       3528         SCOTT
DEDICATED       3288         SCOTT
DEDICATED       2132         BENCHMARK
```

Finally, we press Enter again and see

```
After closing the first connection.
Press Enter to continue...
```

After the first connection is closed, when we execute the preceding query, we get the same results as before (four connections). This is, of course, because closing the logical connection does not result in a closing of the physical connection.

To manage your implicit caches, Oracle provides you with an API in the form of the class `OracleConnectionCacheManager` we'll look at it in the next section.

The OracleConnectionCacheManager Class

`OracleConnectionCacheManager` provides methods for the middle tier to centrally manage one or more connection caches that share a JVM. Each cache is given a unique name (implicitly or explicitly). The `OracleConnectionCacheManager` class also provides information about the cache, such as number of physical connections that are in use and the number of available connections.

The following sections describe some of the more commonly used methods that this class provides, with short descriptions. For a complete list of supported methods, please refer to *Oracle Database JDBC Developer's Guide and Reference* (for 10*g*).

createCache()

Using `createCache()`, you can create a connection cache with a given `DataSource` object and a `Properties` object. It also allows you to give a meaningful name to the cache, which is useful when you are managing multiple caches in the middle tier. The second variant listed generates a name for the cache internally.

```
public void createCache(String cacheName, javax.sql.DataSource datasource,
  java.util.Properties cacheProperties );
public void createCache(javax.sql.DataSource datasource,
   java.util.Properties cacheProperties );
```

removeCache()

This method waits `timeout` number of seconds for the in-use logical connections to be closed before removing the cache.

```
public void removeCache(String cacheName, int timeout);
```

reinitializeCache()

This method allows you to reinitialize the cache with the new set of properties. This is useful in dynamically configuring the cache based on runtime load changes and so forth.

```
public void reinitializeCache(String cacheName, java.util.properties
  cacheProperties)
```

■ **Caution** Invoking `reinitializeCache()` will close all in-use connections.

---

enableCache() and disableCache()

These two methods enable or disable a given cache. When the cache is disabled, in-use connections will work as usual, but no new connections will be served out from the cache.

```
public void enableCache(String cacheName);
public void disableCache(String cacheName);
```

getCacheProperties()

This method gets the cache properties for the specified cache.

```
public java.util.Properties getCacheProperties(String cacheName)
```

getNumberOfAvailableConnections()

This method gets the number of connections in the connection cache that are available for use.

```
public int getNumberOfAvailableConnections(String cacheName)
```

getNumberOfActiveConnections()

This method gets the number of in-use connections at a given point of time for a given cache.

```
public int getNumberOfActiveConnections(String cacheName)
```

setConnectionPoolDataSource()

This method sets the connection pool data source for the cache. All properties are derived from this data source.

```
public void setConnectionPoolDataSource(String cacheName,
  ConnectionPoolDataSource cpds)
```