Managing database connections with JDBC

Presented by developerWorks, your source for great tutorials

`ibm.com/developerWorks`

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial introduces the different concepts involved in establishing and managing a database connection from within a Java application using Java Database Connection (JDBC). It is targeted primarily at developers who want to understand what is "going on under the hood" when using a database from within a Java application.

This tutorial assumes familiarity with the Java programming language. The links in Resources on page 27 include referrals to additional information on both JDBC and specific databases.

What is this tutorial about?

This tutorial demonstrates how to connect to a database using JDBC. While seemingly innocuous, this subject is actually a stumbling block for both newcomers and veterans alike. This tutorial will discuss how a Java application inside a JVM discovers and communicates with a database, starting with the traditional JDBC driver and **`DriverManager`** objects. After several examples that demonstrate the four different types of JDBC drivers, the tutorial moves on to discuss **`DataSource`** objects that use JNDI. A discussion of JNDI, and how to bind, use, rebind, and delete the **`DataSource`** object is also included. Finally, the concept of a connection pool, and specifically **`PooledConnection`** objects are introduced and demonstrated. The tutorial concludes with a discussion of tuning issues that are often overlooked when developing database connectivity applications.

Tools

While the tutorial provides numerous code snippets to reflect concepts and methods described in the text, most people learn better by actually working through the examples. To work through the examples, you will need to have the following tools installed and working correctly:

* A text editor: Java source files are simply text, so to create and read them, all you need is a text editor. If you have access to a Java IDE, you can also use it, but sometimes they hide too many of the details.

* A Java development environment, such as the Java2 SDK, which is available at *http://java.sun.com/j2se/1.4/*. The Java2 SDK, Standard Edition version 1.4, includes the JDBC standard extensions as well as JNDI, which are both necessary for some of the latter examples in the book.

* An SQL-compliant database: The examples in this tutorial use a wide variety of different databases to help demonstrate how database independent JDBC programming can be. Resources on page 27 contains links to more information on both JDBC and databases.

* A JDBC driver: Because the JDBC API is predominantly composed of interfaces, you need to obtain an actual JDBC driver implementation in order to actually connect to a database using JDBC. If your database (or your wallet) does not allow the use of JDBC, you can always use the JDBC-ODBC bridge driver to connect to any database (or data source) that supports the ODBC protocol.

About the author

Robert J. Brunner is an astrophysicist by day and a computer hacker by night. His principal employer is the California Institute of Technology, where he works on knowledge extraction and data-mining from large, highly distributed astronomical databases. He has provided consulting to scientific data centers around the world, provided Java and XML training to a variety of groups, and is currently working on the book *Enterprise Java Database Programming* which will be published by Addison Wesley in 2002. Feel free to contact him via e-mail at *rjbrunner@pacbell.net*.

Section 2. Application architecture

Architecting your system

One of the most important design issues when developing a Java database application is the overall system architecture; in particular, how many different components should be deployed. Traditionally, this is characterized by how many tiers, or layers, the application requires. There are two basic architectural models that can describe a system: the two-tier model and the n-tier model.

Before jumping into the details of managing database connections from a Java application, we need to discuss these two models. Each model has its own advantages and disadvantages; each also requires certain components to be set up appropriately, and, as a result, they each work best in different environments. The next two panels discuss each of the two architectural models in more detail.

The two-tier model

The two-tier model is the traditional client-server framework; it has a client tier and a server tier. This simple model requires that the client be intimately aware of the database server. Thus, for example, the client needs database-specific code resulting in a tight coupling between the two tiers. This tight coupling has several advantages. First, it can decrease development time due to the fact the overall system is considerably simpler and smaller. Second, the tight coupling can potentially improve system performance as the client can easily take advantage of specific server functionality that might not be available to a less tightly coupled system.

On the other hand, this tight coupling can lead to several problems. Most notably, system maintenance can become more difficult because changes in the server can break the client or visa versa. Furthermore, if the database changes, all of the client code will need to be modified. If the client is highly distributed, propagating changes throughout the system can be difficult, and in some scenarios impossible. As a result, two-tier applications can be useful in a corporate LAN environment where complete control of all clients is achieved, or at the initial, rapid prototyping stage of a project where different options are being evaluated.

The n-tier model

The *n*-tier model has a client tier, at least one server tier, and at least one middle layer. Because of the extra tier, many of the problems that affected the two-tier model are no longer an issue. For example, the middle layer now maintains the database connection information. This means the clients only need to know about the middle tier. Because the middle tier is generally operating at the same physical location as the server (for instance, both components can be behind the same firewall), maintaining the middle tier is considerably easier than maintaining several hundred client installations.

Another advantage of the *n*-tier approach is that the overall system can easily scale to handle more users. All one needs to do is add more middle tiers or server tiers, depending on the results of the profiling operations. Because middle tiers are typically implemented using Web servers -- using JavaServer Pages and Servlets technologies -- it is simple to add load-balancing or even new hardware components.

All is not completely rosy, however, as the extra tier introduces additional complexity into the overall system. This means more code, harder unit testing, and potentially difficult and nasty bugs. Fortunately, the Java language provides many of the necessary components, pre-built, for constructing viable n-tier applications. Furthermore, this model lends itself to easily support authentication and internationalization, as the middle layer controls the flow of information and provides a natural location for handling security and localization concerns.

Section 3. JDBC driver fundamentals

An overview of JDBC drivers

A casual inspection of the JDBC API quickly shows the dominance of interfaces within the API, which might lead a user to wonder where the work is done. Actually this is a strength of the approach that the JDBC developers took because the actual implementation is provided by JDBC Driver vendors, who in turn provide the classes that implement the necessary interfaces. This approach introduces competition that provides the consumer with more choices, and for the most part, produces better software. With all of the available drivers, choosing one can be difficult. Fortunately, Sun Microsystems maintains a *searchable database* of over 150 JDBC drivers from a wide array of vendors. This should be your first stop after selecting a database.

From a programming perspective, there are two main classes responsible for establishing a connection with a database. The first class is **`DriverManager`**, which is one of the few actual classes provided in the JDBC API. **`DriverManager`** is responsible for managing a pool of registered drivers, essentially abstracting the details of using a driver so the programmer does not have to deal with them directly. The second class is the actual JDBC **`Driver`** class. These are provided by independent vendors. The JDBC Driver class is responsible for establishing the database connection and handling all of the communication with the database. JDBC drivers come in four different types; the rest of this section discusses each one in detail.

Registering a JDBC driver

The first step in the process of creating a connection between a Java application and a database is the registration of a JDBC driver with the Java virtual machine (JVM) in which the Java application is running. In the traditional connection mechanism (as opposed to the **`DataSource`** connection mechanism, discussed later in Data sources on page 18), the connection and all database communications are controlled by the **`DriverManager`** object. To establish a connection, a suitable JDBC driver for the target database must be registered with the **`DriverManager`** object.

The JDBC specification (for more detail, see the *JDBC API Tutorial and Reference* in Resources on page 27), states that JDBC drivers are supposed to register themselves with the **`DriverManager`** object automatically when they are loaded into a JVM. For example, the following code snippet uses a static initializer to first create an instance of the **`persistentjava`** JDBC driver and then register it with the **`DriverManager`**.

```
static {
  java.sql.DriverManager.registerDriver(new com.persistentjava.JdbcDriver()) ;
}
```

Registering a driver is simply a matter of loading the driver class into the JVM, which can be done in several different ways. One way to do this is with the ClassLoader **`Class.forName(com.persistentjava.JdbcDriver) ;`**. Another method, which is not as well known, uses the **`jdbc.drivers`** system property. This method can be used in one of three different ways:

* From the command line:

```
java -Djdb.drivers=com.persistentjava.JdbcDriver Connect
```

* Within the java application:

```
System.setProperty("jdbc.drivers",
"com.persistentjava.JdbcDriver") ;
```

* By setting the **`jdbc.drivers`** property in the System property file, which is generally system dependent

By separating the drivers with a colon, multiple drivers can be registered using the aforementioned system property technique. One of the benefits of using the system property technique is that drivers can be easily swapped in and out of the JVM without modifying any code (or at least with minimal code changes). If multiple drivers are registered, their order of precedence is: 1) JDBC drivers registered by the **`jdbc.drivers`** property at JVM initialization, and 2) JDBC drivers dynamically loaded. Because the **`jdbc.drivers`** property is only checked once upon the first invocation of a **`DriverManager()`** method, it's important to ensure all drivers are registered correctly before establishing the database connection.

Not all JVMs are created equal, however, and some JVMs do not follow the JVM specification. As a result, static initializers do not always work as advertised. This results in multiple ways to register a JDBC driver, including:

* **`Class.forName("com.persistentjava.JdbcDriver").newInstance() ;`**
* **`DriverManager.registerDriver(new com.persistentjava.JdbcDriver()) ;`**

These alternatives should work fine in all JVMs, so you should feel comfortable using them to work across the widest possible array of JVMs. One final issue is that **`Class.forname()`** can throw a **`ClassNotFoundException`**, so you need to wrap the registration code in an appropriate exception handler.

JDBC driver URLs

Once a JDBC driver has been registered with the **`DriverManager`**, it can be used to establish a connection to a database. But how does the **`DriverManager`** select the right driver, given that any number of different drivers might actually be registered? (Remember, a single JVM might be supporting multiple concurrent applications, which might be connecting to different databases with different drivers.) The technique is quite simple: each JDBC driver uses a specific JDBC URL (which has the same format as Web addresses) as a means of self-identification. The format of the URL is straightforward and probably looks familiar: **`jdbc:`*`sub-protocol`*`:`*`database locator`***. The *sub-protocol* is specific to the JDBC driver and can be *odbc*, *oracle*, *db2*, and so on depending on the actual JDBC driver vendor. The *database locator* is a driver-specific indicator for uniquely specifying the database with which an application wants to interact. Depending on the type of driver, this locater may include a hostname, a port, and a database system name.

When presented with a specific URL, the **`DriverManager`** iterates through the collection of registered drivers until one of the drivers recognizes the specified URL. If no suitable driver is found, an **`SQLException`** is thrown. The following list demonstrates several specific examples of actual JDBC URLs:

```
* jdbc:odbc:jdbc
* jdbc:oracle:thin:@persistentjava.com:1521:jdbc";
* jdbc:db2:jdbc
```

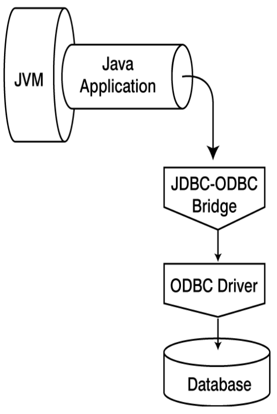
Many drivers, including the JDBC-ODBC bridge driver, accept additional parameters at the end of the URL such as username and password.

The method for obtaining a database connection, given a specific JDBC URL, is to call **`getConnection()`** on the **`DriverManager`** object. This method comes in several flavors:

```
* DriverManager.getConnection(url) ;
* DriverManager.getConnection(url, username, password) ;
* DriverManager.getConnection(url, dbproperties) ;
```

Here, **`url`** is a **`String`** object that is the JDBC URL; username and password are **`String`** objects that are the username and password that the JDBC application should use to connect to the data source; and **`dbproperties`** is a Java **`properties`** object that encapsulates all of the parameters (possibly including username and password) that a JDBC driver requires to successfully make a connection.

Now that we have the driver basics in hand, we can examine the individual driver types in more detail.



Type one drivers

Type one drivers come in one variety: they all use the JDBC-ODBC bridge, which is included as a standard part of the JDK. Type one drivers are differentiated by the ODBC (Open DataBase Connectivity) driver attached to the JDBC-ODBC bridge. To connect to a different data source, you simply have to register (or effectively bind) a different ODBC data source, using the ODBC

Administrator, to the appropriate data source name.

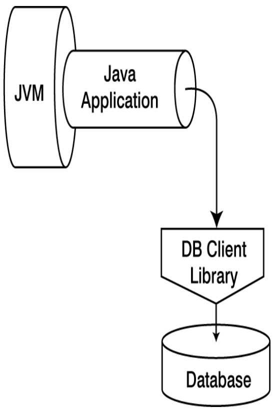
Because ODBC has been around for quite a while (longer than the Java language), ODBC drivers are rather ubiquitous. This makes this type of JDBC driver a good choice for learning how to connect Java programs to databases. In fact, there are even ODBC drivers that let you assign an ODBC data source to Microsoft Excel applications or plain-text files. The extra level of indirection, however, can result in a performance penalty as the JDBC is transferred into ODBC, which is then transferred into the database-specific protocol. Another potential problem for type one drivers is their use in distributed applications. Because the bridge itself does not support distributed communication, the only way type one drivers can work across a network is if the ODBC driver itself supports remote interactions. For simple ODBC drivers, this is not an option, and while big databases do typically have ODBC drivers that can work remotely, they cannot compete performance-wise with the pure Java JDBC drivers.

Coding for type one drivers

The class name for the JDBC-ODBC bridge driver is **`sun.jdbc.odbc.JdbcOdbcDriver`** and the JDBC URL takes the form **`jdbc:odbc:dsn`**, where **`dsn`** is the Data Source Name used to register the database with the ODBC Administrator. For example, if a database is registered with an ODBC data source name of *jdbc*, a username of *java*, and a password of *sun*, the following code snippet can be used to establish a connection.

Note: In the interests of clarity and brevity, proper error handling and checking has been left out of this code snippet. Later examples demonstrate this important mechanism (specifically, trapping errors by chaining SQLException statements).

```
String url = "jdbc:odbc:jdbc" ;
Connection con ;
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver") ;
} catch(java.lang.ClassNotFoundException e) {
  System.err.print("ClassNotFoundException: ") ;
  System.err.println(e.getMessage()) ;
  return ;
}
try {
  con = DriverManager.getConnection(url, "java", "sun");
} catch(SQLException ex) {
  System.err.println("SQLException: " + ex.getMessage());
} finally {
  try{
        con.close ;
  } catch(SQLException ex) {
    System.err.println(SQLException: " + ex.getMessage()) ;
  }
}
```



Type two drivers

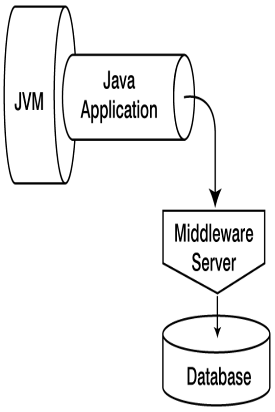
Type two drivers are also known as partial Java drivers, in that they translate the JDBC API directly into a database-specific API. The database client application (for the purposes of this tutorial, the host that is running the JVM) must have the appropriate database client library, which might include binary code installed and possibly running. For a distributed application, this requirement can introduce extra licensing issues, as well as potential nightmare code distribution issues. For example, using a type two model restricts the developer to client platforms and operating systems supported by the database vendor's client library.

This model can work effectively, however, when the client base is tightly controlled. This typically occurs in corporate LANs. One example of a type two driver is the DB2 JDBC application driver. The following example demonstrates how to establish a connection using a DB2 driver.

```
String url = "jdbc:db2:jdbc" ;
try {
  Class.forName("COM.ibm.db2.jdbc.app.DB2Driver") ;
} catch(java.lang.ClassNotFoundException e) {
  System.err.print("ClassNotFoundException: ") ;
  System.err.println(e.getMessage()) ;
  return ;
}
...
```

Note how similar the above code snippet is to the type one example. This is the primary selling feature of type two model: the learning curve for a programmer moving from one model to the other is slight to nonexistent.

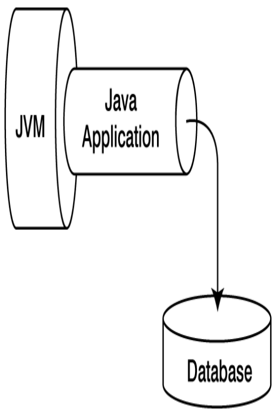
The last two driver types are pure Java drivers. The benefit of pure Java drivers are their ease of deployment in highly distributed environments.



Type three drivers

Type three drivers are pure Java drivers that transform the JDBC API into a database-independent protocol. The JDBC driver does not communicate with the database directly; it communicates with a middleware server, which in turn communicates with the database. This extra level of indirection provides flexibility in that different databases can be accessed from the same code because the middleware server hides the specifics from the Java application. To switch to a different database, you only need to change parameters in the middleware server. (One point of note: the database format you are accessing must be supported by the middleware server.)

The downside to type three drivers is that the extra level of indirection can hurt overall system performance. On the other hand, if an application needs to interact with a variety of database formats, a type three driver is an efficient approach due to the fact that the same JDBC driver is used regardless of the underlying database. In addition, because the middleware server can be installed on a specific hardware platform, certain optimizations can be performed to capitalize on profiling results.



Type four drivers

Type four drivers are pure Java drivers that communicate directly with the database. Many programmers consider this the best type of driver, as it typically provides optimal performance and allows the developer to leverage database-specific functionality. Of course this tight coupling can hinder flexibility, especially if you need to change the underlying database in an application. This type of driver is often used in applets and other highly distributed applications. The following code snippet shows how to use a DB2 type four driver.

```
String url = "jdbc:db2://persistentjava.com:50000/jdbc" ;
try {
  Class.forName("COM.ibm.db2.jdbc.net.DB2Driver") ;
} catch(java.lang.ClassNotFoundException e) {
  System.err.print("ClassNotFoundException: ") ;
  System.err.println(e.getMessage()) ;
  return ;
}
...
```

A complete type four driver example

The following example demonstrates how to use a JDBC driver from a third-party vendor, in this case Merant, to connect to a DB2 database. DB2 UDB requires additional, non-standard information to establish the database connection, which in this example is appended to the JDBC URL as optional parameters.

```
package com.persistentjava;
import java.sql.*;
public class ConnectMerantDB2 {
    static {
        try {
            Class.forName("com.merant.datadirect.jdbc.db2.DB2Driver").newInstance();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
public static void main(String args[]) {
    String url = "jdbc:merant:db2://persistentjava.com:50000;" ;
    url += "DatabaseName=jdbc;CollectionId=DEFAULT;" ;
    url += "PackageName=JDBCPKG;CreateDefaultPackage=TRUE";
    Connection con;
    System.out.println("Connecting");
    try {
        con = DriverManager.getConnection(url, "java", "sun");
        System.out.println("Connection Established");
        con.close();
    // In this example, the proper handling of SQLExceptions is demonstrated
    // as they can be potentially chained.
    } catch (SQLException e) {
        System.out.println("\nERROR:----- SQLException -----\n");
        while (e != null) {
            System.out.println("Message:   " + e.getMessage());
            System.out.println("SQLState:  " + e.getSQLState());
            System.out.println("ErrorCode: " + e.getErrorCode());
            e.printStackTrace();
            e = e.getNextException();
        }
    }
}
}
```

Section 4. Database transactions

Transaction basics

One concept that causes problems for newcomers to the world of database application development is the idea of transactions. Fundamentally, a transaction represents a logical unit of work. Because the primary responsibility of a database is to preserve information, it needs to have some means for a user to indicate that the current program state should be saved. Likewise, when things have gone awry, there needs to be a way to indicate that a database should ignore the current state and go back to the previously saved program state.

In database parlance, these functions are called committing a transaction and rolling a transaction back, respectively. To accomplish these tasks, the JDBC API includes two methods as part of the **`Connection`** interface. Given a **`Connection`** object name **`con`**, the program state is saved by calling **`con.commit() ;`** to return to the previously saved state, **`con.rollback() ;`**. Both of these methods can throw **`SQLExceptions`** if something goes wrong when the database actually performs the operation, so you need to wrap them in **`try ... catch`** blocks.

More on transactions

In single-user mode, transactions are rather simple to understand -- they simply involve the saving of, or forgetting of, an application's state. In multi-user mode, however, transactions become much more complex. The classic demonstration of a multi-user transaction is the bank account where one application is trying to debit an account, while another application is trying to credit the same account. If you are familiar with concurrent programming (also known as multithreaded programming), you probably have seen this problem before. The fundamental issue is that unless the two transactions are isolated from each other, one application might interrupt the other one resulting in an incorrect program state. In our simple demo, that might mean an account has the wrong amount in it, something that is not exactly conducive to retaining customers.

Three common problems can arise when dealing with multiple users accessing the same data:

* **Dirty reads.** A dirty read occurs when an application uses data that has been modified by another application, and that data is in an uncommitted state. The second application then requests that the data it was modifying be rolled back. The data used by the first transaction is then corrupted, or "dirty".

* **Non-repeatable reads.** A non-repeatable read occurs when one transaction obtains data, which is subsequently altered by a separate transaction, and the first transaction re-reads the now altered data. Thus, the first transaction did a non-repeatable read.

* **Phantom reads.** A phantom read occurs when one transaction acquires data via some query, another transaction modifies some of the data, and the original

transaction retrieves the data a second time. The first transaction will now have a different result set, which may contain phantom data.

Transaction levels

To solve the issues associated with multiple threads requesting the same data, transactions are isolated from each other by locks. Most major databases support different types of locks; therefore, the JDBC API supports different types of transactions, which are assigned or determined by the **`Connection`** object. The following transaction levels are available in the JDBC API:

* **`TRANSACTION_NONE`** indicates that transactions are not supported.

* **`TRANSACTION_READ_UNCOMMITTED`** indicates that one transaction can see another transaction's changes before they are committed. Thus dirty reads, nonrepeatable reads, and phantom reads are all allowed.

* **`TRANSACTION_READ_COMMITTED`** indicates that reading uncommitted data is not allowed. This level still permits both nonrepeatable and phantom reads to occur.

* **`TRANSACTION_REPEATABLE_READ`** indicates that a transaction is guaranteed to be able to re-read the same data without fail, but phantom reads can still occur.

* **`TRANSACTION_SERIALIZABLE`** is the highest transaction level and prevents dirty reads, nonrepeatable reads, and phantom reads from occurring.

You might wonder why all transactions don't operate in **`TRANSACTION_SERIALIZABLE`** mode in order to guarantee the highest degree of data integrity. The problem is, similar to the issues involved with handling multiple programming threads, the higher the level of transaction protection, the higher the performance penalty.

Given a **`Connection`** object, you can explicitly set the desired transaction level, assuming your database and JDBC driver support this feature:

```
con.setTransactionLevel(TRANSACTION_SERIALIZABLE) ;
```

You also can determine the current transaction level:

```
 if(con.getTransactionLevel() == TRANSACTION_SERIALIZABLE)
   System.out.println("Highest Transaction Level in operation.") ;
```

Batches and transaction

By default, JDBC drivers operate in what is called *autocommit* mode. In this mode, all commands sent to the database operate in their own transaction. While this can be useful to newcomers, it involves a performance penalty because transactions require a certain amount of overhead to properly set everything up. If you want to be able to explicitly control commits and rollbacks, you need to disable the autocommit mode:

```
con.setAutoCommit(false) ;
```

You can also quickly determine the autocommit mode for a given **Connection** object:

```
if(con.getAutoCommit() == true)
  System.out.println("Auto Commit mode");
```

Many databases support batch operations, in which the transaction overhead is minimized by performing multiple database update operations in a single operation, or *batch*. Batch operations were introduced in JDBC 2.0, and require that a transaction is not in autocommit mode. A batch operation is demonstrated in the following example, which assumes that a **Connection** exists to a database that contains a simple table.

```
con.setAutoCommit(false) ;
Statement stmt = connection.createStatement() ;
stmt.addBatch("INSERT INTO people VALUES('Joe Jackson', 0.325, 25, 105)  ;
stmt.addBatch("INSERT INTO people VALUES('Jim Jackson', 0.349, 18, 99)  ;
stmt.addBatch("INSERT INTO people VALUES('Jack Jackson', 0.295, 15, 84)  ;
int[] updateCounts = stmt.executeBatch() ;
con.commit() ;
```

Notice that the **executeBatch()** method returns an array of update counts, one for each command in the batch operation. One last issue with batch operations is that they can throw a new exception of type **BatchUpdateException**, which indicates at least one of the commands in the batch operation failed. Thus, you need to add an appropriate exception handler to your batch operations.

Fine-grained transaction control

Beginning with the JDBC 3.0 API, a new interface element was added relating to transactions. This interface introduces the concept of *savepoints*. Savepoints provide specific markers within a database application that can be used as arguments when calling the rollback method. As a result, using the JDBC 3.0 API, it is now possible to set a savepoint before starting a complicated database interaction and, depending on the outcome, commit the entire transaction or rollback to the savepoint and return the application to a known state.

To set a savepoint, create a **Savepoint** object from the **Connection** object, as shown here:

```
Savepoint svpt = con.setSavepoint("Savepoint") ;
```

To rollback to a given **Savepoint**, simply pass the desired **Savepoint** object as a parameter to the rollback method:

```
con.rollback(svpt) ;
```

When they are no longer needed, release all **`Savepoint`** objects to free up potentially expensive database resources for other users:

```
con.releaseSavepoint(svpt) ;
```

Note that when you commit or rollback a transaction, any created **`Savepoints`** may become invalid depending on the exact order and type of operation. See the JDBC 3.0 API specification or your driver manual for more information.

Section 5. Data sources

Data source basics

One of the major benefits of using the JDBC API is to facilitate database-independent programming, as the majority of JDBC applications can be easily transferred over to a different database. Two main items still remain tied to a particular database, however, namely the JDBC **`Driver`** class and the JDBC URL. With the introduction of data sources in JDBC API 2.0, even these dependencies can be eliminated.

Essentially a **`DataSource`** object represents a particular source of data in a Java application. Besides encapsulating the database and JDBC driver-specific information into a single, standardized object, data sources can act as a **`Connection`** factory and provide methods for setting and getting the particular properties that the **`DataSource`** object requires for successful operation. Some of the standard properties a **`DataSource`** object might require include:

* **`databaseName`**
* **`serverName`**
* **`portNumber`**
* **`userName`**
* **`password`**

One additional benefit of using a **`DataSource`**, which you might infer from the properties above, is that sensitive security-related information like username, password, and even database server are only coded in one place, which can be done by a systems administrator. While the interaction with a **`DataSource`** object can be done with a graphical application, it is instructive to actually see working examples.

While the concepts of a **`DataSource`** object are rather simple, to be used within a Java application a **`DataSource`** object is referenced using the Java Naming and Directory Interface, or JNDI. Before jumping into **`DataSource`** example code, the next panel introduces the relevant concepts of JNDI that are needed to properly use a **`DataSource`** object.

A quick primer on JNDI

JNDI is a Java API that encapsulates the concept of naming and directory servers in much the same manner that JDBC encapsulates the concepts behind communicating with a database. While this might seem confusing, it is actually quite straightforward -- all computer users use naming and directory services every day. For example, hard drives work by dealing with tracks and sectors, yet a user only worries about filenames and directories. The file system manages the naming service which associates a given filename with a specific location on the hard drive. Another simple example is the Web, where most users worry only about the name of the Web site, like www.persistentjava.com, and not the underlying IP address. However, TCP/IP communication is done using the IP address and not some human readable name. The transformation between the two representations is performed by DNS, or Domain Name System.

While the JNDI provides a rich and useful API in its own right, our needs are considerably simpler. In short, we need to know how to do four things:

* Create a name and bind it to a Java object
* Look up a name to retrieve a Java object
* Delete a name
* Rebind a name to a new Java object

Rather than provide contrived JNDI examples of the above tasks, the following four panels show examples demonstrating these tasks using JDBC data sources. All of these examples use the file system provider, which is a separate download.

Registering a data source

This example uses a third-party **`DataSource`** implementation from i-net software to connect to an MS SQL Server database. The comments in the code detail the important points of registering (or initializing) a JDBC data source.

```
// We need to import the actual DataSource implementation
import com.inet.tds.TdsDataSource;
import java.util.Hashtable;
import javax.naming.*;
import javax.naming.directory.*;
import java.sql.* ;
import javax.sql.* ;
public class InitializeJNDI {
    // First we define the relevant parameters for this datasource
    private String serverName = "persistentjava.com";
    private int portNumber = 1433;
    private String login = "java";
    private String password = "sun";
    private String databaseName = "jdbc";
    // This is the name we will assign to our datasource. Because we are
    // using the file system provider, our name follows the file system
    // naming rules. The JNDI reserved subcontext for JDBC applications is
    // jdbc, thus our name starts appropriately.
    private String filePath = "jdbc/pjtutorial";
    public InitializeJNDI() {
      // To pass in the necessary parameters, we need to create and then
      // populate a Hashtable.
        Hashtable env = new Hashtable();
        env.put(
            Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
        try {
            // Create the initial context
            Context ctx = new InitialContext(env);
            // Here we create the actual DataSource and then set the relevant
            // parameters.
            TdsDataSource ds = new TdsDataSource();
            ds.setServerName(serverName);
            ds.setPortNumber(portNumber);
            ds.setDatabaseName(databaseName);
            ds.setUser(login);
            ds.setPassword(password);
            ds.setDescription("JDBC DataSource Connection");
            // Now we bind the DataSource object to the name we selected earlier.
            ctx.bind(filePath, ds);
            ctx.close();
        // Generic Exception handler, in practice, this would be replaced by an
        // appropriate Exception handling hierarchy.
        } catch (Exception ex) {
            System.err.println("ERROR: " + ex.getMessage());
        }
    }
    public static void main(String args[]) {
        new InitializeJNDI();
    }
}
```

Using a data source

The previous example established the binding relationship between the **DataSource** object and a particular name. The JNDI magic is actually performed by an appropriate service provider. In our case, we are using the file system provider. Other options exist, including LDAP (Lightweight Directory Access Protocol) or even a DNS. To actually make a connection, we need to look up the **DataSource** object using the name to which it was bound. Notice in the following example that there is no database-specific code anywhere in sight.

```
import java.util.Hashtable ;
import javax.naming.* ;
import java.sql.* ;
import javax.sql.* ;
public class UtilizeJNDI {
  public UtilizeJNDI(){
    try {
      // We need to set up the JNDI context so that we can properly interface
      // to the correct service provider, in this case the file system.
      Hashtable env = new Hashtable() ;
      env.put(Context.INITIAL_CONTEXT_FACTORY,
              "com.sun.jndi.fscontext.RefFSContextFactory") ;
      Context ctx = new InitialContext(env) ;
      // Given the JNDI Context, we lookup the object and are returned
      // our DataSource
      DataSource ds = (DataSource)ctx.lookup("jdbc/pjtutorial") ;
      // Now we get a database connection and proceed to do our job.
      Connection con = ds.getConnection() ;
      System.out.println("Connection Established.") ;
      con.close();
    // Note that proper error handling is not included here in order to keep
    // the example short.
    }catch(Exception e ) {
      e.printStackTrace();
    }
  }
  public static void main (String args[]){
    new UtilizeJNDI() ;
  }
}
```

Rebinding a data source

When you want to change the particular database or even JDBC **`DataSource`** vendor that you want your code to communicate with, all you need to do is rebind a new **`DataSource`** to the original name. In this example, we use the same driver, but change several relevant parameters:

```
// We need to import the actual DataSource
import com.inet.tds.TdsDataSource;
import java.util.Hashtable;
import javax.naming.*;
import javax.naming.directory.*;
import java.sql.* ;
import javax.sql.* ;
public class InitializeJNDI {
    // First we define the relevant parameters for this datasource
    private String serverName = "persistentjava.com";
    private int portNumber = 1434; // Note the new port number
    private String login = "sun"; // New username/password combination
    private String password = "java";
    private String databaseName = "ds"; // And even a new database name.
    // We keep the same name for our datasource, just bind a new DataSource
    // to it.
    private String filePath = "jdbc/pjtutorial";
    public InitializeJNDI() {
        // Establish the proper JNDI Context
        Hashtable env = new Hashtable();
        env.put(
            Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
        try {
          // Create the context
          Context ctx = new InitialContext(env);
          TdsDataSource ds = new TdsDataSource();
          ds.setServerName(serverName);
          ds.setPortNumber(portNumber);
          ds.setDatabaseName(databaseName);
          ds.setUser(login);
          ds.setPassword(password);
          ds.setDescription("JDBC DataSource Connection, take two");
          // Now we just call the rebind method with the new DataSource.
          ctx.rebind(filePath, ds);
            ctx.close();
        // Replace this with real Exception handlers in production code.
        } catch (Exception ex) {
            System.err.println("ERROR: " + ex.getMessage());
        }
    }
    public static void main(String args[]) {
        new InitializeJNDI();
    }
}
```

Deleting a data source

Sometimes, you will want to delete a **`DataSource`** name so that it can no longer be used:

```
import java.util.Hashtable ;
import javax.naming.* ;
import java.sql.* ;
import javax.sql.* ;
public class DeleteJNDI {
  public DeleteJNDI() {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
    try {
      Context ctx = new InitialContext(env);
      // unbinding the name object association effectively deletes the object.
      ctx.unbind("jdbc/pjtutorial") ;
      ctx.close() ;
    }catch (Exception ex) {
      System.err.println("ERROR: " + ex.getMessage()) ;
    }
  }
  public static void main (String args[]){
    new DeleteJNDI() ;
  }
}
```

Section 6. Connection pools

Why do we need connection pools?

When using either the **`DriverManager`** or the **`DataSource`** method for obtaining database connections, each request for a new database connection involves significant overhead. This can impact performance if obtaining new connections occurs frequently, as might be the case in a Web server environment. To emphasize why this is true, let's follow the potential path of a typical database connection request.

* The Java application calls **`getConnection()`**.
* The JDBC vendor code (either the driver or the **`DataSource`** implementation) requests a socket connection from the JVM.
* The JVM needs to check the security aspects of the potential call. For example, applets are only allowed to communicate with the server from which they originated.
* If approved, the call needs to pass through the host network interface and onto the Corporate LAN.
* The call may need to pass through a firewall to reach the Internet or WAN.
* The call eventually reaches its destination subnet, where it may need to pass through another firewall.
* The call reaches the database host.
* The database server processes the new connection request.
* The license server may need to be queried to determine if an appropriate license is available.
* The database initializes a new client connection, including all memory and operating system overheads.
* The return call is sent back to the JDBC client (where it has to pass through all of the firewalls and routers).
* The JVM receives the return call and creates an appropriate **`Connection`** object.
* The requesting Java application receives the **`Connection`** object.

Clearly requesting a new **`Connection`** object introduces a large overhead and many potential pitfalls. To minimize this overhead, why not just reuse database connections rather than delete them when we are done using them? The JDBC designers used this popular design pattern when creating the **`ConnectionPoolDataSource`**, which allows you to create a pool of database connections that are reused rather than deleted when closed.

What is a PooledConnection?

A **PooledConnection** is a special type of database connection that is not deleted when it is closed, unlike regular **Connection** objects (the garbage collector can delete regular connections once they are no longer referenced). Instead, the **PooledConnection** is cached for later reuse, producing potentially large performance improvements. Working with a pool of database connections is nearly identical to working with **DataSource** objects. First, rather than create an instance of a class that implements the **DataSource** interface, we create an instance of a class that implements the **ConnectionPoolDataSource**. We can use JNDI to bind this new data source to a name as before.

To actually use a pooled data source object, call **getPooledConnection()** on the **ConnectionPooledDataSource**, which in turn establishes the database connection. To create the **Connection** object that will be used, call **getConnection()** on the **PooledConnection** object rather than the **DriverManager** or **DataSource** object as before. One additional benefit of this approach is that it is much easier to manage a set number of database connections with a **ConnectionPool** because it is taken care of automatically. This automation can be very important if your client licenses limit the number of clients that can simultaneously connect to a database.

The entire process is much easier than it sounds, as the following examples show.

Initializing a connection pool

In this example, we use the mSQL database and the open source mSQL JDBC driver to create a **PooledDataSource**. All database-specific code is contained in the initialization or binding process.

```
// First we import the relevant package
import com.imaginary.sql.msql.* ;
import java.util.Hashtable ;
import javax.naming.* ;
public class InitializeJNDI {
  private String serverName = "localhost" ;
  private String databaseName = "jdbc" ;
  private String userName = "java" ;
  private String password = "sun" ;
  // The appropriate JNDI subcontext for PooledDataSources is jdbcpool
  private String filePath = "jdbcPool/pjtutorial" ;
  private int portNumber = 1114 ;
  private int poolSize= 10 ; // We want to create a pool with 10 connections.
  public InitializeJNDI() {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.fscontext.RefFSContextFactory");
    try {
      Context ctx = new InitialContext(env);
      // Create the PooledDataSource and set the relevant parameters.
      MsqlPooledDataSource ds = new MsqlPooledDataSource() ;
      ds.setServerName(serverName) ;
      ds.setPort(portNumber) ;
      ds.setDatabaseName(databaseName) ;
      ds.setUser(userName) ;
      ds.setPassword(password) ;
      ds.setMaxPoolSize(poolSize) ;
      // Bind the name and the DataSource object together
      ctx.bind(filePath, ds) ;
      ctx.close() ;
    } catch (Exception ex) {
      System.err.println("ERROR: " + ex.getMessage()) ;
    }
  }
  public static void main (String args[]){
    new InitializeJNDI() ;
  }
}
```

Using a connection pool

Once we have initialized the **PooledDataSource**, we can now use it in Java applications to create pools of database connections. The following example is somewhat contrived, but it effectively demonstrates the important points. A more realistic example might be a servlet, which would set up the pool of database connections in the servlet **init()** method and (re-)use a connection for each new **service** request.

```
import java.util.Hashtable ;
import javax.naming.* ;
import java.sql.* ;
import javax.sql.* ;
public class UtilizeJNDI {
  public UtilizeJNDI(){
    try {
      Hashtable env = new Hashtable() ;
      env.put(Context.INITIAL_CONTEXT_FACTORY,
              "com.sun.jndi.fscontext.RefFSContextFactory") ;
      Context ctx = new InitialContext(env) ;
      // Look up the DataSource given the appropriate name.
      ConnectionPoolDataSource ds
              = (ConnectionPoolDataSource)ctx.lookup("jdbcPool/pjtutorial") ;
      // A PooledConnection provides a special Connection which is not
      // destroyed when it is closed, but is instead placed back into the
      // pool of connections.
      PooledConnection pcon = ds.getPooledConnection() ;
      Connection con = pcon.getConnection() ;
      System.out.println("Connection Established") ;
      con.close();
    }
    catch(Exception e ) {
      e.printStackTrace();
    }
  }
  public static void main (String args[]){
    new UtilizeJNDI() ;
  }
}
```

Section 7. Optimizing database communications

JDBC DataSource and Driver methods

This section discusses some of the lesser known methods in the JDBC API that can be used to improve system performance. This panel looks specifically at **`DataSource`** and JDBC **`Driver`** methods, while the next panel looks at **`Connection`** methods.

First, all JDBC data source classes, either the **`Driver`**, the **`DataSource`**, or the **`ConnectionPooledDataSource`**, provide the ability to explicitly designate a specific character output stream, **`setLogWriter()`**. This stream accepts all logging and tracing messages. A method is also available to obtain the current stream, **`getLogWriter()`**, as well. This can be extremely valuable when trying to diagnose strange bugs or to trace the flow of your application. In addition, all JDBC data source classes provide a means for assigning (**`setLoginTimeout()`**) or obtaining (**`getLoginTimeout()`**) the maximum amount of time that the data source class should wait for a connection to be established.

One final interesting and seldom used class is the **`DriverPropertyInfo`** class. This class encapsulates all of the property information needed by a driver to establish a database connection. **`DriverPropertyInfo`** can be used by a graphical tool to interactively find the parameters that a driver needs and prompt the database user for the correct values.

JDBC Connection methods

A **`Connection`** or **`PooledConnection`** object also has several methods that can be used to improve system performance, either directly or indirectly. First, assuming the JDBC driver and underlying database support it, a connection object can be set to be read-only, **`setReadOnly()`**. This can improve system performance for applications that do not need to make any database changes, as the database does not need to worry about caching new pages, maintaining journal entries, or acquiring write locks on any of the data.

Two other methods that are somewhat related are the **`setAutoCommit()`** and **`setTransactionIsolation()`** methods, which were discussed earlier in Transaction basics on page 14 . Because transaction mismanangement can cause some of the largest database performance penalties, these methods and their effects should be clearly understood and carefully applied.

Finally, one last and frequently overlooked method is the **`nativeSQL()`** method, which translates a given SQL query into the native query language of the underlying database. By examining the native SQL version of your query, you might be able to better understand how the database is interpreting your query, and then make suitable modifications to your application to improve the overall performance.

Section 8. Summary

Managing database connections

Hopefully this tutorial clearly explained what is going on "under the hood" when Java applications manage database connections. Although there is a great deal of information available online, it's often overly confusing if not contradictory. To clarify this situation, in this tutorial we explored the entire arena of establishing and managing a database connection using JDBC, including:

* The differences in 2-tier and n-tier applications
* The role and four types of JDBC drivers
* The basics behind **`DataSource`** objects
* **`PooledConnection`** objects

Now all that remains is for you to go out and build your own Java database application.

Resources

JDBC information

* Visit the official *JDBC home page* for the JDBC 2.0 and 3.0 specifications and other information.
* *JDBC API Tutorial and Reference, Second Edition* (Addison-Wesley, 1999) by White, Fisher, Cattell, Hamilton, and Hapner is *the* reference for JDBC developers.
* "*What's new in JDBC 3.0*" by Josh Heidebrecht (develperWorks, July 2001) provides an overview of the new features and enhancements in the new spec.
* "*An easy JDBC wrapper*" by Greg Travis (developerWorks, August 2001) describes a simple wrapper library that makes basic database usage a snap.
* The *Java Enablement with DB2* Web site provides an important collection of useful DB2 and Java links.
* "*Improved Performance with a Connection Pool*" (Web Developer's Journal) discusses how **`ConnectionPool`** objects can be used with Java servlets.
* Learn how to write your own Connection pool in "*Implement a JDBC Connection Pool via the Object Pool Pattern*" (developer.com).
* In "*Use JDBC for industrial strength performance*" (developerWorks, January 2000), Lennart Jorelid discusses using server-side Java patterns with JDBC.

JDBC driver information

* To find a JDBC driver for a particular database, visit Sun's *searchable database of JDBC drivers*.

* *Merant* is a third-party vendor that provides JDBC drivers for a range of databases.

* *i-net software* is another third-party vendor for JDBC drivers.

* *SourceForge* offers an open source JDBC driver for the MySQL database.

* The *Center for Imaginary Environments* offers an open source JDBC driver for the mSQL database.

Databases
Two other open-source databases that might be of interest are:

* *PostgreSQL*
* *mySQL*

Much of the example code in this tutorial was developed using *DB2 Universal Database*. If you're using this platform or want to learn more about it from a technical perspective, visit the *DB2 Developer Domain*, a centralized technical resource for the DB2 developer community.

Feedback

We welcome your feedback on this tutorial, and look forward to hearing from you. Additionally, you are welcome to contact the author, Robert Brunner, directly at rjbrunner@pacbell.net.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.