NOTES:
JDBC Java Database Connectivity

foxmula.com

JDBC Java Database Connectivity

Here we'll cover the basics of using JDBC (Java Database Connectivity) for working with databases. You will learn how to perform all the standard database operations, as well as some advanced techniques for manipulating data.

You'll also learn how you can create secure database applications and save time on development using some of the latest advancements in the API. In the end, you will be able to develop Java applications that work with traditional RDBMSs such as Oracle database, PostgreSQL, and MySQL.

Note To follow along with the examples in this blog, run the create_user.sql script to create a database user schema. Then, run the create_database.sql script within the database schema that you just created.

The database examples in this blog are tailored for use with an Apache Derby or Oracle database, but they can be altered to work with any relational database.

Connecting to a Database

Problem You want to create a connection to a database from within a desktop Java application.

Solution 1

Use a JDBC Connection object to obtain the connection. Do this by creating a new connection object, and then load the driver that you need to use for your particular database vendor.

Once the connection object is ready, call its getConnection() method. The following code demonstrates how to obtain a connection to an Oracle or Apache Derby database, depending on the specified driver.

```
 style="margin:0;width:955px;height:165px">public Connection getConnection() throws SQLException { Connection conn = null;
String jdbcUrl;
if(driver.equals("derby")){
jdbcUrl = "jdbc:derby://" + this.hostname + ":" + this.port + "/" + this.database;
} else {
jdbcUrl = "jdbc:oracle:thin:@" + this.hostname + ":" + this.port + ":" + this.database;
}
System.out.println(jdbcUrl);
conn = DriverManager.getConnection(jdbcUrl, username, password); System.out.println("Successfully connected"); return conn;
}
```

The method portrayed in this example returns a Connection object that is ready to be used for database access.

Solution 2

Use a data source to create a connection pool. The DataSource object must have been properly

implemented and deployed to an application server environment.

After a DataSource object has been implemented and deployed, it can be used by an application to obtain a connection to a database. The following code shows the code that you can use to obtain a database connection via a DataSource object:

```
 style="margin:0;width:962px;height:149px">public Connection getDSConnection() {
Connection conn = null;
try {
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/myOracleDS"); conn = ds.getConnection();
} catch (NamingException | SQLException ex) { ex.printStackTrace();
}
return conn;
}
```

Notice that the only information required in the DataSource implementation is the name of a valid DataSource object. All the information that is required to obtain a connection with the database is managed within the application server.

How It Works

There are a couple of different ways to create a connection to a database within a Java application. How you do so depends on the type of application you are writing.

Utilization of the DriverManager is often used if an application will be stand-alone or if it is a desktop application. Web-based and intranet applications commonly rely on the application server to provide the connection for the application via a DataSource object.

Creating a JDBC connection involves a few steps. First, you need to determine which database driver you will need. After you've determined which driver you will need, you download the JAR file containing that driver and place it into your CLASSPATH. For this recipe, either an Oracle database or Apache Derby connection is made.

Each of the database vendors will provide different JDBC drivers packaged in JAR files that have different names; consult the documentation for your particular database for more information.

Once you have obtained the appropriate JAR file for your database, include it in your application CLASSPATH. Next, use a JDBC DriverManager to obtain a connection to the database.

As of JDBC version 4.0, drivers that are contained within the CLASSPATH are automatically loaded into the DriverManager object. If you are using a JDBC version prior to 4.0, the driver will have to be manually loaded.

To obtain a connection to your database using the DriverManager, you need to pass a String containing the JDBC URL to it. The JDBC URL consists of the database vendor name, along with the name of the server that hosts the database, the name of the database, the database port number, and a valid database username and password that has access to the schema or database objects that you want to work with.

Many times, the values used to create the JDBC URL are obtained from a Properties file so that they can be easily changed if needed. The code that is used to create the Oracle database JDBC URL for Solution 1 looks like the following:

```
 style="margin:0;width:922px;height:122px">String jdbcUrl = "jdbc:oracle:thin:@" + this.hostname + ":" + this.port + ":" + this.database;
Once all the variables have been substituted into the String, it will look something like the following:
jdbc:oracle:thin:@hostname:1521:database
Similarly, the Apache Derby URL String would look like the following:
jdbc:derby://hostname:1521/database
```

Once the JDBC URL has been created, it can be passed to the DriverManager.getConnection() method to obtain a java.sql.Connection object. If incorrect information has been passed to the getConnection() method, a java.sql.SQLException will be thrown; otherwise, a valid Connection object will be returned.

The preferred way to obtain a database connection is to use a DataSource when running on an application server or to have access to a Java Naming and Directory Interface (JNDI) service.

To work with a DataSource object, you need to have an application server deploy it to. Any compliant Java application server such as GlassFish, Oracle Weblogic, Payara, or WildFly will work.

Most of the application servers contain a web interface that can be used to easily deploy a DataSource object. However, you can manually deploy a DataSource object by using code that will look like the following:

```
 style="margin:0;width:925px;height:89px">org.java9recipes.blog13.recipe13_01.FakeDataSourceDriver ds =
new org.java9recipes.blog13.recipe13_1.FakeDataSourceDriver();
ds.setServerName("my-server");
ds.setDatabaseName("JavaThesis");
ds.setDescription("Database connection for JavaThesis");
```

This code instantiates a new DataSource driver class and then sets properties based on the database that you want to register. DataSource code such as that demonstrated here is typically used when registering a DataSource in an application server or with access to a JNDI server.

Application servers usually do this work behind the scenes if you are using a web-based administration tool to deploy a DataSource. Most database vendors will supply a DataSource driver along with their JDBC drivers, so if the correct JAR resides within the application or server CLASSPATH, it should be recognized and available for use.

Once a DataSource has been instantiated and configured, the next step is to register the DataSource with a JNDI naming service.

The following code demonstrates the registration of a DataSource with JNDI:

```
 style="margin:0;width:962px;height:90px">try {
Context ctx = new InitialContext();
DataSource ds =
(DataSource) ctx.bind("java9recipesDB");
} catch (NamingException ex) { ex.printStackTrace();
}
```

Once the DataSource has been deployed, any application that has been deployed to the same application server will have access to it.

The beauty of working with a DataSource object is that your application code doesn't need to know anything about the database; it only needs to know the name of the DataSource. Usually, the name of the DataSource begins with a jdbc/ prefix, followed by an identifier.

To look up the DataSource object, an InitialContext is used. The InitialContext looks at all the data sources available within the application server and returns a valid DataSource if it is found;

otherwise, it will throw a java.naming.NamingException exception. In Solution 2, you can see that the InitialContext returns an object that must be cast as a DataSource.

```
Context ctx = new InitialContext();

DataSource ds = (DataSource)ctx.lookup("jdbc/myOracleDS");
```

If the DataSource is a connection pool cache, it will send one of the available connections within the pool when an application requests it. The following line of code returns a Connection object from the DataSource:

```
conn = ds.getConnection();
```

Of course, if no valid connection can be obtained, a java.sql.SQLException is thrown. The DataSource technique is preferred over the DriverManager because database connection information is stored in only one place: the application server. Once a valid DataSource is deployed, it can be used by many applications.

After a valid connection has been obtained by your application, it can be used to work with the database.

Handling Connection and SQL Exceptions

Problem A database activity in your application has thrown an exception. You need to handle the SQL exception so that your application does not crash.

Solution

Use a try-catch block in order to capture and handle any SQL exceptions that are thrown by your JDBC connection or SQL queries. The following code demonstrates how to implement a try-catch block in order to capture SQL exceptions:

```
 style="margin:0;height:89px;width:905px">try {
perform database tasks
} catch (java.sql.SQLException){
perform exception handling
}
```

How It Works

A standard try-catch block can be used to catch java.sql.Connection or java.sql.SQLException exceptions. Your code will not compile if these exceptions are not handled, and it is a good idea to handle them properly in order to prevent your application from crashing if one of these exceptions is thrown.

Almost any work that is performed against a java.sql.Connection object will need to contain error handling to ensure that database exception are handled correctly. In fact, nested try-catch blocks are often required to handle all the possible exceptions.

You need to ensure that connections are closed once work has been performed and the Connection object is no longer used. Similarly, it is a good idea to close java.sql.Statement objects for memory allocation cleanup as well.

Because Statement and Connection objects need to be closed, it is common to see try-catch-finally blocks used to ensure that all resources have been tended to as needed. It is not unlikely that you will see older JDBC code that resembles the following style:

```
 style="margin:0;width:910px;height:150px">try {
perform database tasks
} catch (java.sql.SQLException ex) {
perform exception handling } finally {
try {
close Connection and Statement objects } catch (java.sql.SQLException ex){
perform exception handling
}
}
```

The newer code should be written to take advantage of the try-with-resources statement, which allows one to offload resource management to Java, rather than performing manual closes.

The following code demonstrates how to use try-with-resources to open a connection, create a statement, and then close both the connection and statement when finished.

Note The createConn object in the examples abstracts away the details of obtaining a connection to the database, which can be returned via a call to the getConnection() method.

```
 style="margin:0;width:965px;height:109px">try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement();) {
ResultSet rs = stmt.executeQuery(qry);
while (rs.next()) {
// PERFORM SOME WORK
}
} catch (SQLException e) { e.printStackTrace();
}
```

As seen in the previous pseudo code, nested try-catch blocks are often required in order to clean unused resources. Proper exception handling sometimes makes JDBC code rather laborious to write, but it will also ensure that an application requiring database access will not fail, causing data to be lost.

Querying a Database and Retrieving Results

Problem A process in your application needs to query a database table for data.

Solution: Obtain a JDBC connection using one of the techniques as described in Recipe, and then use the java.sql.Connection object to create a Statement object. A java.sql.Statement object contains the executeQuery() method, which parses a String of text and uses it to query a database.

Once you've executed the query, you can retrieve the results of the query into a ResultSet object. The following example queries a database table named RECIPES and prints the results:

```
 style="margin:0;width:961px;height:148px">String qry = "select recipe_num, name, description from recipes"; try (Connection conn = createCo
nn.getConnection();
Statement stmt = conn.createStatement();) { ResultSet rs = stmt.executeQuery(qry); while (rs.next()) {
String recipe = rs.getString("RECIPE_NUM");
String name = rs.getString("NAME");
String desc = rs.getString("DESCRIPTION");
System.out.println(recipe + "\t" + name + "\t" + desc);
}
} catch (SQLException e) { e.printStackTrace();
}
```

If you execute this code using the database script that is included with this blog, you will receive the following results:

- Connecting to a Database DriverManager and DataSource Implementations
- Querying a Database and Retrieving Results Obtaining and Using Data from a DBMS
- Handling SQL Exceptions Using SQLException

How It Works

One of the most commonly performed operations against a database is a query. Performing database queries using JDBC is quite easy, although there is a bit of boilerplate code that needs to be used each time a query is executed. First, you need to obtain a Connection object for the database and schema that you want to run the query against.

Next, you need to form a query and store it in String format. The Connection object is then used to create a Statement. Your query String will be passed to the Statement object's executeQuery() method in order to actually query the database.

Here, you can see what this looks like without the use of try-with-resources for resource management.

```
 style="margin:0;height:169px;width:957px">String qry = "select recipe_num, name, description from recipes";
Connection conn;
Statement stmt = null;
try {
conn = createConn.getConnection()
stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(qry);
...
The same code can be more efficiently written as follows:
try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement();) {
ResultSet rs = stmt.executeQuery(qry);
```

As you can see, the Statement object's executeQuery() method accepts a String and returns a ResultSet object. The ResultSet object makes it easy to work with the query results so that you can obtain the information you need in any order. If you take a look at the next line of code in the example, a while loop is created on the ResultSet object.

This loop will continue to call the ResultSet object's next() method, obtaining the next row that is returned from the query with each iteration.

In this case, the ResultSet object is named rs, so while rs.next() returns true, the loop will continue to be processed. Once all the returned rows have been processed, rs.next() will return a false to indicate that there are no more rows to be processed.

Within the while loop, each returned row is processed. The ResultSet object is parsed to obtain the values of the given column names with each pass. Notice that if the column is expected to return a String, you must call the ResultSet getString() method, passing the column name in String format.

Similarly, if the column is expected to return an int, you'd call the ResultSet getInt() method, passing the column name in String format. The same holds true for the other data types.

These methods will return the corresponding column values. In the example in the solution to this recipe, those values are stored into local variables.

```
 style="margin:0;width:968px;height:104px">String recipe = rs.getString("RECIPE_NUM");

String name = rs.getString("NAME");

String desc = rs.getString("DESCRIPTION");
```

Once the column value has been obtained, you can do what you want to do with the values you have stored within local variables. In this case, they are printed out using the System.out() method.

```
System.out.println(recipe + "\t" + name + "\t" + desc);
```

A java.sql.SQLException could be thrown when attempting to query a database (for instance, if the Connection object has not been properly obtained or if the database tables that you are trying to query do not exist).

You must provide exception handling to handle errors in these situations. Therefore, all database-processing code should be placed within a try block.

The catch block then handles an SQLException, so if one is thrown, the exception will be handled using the code within the catch block. Sounds easy enough, right? It is, but you must do it each time you perform a database query. Lots of boilerplate code.

It is always a good idea to close statements and connections if they are open. Using the try-with-resources construct is the most efficient solution to resource management.

Closing resources when finished will help ensure that the system can reallocate resources as needed, and act respectfully on the database. It is important to close connections as soon as possible so that other processes can use them.

Performing CRUD Operations

Problem

You need to have the ability to perform standard database operations within your application. That is, you need the ability to create, retrieve, update, and delete (CRUD) database records.

Solution

Create a Connection object and obtain a database connection using one of the solutions provided in Recipe; then perform the CRUD operation using a java.sql.Statement object that is obtained from the java.sql.Connection object. The database table that will be used for these operations has the following format:

```
 style="margin:0;width:970px;height:302px">RECIPES (
id int not null,
recipe_number varchar(10) not null,
recipe_name varchar(100) not null,
description varchar(500),
text clob,
constraint recipes_pk primary key (id) enable
);
```

The following code excerpts demonstrate how to perform each of the CRUD operations using JDBC:

```
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import org.java9recipes.blog13.recipe13_01.CreateConnection;
public class CrudOperations {
static CreateConnection createConn;
public static void main(String[] args) {
createConn = new CreateConnection();
performCreate();
performRead();
performUpdate();
performDelete();
System.out.println("-- Final State --");
performRead();
}
private static void performCreate(){
String sql = "INSERT INTO RECIPES VALUES(" +
"next value for recipes_seq, " +
"'13-4', " +
"'Performing CRUD Operations', " +
"'How to perform create, read, update, delete functions', " + "'Recipe Text')";
try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement();) {
Returns row-count or 0 if not successful int result = stmt.executeUpdate(sql);
if (result == 1{
System.out.println("-- Record created --"); } else {
System.err.println("!! Record NOT Created !!");
}
} catch (SQLException e) {
e.printStackTrace();
}
}
private static void performRead(){
String qry = "select recipe_number, recipe_name, description from recipes";
try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement();) {
ResultSet rs = stmt.executeQuery(qry);
while (rs.next()) {
String recipe = rs.getString("RECIPE_NUMBER"); String name = rs.getString("RECIPE_NAME"); String desc = rs.getString("DESCRIPTION");
System.out.println(recipe + "\t" + name + "\t" + desc);
}
} catch (SQLException e) { e.printStackTrace();
}
}
private static void performUpdate(){
String sql = "UPDATE RECIPES " +
"SET RECIPE_NUMBER = '13-5' " +
"WHERE RECIPE_NUMBER = '13-4'";
try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement();) {
int result = stmt.executeUpdate(sql);
if (result > 0){
System.out.println("-- Record Updated --"); } else {
System.out.println("!! Record NOT Updated !!");
}
} catch (SQLException e) { e.printStackTrace();
}
}
private static void performDelete(){
```

```
private static void performDelete(){
String sql = "DELETE FROM RECIPES WHERE RECIPE_NUMBER = '13-5'";
try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement();) {
int result = stmt.executeUpdate(sql);
if (result > 0){
System.out.println("-- Record Deleted --"); } else {
System.out.println("!! Record NOT Deleted!!");
}
} catch (SQLException e) { e.printStackTrace();
}
}
}
```

Here is the result of running the code:

```
 style="margin:0;height:197px;width:971px">Successfully connected
-- Record created --
Connecting to a Database—DriverManager and DataSource Implementations
Querying a Database and Retrieving Results Obtaining and Using Data from a DBMS
Handling SQL Exceptions Using SQLException
Performing CRUD Operations How to Perform Create, Read, Update, Delete Functions
Record Updated --
Record Deleted --
Final State --
Connecting to a Database DriverManager and DataSource Implementations
Querying a Database and Retrieving Results Obtaining and Using Data from a DBMS
Handling SQL Exceptions Using SQLException
```

How It Works

The same basic code format is used for performing just about every database task. The format is as follows:

- \1.\ Obtain a connection to the database.
- \2.\ Create a statement from the connection.
- \3.\ Perform a database task with the statement.
- \4.\ Do something with the results of the database task.
- \5.\ Close the statement (and database connection if you’re finished using it).

The main difference between performing a query using JDBC and using data manipulation language (DML) is that you will call different methods on the Statement object, depending on which operation you want to perform.

To perform a query, you need to call the Statement executeQuery() method. In order to perform DML tasks such as insert, update, and delete, call the executeUpdate() method.

The perform create() method in the solution to this recipe demonstrates the operation of inserting a record into a database. To insert a record in the database, construct a SQL INSERT statement in String format. To perform the insert, pass the SQL String to the Statement object’s executeUpdate() method.

If the INSERT is performed, an int value will be returned that specifies the number of rows that have been inserted. If the INSERT operation is not performed successfully, either a zero will be returned or

been inserted. If the INSERT operation is not performed successfully, either a zero will be returned or an SQLException will be thrown, indicating a problem with the statement or database connection.

The performRead() method in the solution to this recipe demonstrates the operation of querying the database. To execute a query, call the Statement object's executeQuery() method, passing a SQL statement in String format. The result will be a ResultSet object, which can then be used to work with the returned data. For more information on performing queries.

The perform update() method in the solution to this recipe demonstrates the operation of updating record(s) within a database table. First, construct a SQL UPDATE statement in String format. Next, to perform the update operation pass the SQL String to the Statement object's executeUpdate() method.

If the UPDATE is successfully performed, an int value will be returned, which specifies the number of records that were updated. If the UPDATE operation is not performed successfully, either a zero will be returned or an SQLException will be thrown, indicating a problem with the statement or database connection.

The last database operation that needs to be covered is the DELETE operation. The perform delete() method in the solution to this recipe demonstrates the operation of deleting a record(s) from the database.

First, construct a SQL DELETE statement in String format. Next, to execute the deletion, pass the SQL String to the Statement object's executeUpdate() method.

If the deletion is successful, an int value specifying the number of rows deleted will be returned. Otherwise, if the deletion fails, a zero will be returned or an SQLException will be thrown, indicating a problem with the statement or database connection.

Almost every database application uses at least one of the CRUD operations at some point. This is foundational JDBC that needs to be known if you are working with databases within Java applications. Even if you will not work directly with the JDBC API, it is good to know these foundational basics.

Simplifying Connection Management

Problem

Your application requires the use of a database, and in order to work with the database, you need to open a connection for each interaction. Rather than code the logic to open a database connection every time you need to access the database, you want to use a single class to perform that task.

Solution

Write a class to handle all the connection management within your application. Doing so will allow you to call that class in order to obtain a connection, rather than setting up a new Connection object each time you need access to the database. Perform the following steps to set up a connection management environment for your JDBC application:

- \1.\ Create a class named CreateConnection.java that will encapsulate all the connection logic for your application.
- \2.\ Create a PROPERTIES file to store your connection information. Place the file somewhere on your CLASSPATH so that the CreateConnection class can load it.
- \3.\ Use the CreateConnection class to obtain your database connections.

The following code is a listing of the CreateConnection class that can be used for centralized connection management:

```
 style="margin:0;width:960px;height:333px">import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.nio.file.FileSystems;
import java.nio.file.Files;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
public class CreateConnection {
```

```
public class CreateConnection {
static Properties props = new Properties();
String hostname = null;
String port = null;
String database = null;
String username = null;
String password = null;
String driver = null;
String jndi = null;
public CreateConnection() {
Looks for properties file in the root of the class='lazy' data-src directory in Netbeans project try (InputStream in = Files.newInputStream(FileSys
tems.getDefault().
getPath(System.getProperty("user.dir") + File.separator + "db_props. properties"));) {
props.load(in);
in.close();
} catch (IOException ex) { ex.printStackTrace();
}
loadProperties();
}
public final void loadProperties() {
hostname = props.getProperty("host_name");
port = props.getProperty("port_number");
database = props.getProperty("db_name");
username = props.getProperty("username");
password = props.getProperty("password");
driver = props.getProperty("driver");
jndi = props.getProperty("jndi");
}
public Connection getConnection() throws SQLException { Connection conn = null;
String jdbcUrl;
if (driver.equals("derby")) {
jdbcUrl = "jdbc:derby://" + this.hostname + ":"
+ this.port + "/" + this.database;
} else {
jdbcUrl = "jdbc:oracle:thin:@" + this.hostname + ":"
+ this.port + ":" + this.database;
}
conn = DriverManager.getConnection(jdbcUrl, username, password); System.out.println("Successfully connected"); return conn;
}
/**
Demonstrates obtaining a connection via a DataSource object
@return
*/
public Connection getDSConnection() {
Connection conn = null;
try {
Context ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup(this.jndi); conn = ds.getConnection();
} catch (NamingException | SQLException ex) { ex.printStackTrace();
}
return conn;
}
}
```

Next, the following lines of text are an example of what should be contained in the properties file that is used for obtaining a connection to the database. For this example, the properties file is named db_props. properties:

```
host_name=your_db_server_name
db_name=your_db_name
username=db_username
password=db_username_password
port_number=db_port_number
#driver = derby or oracle
driver=db_driver
jndi=jndi_connection_String
```

Finally, use the CreateConnection class to obtain connections for your application. The following code demonstrates this concept:

```
CreateConnection createConn = new CreateConnection(); try(Connection conn = createConn.getConnection()) {
performDbTask();
} catch (java.sql.SQLException ex) {
ex.printStackTrace();
}
```

This code uses try-with-resources to automatically close the connection when it is finished performing the database task.

How It Works

Obtaining a connection within a database application can be code-intensive. Moreover, the process can be prone to error if you retype the code each time you need to obtain a connection.

By encapsulating database connection logic within a single class, you can reuse the same connection code each time you require a connection to the database.

This increases your productivity, reduces the chances of typing errors, and also enhances manageability because if you have to make a change, it can occur in one place rather than in several different locations.

Creating a strategic connection methodology is beneficial to you and others who might need to maintain your code in the future. Although data sources are the preferred technique for managing database connections when using an application server or JNDI, the solution to this recipe demonstrates the use of standard JDBC DriverManager connections.

One of the security implications of using the DriverManager is that you will need to store the database credentials somewhere for use by the application.

It is not safe to store those credentials in plain text anywhere, and it is also not safe to embed them in application code, which might be decompiled at some point in the future.

As seen in the solution, properties file that on disk is used to store the database credentials. Assume that this properties file will be encrypted at some point before deployment to a server and that the application will be able to handle decryption.

As seen in the solution, the code reads the database credentials, hostname, database name, and port number from the properties file. That information is then pieced together to form a JDBC URL that can be used by DriverManager to obtain a connection to the database.

Once obtained, that connection can be used anywhere and then closed. Similarly, if using a DataSource that has been deployed to an application server, the properties file can be used to store the JNDI connection.

That is the only piece of information that is needed to obtain a connection to the database using the DataSource. To the developer using the connection class, the only difference between the two types of connections is the method name that is called in order to obtain the Connection object.

You could develop a JDBC application so that the code used to obtain a connection needs to be hard-coded throughout. Instead, this solution enables all the code for obtaining a connection to be encapsulated by a single class so that the developer does not need to worry about it. Such a technique also allows the code to become more maintainable.

For instance, if the application were originally deployed using the DriverManager, but then later had the ability to use a DataSource, very little code would need to be changed.

Guarding Against SQL Injection

Guarding Against SQL Injection

Problem

Your application performs database tasks. To reduce the chances of a SQL injection attack, you need to ensure that no unfiltered Strings of text are being appended to SQL statements and executed against the database.

Tip Although prepared statements are the solution to this recipe, they can be used for more than just protecting against SQL injection. They also provide a way to centralize and better control the SQL used in an application.

Instead of creating multiple, possibly different, versions of the same query, for example, you can create the query once as a prepared statement and invoke it from many different places throughout your code. Any change to the query logic need happen only at the point where you prepare the statement.

Solution

Use PreparedStatements for performing the database tasks. PreparedStatements send a precompiled SQL statement to the DBMS rather than a String. The following code demonstrates how to perform a database query and a database update using a java.sql.PreparedStatement object.

In the following code example, a PreparedStatement is used to query a database for a given record. Assume that the String[] of recipe numbers is passed to this code as a variable.

```
 style="margin:0;width:977px;height:372px">private static void queryDbRecipe(String[] recipeNumbers) {
String sql = "SELECT ID, RECIPE_NUMBER, RECIPE_NAME, DESCRIPTION "
"FROM RECIPES "
"WHERE RECIPE_NUMBER = ?";
try (PreparedStatement pstmt = conn.prepareStatement(sql)) { for (String recipeNumber : recipeNumbers) {
pstmt.setString(1, recipeNumber);
ResultSet rs = pstmt.executeQuery();
while (rs.next()) {
System.out.println(rs.getString(2) + ": " + rs.getString(3)
+ " - " + rs.getString(4));
}
```

```
}
} catch (SQLException ex) {
ex.printStackTrace();
}
}
The next example demonstrates the use of a PreparedStatement for inserting a record into the database.
Assume that the recipeNumber, title, description, and text Strings are passed to this code as variables.
String sql = "INSERT INTO RECIPES VALUES(" +
"NEXT VALUE FOR RECIPES_SEQ, ?,?,?,?)"; try(PreparedStatement pstmt = conn.prepareStatement(sql);) {
pstmt.setString(1, recipeNumber);
pstmt.setString(2, title);
pstmt.setString(3, description);
pstmt.setString(4, text);
pstmt.executeUpdate();
System.out.println("Record successfully inserted.");
} catch (SQLException ex){ ex.printStackTrace();
}
In this last example, a PreparedStatement is used to delete a record from the database. Again, assume that the recipeNumber String is passe
d to this code as a variable.
String sql = "DELETE FROM RECIPES WHERE " +
"RECIPE_NUMBER = ?";
try(PreparedStatement pstmt = conn.prepareStatement(sql);) { pstmt.setString(1, recipeNumber); pstmt.executeUpdate();
System.out.println("Recipe " + recipeNumber + " successfully deleted.");
} catch (SQLException ex){ ex.printStackTrace();
}
```

As you can see, a PreparedStatement is very much the same as a standard JDBC statement object, but instead it sends precompiled SQL to the DBMS rather than Strings of text.

How It Works

While standard JDBC statements will get the job done, the harsh reality is that they can sometimes be insecure and cumbersome to work with. For instance, bad things can occur if a dynamic SQL statement is used to query a database, and a user-accepted String is assigned to a variable and concatenated with the intended SQL String.

In most ordinary cases, the user-accepted String would be concatenated, and the SQL String would be used to query the database as expected. However, an attacker could decide to place malicious code inside of the String (a.k.a. SQL Injection), which would then be inadvertently sent to the database using a standard Statement object.

The use of PreparedStatements prevents such malicious Strings from being concatenated into a SQL String and passed to the DBMS because they use a different approach. PreparedStatements use substitution variables rather than concatenation to make SQL Strings dynamic.

They are also precompiled, which means that a valid SQL String is formed prior to the SQL being sent to the DBMS. Moreover, PreparedStatements can help your application perform better because if the same SQL has to be run more than one time, it has to be compiled only once.

After that, the substitution variables are interchangeable, but the overall SQL can be executed by the PreparedStatement very quickly.

Let's take a look at how a PreparedStatement works in practice. If you look at the first example in the solution to this recipe, you can see that the database table RECIPES is being queried, passing a RECIPE_ NUMBER and retrieving the results for the matching record. The SQL String looks like the following:

```
String sql = "SELECT ID, RECIPE_NUMBER, RECIPE_NAME, DESCRIPTION " + "FROM RECIPES " +

"WHERE RECIPE_NUM = ?";
```

Everything looks standard with the SQL text except for the question mark (?) at the end of the String. Placing a question mark in a String of SQL signifies that a substitute variable will be used in place of that question mark when the SQL is executed.

The next step for using a PreparedStatement is to declare a variable of type PreparedStatement. This can be seen with the following line of code:

```
PreparedStatement pstmt = null;
```

A PreparedStatement implements AutoCloseable, and therefore it can be utilized within the context of a try-with-resources block. Once a PreparedStatement has been declared, it can be put to use.

However, use of a PreparedStatement might not cause an exception to be thrown. Therefore, in the event that try-with-resources is not used, a PreparedStatement should occur within a try-catch block so that any exceptions can be handled gracefully.

For instance, exceptions can occur if the database connection is unavailable for some reason or if the SQL String is invalid.

Rather than crashing an application due to such issues, it is best to handle the exceptions wisely within a catch block. The following try-catch block includes the code that is necessary to send the SQL String to the database and retrieve results:

```
 style="margin:0;width:973px;height:97px">try(PreparedStatement pstmt = conn.prepareStatement(sql);) { pstmt.setString(1, recipeNumber);
ResultSet rs = pstmt.executeQuery();
while(rs.next()){
System.out.println(rs.getString(2) + ": " + rs.getString(3) + " - " + rs.getString(4));
}
} catch (SQLException ex) { ex.printStackTrace();
}
```

First, you can see that the Connection object is used to instantiate a PreparedStatement object. The SQL String is passed to the PreparedStatement object's constructor on creation. Since the PreparedStatement is instantiated within the try-with-resources construct, it will be automatically closed when it is no longer in use.

Next, the PreparedStatement object is used to set values for any substitution variables that have been placed into the SQL String. As you can see, the PreparedStatement setString() method is used in the example to set the substitution variable at position 1 equal to the contents of the recipeNumber variable.

The positioning of the substitution variable is associated with the placement of the question mark (?) within the SQL String. The first question mark within the String is assigned to the first position, the second one is assigned to the second position, and so forth.

If there were more than one substitution variable to be assigned, there would be more than one call against the PreparedStatement, assigning each of the variables until each one has been accounted for. PreparedStatements can accept substitution variables of many different data types.

For instance, if an int value were being assigned to a substitution variable, a call to the setInt(position

For instance, if an int value were being assigned to a substitution variable, a call to the setInt(position, variable) method would be in order. See the online documentation or your IDE's code completion for a complete set of methods that can be used for assigning substitution variables using PreparedStatement objects.

Once all the variables have been assigned, the SQL String can be executed. The PreparedStatement object contains an executeQuery() method that is used to execute a SQL String that represents a query.

The executeQuery() method returns a ResultSet object, which contains the results that have been fetched from the database for the particular SQL query. Next, the ResultSet can be traversed to obtain the values retrieved from the database.

Again, positional assignments are used to retrieve the results by calling the ResultSet object's corresponding getter methods and passing the position of the column value that you want to obtain.

The position is determined by the order in which the column names appear within the SQL String. In the example, the first position corresponds to the RECIPE_NUMBER column, the second corresponds to the RECIPE_NAME column, and so forth.

If the recipeNumber String variable was equal to "13-1," the results of executing the query in the example would look something like the following:

Connecting to a Database - DriverManager and DataSource Implementations

Of course, if the substitution variable is not set correctly or if there is an issue with the SQL String, an exception will be thrown.

This would cause the code that is contained within the catch block to be executed. You should also be sure to clean up after using PreparedStatements by closing the statement when you are finished using it.

If you're not using a try-with-resources construct, it is a good practice to put all the cleanup code within a finally block to be sure that the PreparedStatement is closed properly even if an exception is thrown. In the example, the finally block looks like the following:

```
 style="margin:0;width:940px;height:124px">finally {
if (pstmt != null){
try {
pstmt.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
```

You can see that the PreparedStatement object that was instantiated, pstmt, is checked to see whether it is NULL. If not, it is closed by calling the close() method.

Working through the code in the solution to this recipe, you can see that similar code is used to process database INSERT, Update, and DELETE statements.

The only difference in those cases is that the PreparedStatement executeUpdate() method is called rather than the executeQuery() method. The executeUpdate() method will return an int value representing the number of rows affected by the SQL statement.

The use of PreparedStatement objects is preferred over JDBC Statement objects. This is due to the fact that they are more secure and perform better. They can also make your code easier to follow and maintain.

Performing Transactions

Problem

The way in which your application is structured requires a sequential processing of tasks. One task depends on another, and each process performs a different database action. If one of the tasks along the way fails, the database processing that has already occurred needs to be reversed.

Solution

Set your Connection object autocommit to false and then perform the transactions you want to complete. Once you've successfully performed each of the transactions, manually commit the Connection object; otherwise, roll back each of the transactions that have taken place. The following code example demonstrates transaction management.

If you look at the main() method of the transaction example class, you will see that the Connection object's autoCommit() preference has been set to false, so that database statements are grouped together to form one transaction.

If all the statements within the transaction are successful, the Connection object is manually committed by calling the commit() method; otherwise, all the statements are rolled back by calling the rollback() method. By default, autoCommit is set to true, which automatically treats every statement as a single transaction.

```
 style="margin:0;width:945px;height:351px">import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.java9recipes.blog13.recipe13_01.CreateConnection;
```

```
public class TransactionExample {
public static Connection conn = null;
public static void main(String[] args) {
boolean successFlag = false;
try {
CreateConnection createConn = new CreateConnection();
conn = createConn.getConnection();
conn.setAutoCommit(false);
queryDbRecipes();
successFlag = insertRecord(
"13-6",
"Simplifying and Adding Security with Prepared Statements", "Working with Prepared Statements", "Recipe Text");
if (successFlag == true){
successFlag = insertRecord(
"13-6B",
"Simplifying and Adding Security with Prepared Statements", "Working with Prepared Statements", "Recipe Text");
}
Commit Transactions if (successFlag == true) conn.commit();
else
conn.rollback();
conn.setAutoCommit(true);
queryDbRecipes();
} catch (java.sql.SQLException ex) { System.out.println(ex);
} finally {
if (conn != null) {
try {
conn.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
}
private static void queryDbRecipes(){
String sql = "SELECT ID, RECIPE_NUMBER, RECIPE_NAME, DESCRIPTION " + "FROM RECIPES";
try(PreparedStatement pstmt = conn.prepareStatement(sql);) { ResultSet rs = pstmt.executeQuery(); while(rs.next()){
System.out.println(rs.getString(2) + ": " + rs.getString(3) + " - " + rs.getString(4));
}
} catch (SQLException ex) {
ex.printStackTrace();
}
}
private static boolean insertRecord(String recipeNumber, String title,
String description,
String text){
String sql = "INSERT INTO RECIPES VALUES(" +
"NEXT VALUE FOR RECIPES_SEQ, ?,?,?,?)"; boolean success = false;
try(PreparedStatement pstmt = conn.prepareStatement(sql);) { pstmt.setString(1, recipeNumber); pstmt.setString(2, title);
pstmt.setString(3, description);
pstmt.setString(4, text);
pstmt.executeUpdate();
System.out.println("Record successfully inserted.");
success = true;
} catch (SQLException ex){
success = false;
ex.printStackTrace();
}
return success;
}
}
```

In the end, if any of the statements fail, all transactions will be rolled back. However, if all the statements execute properly, everything will be committed.

How It Works

How it works

Transaction management can play an important role in an application. This holds true especially for applications that perform different tasks that depend on each other. In many cases, if one of the tasks performed within a transaction fails, it is preferable for the entire transaction to fail rather than having it only partially complete.

For instance, imagine that you were adding database user records to your application database. Now let's say that adding a user for your application required a couple of different database tables to be modified, maybe a table for roles, and so on.

What would happen if your first table was modified correctly and the second table modification failed? You would be left with a partially complete application user addition, and your user would most likely not be able to access the application as expected.

In such a situation, it would be nicer to roll back all the already-completed database modifications if one of the updates failed so that the database was left in a clean state and the transaction could be

attempted once again.

By default, a Connection object is set up so that autocommit is turned on. That means that each database INSERT, UPDATE, or DELETE statement is committed right away. Usually, this is the way that you will want your applications to function.

However, in circumstances where you have many database statements that rely on one another, it is important to turn off autocommit so that all the statements can be committed at once. To do so, call the Connection object's setAutoCommit() method and pass a false value.

As you can see in the solution to this recipe, the setAutoCommit() method is called passing a false value, the database statements are executed. Doing so will cause all the database statement changes to be temporary until the Connection object's commit() method is called.

This provides you with the ability to ensure that all the statements execute properly before issuing a commit(). Take a look at this transaction management code that is contained within the main() method of the transaction example class within the solution to this recipe:

```
 style="margin:0;width:936px;height:278px">boolean successFlag = false;
...
CreateConnection createConn = new CreateConnection();
conn = createConn.getConnection();
conn.setAutoCommit(false);
queryDbRecipes();
successFlag = insertRecord(
"13-6",
"Simplifying and Adding Security with Prepared Statements", "Working with Prepared Statements", "Recipe Text");
if (successFlag == true){
successFlag = insertRecord(
null,
"Simplifying and Adding Security with Prepared Statements", "Working with Prepared Statements", "Recipe Text");
}
Commit Transactions if (successFlag == true) conn.commit();
else
conn.rollback();
conn.setAutoCommit(true);
```

Note that the commit() method is called only if all transaction statements were processed successfully. If any of them fail, the successFlag is equal to false, which would cause the rollback() method to be called instead.

In the solution to this recipe, the second call to insertRecord() attempts to insert a NULL value into the RECIPE.ID column, which is not allowed. Therefore, that insert fails and everything, including the previous insert, gets rolled back.

Creating a Scrollable ResultSet

Problem

You have queried the database and obtained some results. You want to store those results in an object that will allow you to traverse forward and backward through the results, updating values as needed.

Solution

Create a scrollable ResultSet object and then you will have the ability to read the next, first record, last, and previous record. Using a scrollable ResultSet allows the results of a query to be fetched in any direction so that the data can be retrieved as needed. The following example method demonstrates the creation of a scrollable ResultSet object:

```
 style="margin:0;width:948px;height:283px">private static void queryDbRecipes(){
String sql = "SELECT ID, RECIPE_NUMBER, RECIPE_NAME, DESCRIPTION " + "FROM RECIPES";
try(PreparedStatement pstmt =conn.prepareStatement(sql,
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = pstmt.executeQuery()) {
rs.first();
System.out.println(rs.getString(2) + ": " + rs.getString(3) + " - " + rs.getString(4));
rs.next();
System.out.println(rs.getString(2) + ": " + rs.getString(3) + " - " + rs.getString(4));
rs.previous();
System.out.println(rs.getString(2) + ": " + rs.getString(3) + " - " + rs.getString(4));
rs.last();
System.out.println(rs.getString(2) + ": " + rs.getString(3) +
- " + rs.getString(4));
} catch (SQLException ex) { ex.printStackTrace();
```

```
} catch (SQLException ex) { ex.printStackTrace();
}
}
```

Executing this method will result in the following output using the data that was originally loaded for this blog:

```
Successfully connected
13-1: Connecting to a Database - DriverManager and DataSource Implementations - More to Come
13-2: Querying a Database and Retrieving Results - Obtaining and Using Data from a DBMS
13-1: Connecting to a Database - DriverManager and DataSource Implementations - More to Come
13-3: Handling SQL Exceptions - Using SQLException
```

How It Works

Ordinary ResultSet objects allow results to be fetched in a forward direction. That is, an application can process a default ResultSet object from the first record retrieved forward to the last. Sometimes an application requires more functionality when it comes to traversing a ResultSet.

For instance, let's say you want to write an application that allows for someone to display the first or last record that was retrieved, or perhaps page forward or backward through results.

You could not do this very easily using a standard ResultSet. However, by creating a scrollable ResultSet, you can easily move backward and forward through the results.

To create a scrollable ResultSet, you must first create an instance of a Statement or PreparedStatement that has the ability to create a scrollable ResultSet. That is, when creating the Statement, you must pass the ResultSet scroll type constant value to the Connection object's createStatement() method.

Likewise, you must pass the scroll type constant value to the Connection object's prepareStatement() method when using a PreparedStatement. There are three scroll type constants that can be used.

```
 style="margin:0;width:955px;height:163px">Table  ResultSet Scroll Type Constants

Constant Description

ResultSet.TYPE_FORWARD_ONLY Default type, allows forward movement only.
ResultSet.TYPE_SCROLL_INSENSITIVE Allows forward and backward movement. Not sensitive to ResultSet updates.
ResultSet.TYPE_SCROLL_SENSITIVE Allows forward and backward movement. Sensitive to ResultSet updates.
```

You must also pass a ResultSet concurrency constant to advise whether the ResultSet is intended to be updatable.

The default is ResultSet.CONCUR_READ_ONLY, which means that the ResultSet is not updatable. The other concurrency type is ResultSet.CONCUR_UPDATABLE, which signifies an updatable ResultSet object.

In the solution to this recipe, a PreparedStatement object is used, and the code to create a PreparedStatement object that has the ability to generate a scrollable ResultSet looks like the following line:

```
pstmt = conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE,

ResultSet.CONCUR_READ_ONLY);
```

Once the PreparedStatement has been created as such, a scrollable ResultSet is returned. You can traverse in several directions using a scrollable ResultSet by calling the ResultSet methods indicating the direction you want to move or the placement that you want to be. The following line of code will retrieve the first record within the ResultSet:

```
ResultSet rs = pstmt.executeQuery();

rs.first();
```

The solution to this recipe demonstrates a few different scroll directions. Specifically, you can see that the ResultSet first(), next(), last(), and previous() methods are called in order to move to different positions within the ResultSet.

For a complete reference to the ResultSet object, see the online documentation that can be found at http://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html.

Scrollable ResultSet objects have a niche in application development. They are one of those niceties that are there when you need them, but they are also something that you might not need very often.

Creating an Updatable ResultSet

Problem

An application task has queried the database and obtained results. You have stored those results into a ResultSet object, and you want to update some of those values in the ResultSet and commit them back to the database.

Solution

Make your ResultSet object updatable, and then update the rows as needed while iterating through the results. The following example method demonstrates how to make ResultSet updatable and then how to update content within that ResultSet, eventually persisting it in the database:

```
 style="margin:0;height:361px;width:994px">private static void queryAndUpdateDbRecipes(String recipeNumber){
String sql = "SELECT ID, RECIPE_NUMBER, RECIPE_NAME, DESCRIPTION " + "FROM RECIPES " +
"WHERE RECIPE_NUMBER = ?";
ResultSet rs = null;
try (PreparedStatement pstmt =
conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.
CONCUR_UPDATABLE);){
pstmt.setString(1, recipeNumber);
rs = pstmt.executeQuery();
while(rs.next()){
String desc = rs.getString(4);
System.out.println("Updating row" + desc);
rs.updateString(4, desc + " -- More to come"); rs.updateRow();
}
} catch (SQLException ex) { ex.printStackTrace();
} finally {
if (rs != null){
try {
rs.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
This method could be called passing a String value containing a recipe number. Suppose that the recipe number "13-1" was passed to this met
hod; the following output would be the result:
Successfully connected
13-1: Connecting to a Database - DriverManager and DataSource Implementations
13-2: Querying a Database and Retrieving Results - Obtaining and Using Data from a DBMS
13-3: Handling SQL Exceptions - Using SQLException Updating rowDriverManager and DataSource Implementations
13-1: Connecting to a Database - DriverManager and DataSource Implementations - More to come
13-2: Querying a Database and Retrieving Results - Obtaining and Using Data from a DBMS
13-3: Handling SQL Exceptions - Using SQLException
```

How it works

Sometimes you need to update data as you are parsing it. Usually, this technique involves testing the values that are being returned from the database and updating them after comparison with another value.

The easiest way to do this is to make the ResultSet object updatable by passing the ResultSet.CONCUR_ UPDATABLE constant to the Connection object's createStatement() or prepareStatement() method. Doing so causes the Statement or PreparedStatement to produce an updatable ResultSet.

Note Some database JDBC drivers do not support updatable ResultSets. See the documentation of your JDBC driver for more information. This code was run using Oracle's ojdbc6.jar JDBC driver on Oracle database 11.2 release.

The format for creating a Statement that will produce an updatable ResultSet is to pass the ResultSet type as the first argument and the ResultSet concurrency as the second argument.

The scroll type must be TYPE_SCROLL_SENSITIVE to ensure that the ResultSet will be sensitive to any updates that are made. The following code demonstrates this technique by creating a Statement object that will produce a scrollable and updatable ResultSet object:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONC
UR_UPDATABLE);
```

The format for creating a PreparedStatement that will produce an updatable ResultSet is to pass the

SQL String as the first argument, the ResultSet type as the second argument, and the ResultSet concurrency as the third argument.

The solution to this recipe demonstrates this technique using the following line of code:

```
pstmt = conn.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,

ResultSet.CONCUR_UPDATABLE);
```

Both of the lines of code discussed in this section will produce scrollable and updatable ResultSet objects. Once you have obtained an updatable ResultSet, you can use it just like an ordinary ResultSet for fetching values that are retrieved from the database.

In addition, you can call one of the ResultSet object's updateXXX() methods to update any value within the ResultSet.

In the solution to this recipe, the updateString() method is called, passing the position of the value from the query as the first argument and the updated text as the second argument. In this case, the fourth element column listed in the SQL query will be updated.

```
rs.updateString(4, desc + " -- More to come");
```

Finally, to persist the values that you have changed, call the ResultSet updateRow() method, as seen in the solution to this recipe:

```
rs.updateRow();
```

Creating an updatable ResultSet is not something that you will need to do every day. In fact, you might never need to create an updatable ResultSet. However, for the cases in which such a strategy is needed, this technique can come in very handy.

Caching Data for Use When Disconnected

Problem

You want to work with data from a DBMS when you are in a disconnected state. That is, you are working on a device that is not connected to the database, and you still want to have the ability to work with a set of data as though you are connected. For instance, you are working with data on a portable device, and you are away from the office without a connection.

You want to have the ability to query, insert, update, and delete data, even though there is no connection. Once a connection becomes available, you want to have your device synchronize any database changes that have been made while you were disconnected.

Solution

Use a CachedRowSet object to store the data that you want to work with while offline. This will afford your application the ability to work with data as though it were connected to a database.

Once your connection is restored or you connect back to the database, synchronize the data that has been changed within the CachedRowSet with the database repository. The following example class demonstrates the use of a CachedRowSet. In this scenario, the main() method executes the example.

Suppose that there were no main() method, though, and that another application on a portable device were to invoke the methods of this class. Follow the code in the example and consider the possibility of working with the results that are stored within the CachedRowSet while not connected to the database.

For instance, suppose that you began some work in the office while connected to the network and are

For instance, suppose that you began some work in the office while connected to the network and are now outside of the office, where the network is spotty and you cannot maintain a constant connection to the database:

```
 style="margin:0;width:963px;height:249px">package org.java9recipes.blog13.recipe13_10;
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;
import javax.sql.rowset.spi.SyncProviderException;
import org.java9recipes.blog13.recipe13_01.CreateConnection;
public class CachedRowSetExample {
public static Connection conn = null;
public static CreateConnection createConn;
public static CachedRowSet crs = null;
public static void main(String[] args) {
boolean successFlag = false;
try {
createConn = new CreateConnection();
conn = createConn.getConnection();
Perform Scrollable Query queryWithRowSet();
Update the CachedRowSet updateData();
Synchronize changes syncWithDatabase();
} catch (java.sql.SQLException ex) { System.out.println(ex);
} finally {
if (conn != null) {
try {
conn.close();
} catch (SQLException ex) {
ex.printStackTrace();
}
}
}
}
/**
Call this method to synchronize the data that has been used in the
CachedRowSet with the database
*/
public static void syncWithDatabase() {
try {
crs.acceptChanges(conn);
} catch (SyncProviderException ex) {
If there is a conflict while synchronizing, this exception
will be thrown.
ex.printStackTrace();
} finally {
Clean up resources by closing CachedRowSet if (crs != null) {
try { crs.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
}
public static void queryWithRowSet() {
RowSetFactory factory;
try {
// Create a new RowSetFactory
factory = RowSetProvider.newFactory();
Create a CachedRowSet object using the factory crs = factory.createCachedRowSet();
Alternatively populate the CachedRowSet connection settings
crs.setUsername(createConn.getUsername());
crs.setPassword(createConn.getPassword());
crs.setUrl(createConn.getJdbcUrl());
Populate a query that will obtain the data that will be used crs.setCommand("select id, recipe_number, recipe_name, description from recipes")
;
```

```
Set key columns
int[] keys = {1};
crs.setKeyColumns(keys);
crs.execute(conn);
You can now work with the object contents in a disconnected state while (crs.next()) {
System.out.println(crs.getString(2) + ": " + crs.getString(3)
+ " - " + crs.getString(4));
}
} catch (SQLException ex) {
ex.printStackTrace();
}
}
public static boolean updateData() {
boolean returnValue = false;
try {
Move to the position before the first row in the result set crs.beforeFirst();
traverse result set
while (crs.next()) {
// If the recipe_num equals 11-2 then update
if (crs.getString("RECIPE_NUMBER").equals("13-2")) { System.out.println("updating recipe 13-2"); crs.updateString("description", "Subject to c
hange"); crs.updateRow();
}
}
returnValue = true;
Move to the position before the first row in the result set crs.beforeFirst();
traverse result set to see changes while (crs.next()) {
System.out.println(crs.getString(2) + ": " + crs.getString(3)
+ " - " + crs.getString(4));
}
} catch (SQLException ex) { returnValue = false; ex.printStackTrace();
}
return returnValue;
}
}
```

Running this example code will display output that looks similar to the following code, although the text might vary depending on the values in the database.

```
 style="margin:0;width:964px;height:125px">Successfully connected

Connecting to a Database - DriverManager and DataSource Implementations - More to Come
Querying a Database and Retrieving Results - Subject to Change
Handling SQL Exceptions - Using SQLException
Connecting to a Database - DriverManager and DataSource Implementations - More to Come
Querying a Database and Retrieving Results - Obtaining and Using Data from a DBMS
```

Handling SQL Exceptions - Using SQLException

How It Works

It is not possible to remain connected to the Internet all the time if you are working on a mobile device and traveling.

Nowadays there are devices that allow you to perform substantial work while you are on the go, even when you are not connected directly to a database. In such cases, solutions like the CachedRowSet object can come into play.

The CachedRowSet is the same as a regular ResultSet object, except it does not have to maintain a connection to a database in order to remain usable. You can query the database, obtain the results, and place them into a CachedRowSet object; and then work with them while not connected to the database. If changes are made to the data at any point, those changes can be synchronized with the database at a later time.

There are a couple of ways to create a CachedRowSet. The solution to this recipe uses a RowSetFactory to instantiate a CachedRowSet. However, you can also use the CachedRowSet default constructor to create a new instance. Doing so would look like the following line of code:

```
CachedRowSet crs = new CachedRowSetImpl();
```

Once instantiated, you need to set up a connection to the database. There are also a couple of ways to do this. Properties could be set for the connection that will be used, and the solution to this recipe demonstrates this technique within comments.

The following excerpt from the solution sets the connection properties using the CachedRowSet object's setUsername(), setPassword(), and setUrl() methods. Each of them accepts a String value, and in the example that String is obtained from the CreateConnection class:

Alternatively populate the CachedRowSet connection settings

```
style="margin:0;width:982px;height:77px">crs.setUsername(createConn.getUsername());
```

```
crs.setPassword(createConn.getPassword());

crs.setUrl(createConn.getJdbcUrl());
```

Another way to set up the connection is to wait until the query is executed and pass a Connection object to the executeQuery() method. This is the technique that is used in the solution to this recipe.

But before you can execute the query, it must be set using the setCommand() method, which accepts a String value. In this case, the String is the SQL query that you need to execute:

```
crs.setCommand("select id, recipe_number, recipe_name, description from recipes");
```

Next, if a CachedRowSet will be used for updates, the primary key values should be noted using the setKeys() method. This method accepts an int array that includes the positional indices of the key columns. These keys are used to identify unique columns. In this case, the first column listed in the query, ID, is the primary key:

```
int[] keys = {1};

crs.setKeyColumns(keys);
```

Finally, execute the query and populate the CachedRowSet using the execute() method. As mentioned previously, the execute() method optionally accepts a Connection object, which allows the CachedRowSet to obtain a database connection.

```
crs.execute(conn);
```

Once the query has been executed and the CachedRowSet has been populated, it can be used just like any other ResultSet. You can use it to fetch records forward and backward, or by specifying the absolute position of the row you'd like to retrieve. The solution to this recipe demonstrates only a couple of these fetching methods.

It is possible to insert and update rows within a CachedRowSet. To insert rows, use the moveToInsertRow() method to move to a new row position. Then populate a row by using the various methods [CachedRowSet, updateString(), updateInt(), and so on] that correspond to the data type of the column you are populating within the row.

Once you have populated each of the required columns within the row, call the insertRow() method, followed by the moveToCurrentRow() method. The following lines of code demonstrate inserting a record into the RECIPES table:

```
 style="margin:0;width:944px;height:90px">crs.moveToInsertRow();
crs.updateInt(1, sequenceValue); // obtain current sequence values with a prior query crs.updateStr
ing(2, “13-x”);
crs.updateString(3, “This is a new recipe title”);
```

```
crs.insertRow();
crs.moveToCurrentRow();
```

Updating rows is similar to using an updatable ResultSet. Simply update the values using the CachedRowSet object's methods [updateString(), updateInt(), and so on] that correspond to the data type of the column that you are updating within the row.

Once you have updated the column or columns within the row, call the updateRow() method. This technique is demonstrated in the solution to this recipe.

```
crs.updateString("description", "Subject to change"); crs.updateRow();
```

To propagate any updates or inserts to the database, the acceptChanges() method must be called. This method can accept an optional Connection argument in order to connect to the database.

Once called, all changes are flushed to the database. Unfortunately, because time might have elapsed since the data was last retrieved for the CachedRowSet, there could be conflicts.

If such a conflict arises, a SyncProviderException will be thrown. You can catch these exceptions and handle the conflicts manually using a SyncResolver object. However, resolving conflicts is out of the scope of this recipe, so for more information, see the online documentation that can be found at http://download.oracle.com/javase/tutorial/jdbc/basics/ cachedrowset.html.

CachedRowSet objects provide great flexibility for working with data, especially when you are using a device that is not always connected to the database. However, they can also be overkill in situations where you can simply use a standard ResultSet or even a scrollable ResultSet.

Joining RowSet Objects When Not Connected to the Data Source

Problem

You want to join two or more RowSets while not connected to a database. Perhaps your application is loaded on a mobile device that is not always connected to the database. In such a case, you are looking for a solution that will allow you to join the results of two or more queries.

Solution

Use a JoinRowSet to take data from two relational database tables and join them. The data from each table that will be joined should be fetched into a RowSet and then the JoinRowSet can be used to join each of those RowSet objects based on related elements contained within them. For instance, suppose that there were two related tables within a database.

One of the tables stores a list of authors, and the other table contains a list of blogs that are written by those authors. The two tables can be joined using SQL by the primary and foreign key relationship.

Note A primary key is a unique identifier within each record of a database table, and a foreign key is a referential constraint between two tables.

However, the application will not be connected to the database to make the JOIN query, so it must be done using a JoinRowSet. The following class listing demonstrates one strategy that can be used. In this scenario, the database table blog_AUTHOR and is set up as follows:

```
 style="margin:0;width:963px;height:319px">blog_AUTHOR(
id int primary key,
last varchar(30),
first varchar(30));
author_work(
id int primary key,
author_id int not null,
blog_number int not null,
blog_title varchar(100) not null,
constraint author_work_fk
foreign key(author_id) references blog_author(id));
blog(
id int primary key,
title varchar(150),
```

image varchar(150),
description clob);
The Java code to work with this table is as follows:
package org.java9recipes.blog13.recipe13_11;
import com.sun.rowset.JoinRowSetImpl;
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.rowset.CachedRowSet;
import javax.sql.rowset.JoinRowSet;
import javax.sql.rowset.RowSetFactory;
import javax.sql.rowset.RowSetProvider;
import org.java9recipes.blog13.recipe13_01.CreateConnection;
public class JoinRowSetExample {
public static Connection conn = null; public static CreateConnection createConn; public static CachedRowSet blogAuthors = null; public static CachedRowSet authorWork = null; public static JoinRowSet jrs = null;
public static void main(String[] args) {
boolean successFlag = false;
try {
createConn = new CreateConnection();
conn = createConn.getConnection();
Perform Scrollable Query queryblogAuthor(); queryAuthorWork();
joinRowQuery();
} catch (java.sql.SQLException ex) { System.out.println(ex);
} finally {
if (conn != null) {
try {
conn.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
if (blogAuthors != null) {
try {
blogAuthors.close();
} catch (SQLException ex) {
ex.printStackTrace();
}
}
if (authorWork != null) {
try {
authorWork.close();
} catch (SQLException ex) {
ex.printStackTrace();
}
}
if (jrs != null) {
try {
jrs.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
}
public static void queryblogAuthor() {
RowSetFactory factory;
try {
// Create a new RowSetFactory
factory = RowSetProvider.newFactory();
Create a CachedRowSet object using the factory blogAuthors = factory.createCachedRowSet();
Alternatively populate the CachedRowSet connection settings
crs.setUsername(createConn.getUsername());
crs.setPassword(createConn.getPassword());
crs.setUrl(createConn.getJdbcUrl());
Populate a query that will obtain the data that will be used blogAuthors.setCommand("SELECT ID, LASTNAME, FIRSTNAME FROM blog_AUTHOR");
blogAuthors.execute(conn);

You can now work with the object contents in a disconnected state while (blog authors next()) {

You can now work with the object contents in a disconnected state while (blog authors.next()) {

```
 style="margin:0;width:968px;height:319px">System.out.println(blogAuthors.getString(1) + ": " + blogAuthors.getString(2) + ", " + blogAuthors.g
etString(3));
}
} catch (SQLException ex) { ex.printStackTrace();
}
}
public static void queryAuthorWork() {
RowSetFactory factory;
try {
// Create a new RowSetFactory
factory = RowSetProvider.newFactory();
Create a CachedRowSet object using the factory authorWork = factory.createCachedRowSet();
Alternatively populate the CachedRowSet connection settings
crs.setUsername(createConn.getUsername());
crs.setPassword(createConn.getPassword());
crs.setUrl(createConn.getJdbcUrl());
Populate a query that will obtain the data that will be used authorWork.setCommand("SELECT AW.ID, AUTHOR_ID, B.TITLE FROM AUTHOR
_WORK AW, " +
"blog B " +
"WHERE B.ID = AW.blog_ID");
authorWork.execute(conn);
You can now work with the object contents in a disconnected state while (authorWork.next()) {
System.out.println(authorWork.getString(1) + ": " + authorWork.getString(2)
" - " + authorWork.getString(3));
}
} catch (SQLException ex) { ex.printStackTrace();
}
}
public static void joinRowQuery() {
try {
// Create JoinRowSet
jrs = new JoinRowSetImpl();
Add RowSet & Corresponding Keys jrs.addRowSet(blogAuthors, 1); jrs.addRowSet(authorWork, 2);
Alternatively use join-column name
jrs.addRowSet(authorWork, "AUTHOR_ID");
Traverse Results
while(jrs.next()){
System.out.println(jrs.getInt("ID") + ": " +
jrs.getString("TITLE") + " - " +
jrs.getString("FIRSTNAME") + " " +
jrs.getString("LASTNAME"));
}
} catch (SQLException ex) {
ex.printStackTrace();
}
}
}
```

Running this class will result in output that resembles the following:

```
 style="margin:0;width:975px;height:301px">Successfully connected
ThesisScientist, JOSH
DEA, CARL
BEATY, MARK
GUIME, FREDDY
JOHN, OCONNER
TESTER, JOE
TESTER, JOE
OCONNER, JOHN
```

100 - Java 8 Recipes
100 - Java 7 Thesis
100 - Java EE 7 Recipes
100 - Introducing Java EE 7
103 - Java 7 Thesis
101 - Java 7 Thesis
111 - Java 7 Thesis
102 - Java 7 Thesis
101 - Java FX 2.0 - Introduction by Example
Java 7 Thesis - JOHN OCONNER
Java 7 Thesis - FREDDY GUIME
Java 7 Thesis - MARK BEATY
Java FX 2.0 - Introduction by Example - CARL DEA
Java 7 Thesis - CARL DEA
Introducing Java EE 7 - JOSH ThesisScientist
Java EE 7 Recipes - JOSH ThesisScientist
Java 7 Thesis - JOSH ThesisScientist
Java 8 Recipes - JOSH ThesisScientist

How It Works

A JoinRowSet is a combination of two or more populated RowSet objects. It can be used to join two RowSet objects based on key/value relationships, just as if it were a SQL JOIN query.

In order to create a JoinRowSet, you must first populate two or more RowSet objects with related data, and then they can each be added to the JoinRowSet to create the combined result.

In the solution to this recipe, the tables that are queried are named blog_AUTHOR, blog, and AUTHOR_WORK. The blog_AUTHOR table contains a list of author names, while the AUTHOR_WORK table contains the list of blogs along with the corresponding AUTHOR_ID.

The blog table contains blog specifics. Following along with the main() method, first, the blog_AUTHOR table is queried, and its results are fetched into a CachedRowSet using the queryblogAuthor() method. For more details regarding the use of CachedRowSet objects.

Next, another CachedRowSet is populated with the results of querying the AUTHOR_WORK and log tables, as the queryAuthorblog() method is called. At this point, there are two populated CacheRowSet objects, and they can now be combined using a JoinRowSet.

In order to do so, each query must contain one or more columns that relate to the other table. In this case, the blog_AUTHOR.ID column relates to the AUTHOR_WORK. AUTHOR_ID column, so the RowSet objects must be joined on those column values.

The final method that is invoked within the main() is joinRowQuery(). This method is where all

the JoinRowSet work takes place. First, a new JoinRowSet is created by instantiating a JoinRowSetImpl() object: jrs = new JoinRowSetImpl();

Note You will receive a compile-time warning when using JoinRowSetImpl because it is an internal SUN proprietary API. However, the Oracle version is OracleJoinRowSet, which is not as versatile.

Next, the two CachedRowSet objects are added to the newly created JoinRowSet by calling its addRowSet() method. The addRowSet() method accepts a couple of arguments.

The first is the name of the RowSet object that you want to add to the JoinRowSet, and the second is an int value indicating the position within the CachedRowSet, which contains the key value that will be used to implement the join.

In the solution to this recipe, the first call to addRowSet() passes the blog authors CachedRowSet, along with the number 1 because the element in the first position of the blog authors CachedRowSet corresponds to the blog_AUTHOR.ID column.

The second call to addRowSet() passes the author work CachedRowSet, along with number 2 because the element in the second position of the author work CachedRowSet corresponds to the AUTHOR_WORK.AUTHOR_ID column.

```
 style="margin:0;width:966px;height:99px">Add RowSet & Corresponding Keys jrs.addRowSet(blogAuthors, 1); jrs.addRowSet(authorWork, 2)
;

Alternatively specify the join-column name jrs.addRowSet(authorWork, "AUTHOR_ID");
```

The JoinRowSet can now be used to fetch the results of the join, just as if it were a normal RowSet. When calling the corresponding methods [getString(), getInt(), and so on] of the JoinRowSet, pass the name of the database column corresponding to the data you want to store:

```
 style="margin:0;width:975px;height:90px">while(jrs.next()){
System.out.println(jrs.getInt("ID") + ": " +
jrs.getString("TITLE") + " - " +
jrs.getString("FIRSTNAME") + " " +
jrs.getString("LASTNAME"));
}
```

Although a JoinRowSet is not needed every day, it can be handy when performing work against two related sets of data. This holds true especially when the application is not connected to a database all the time, or if you are trying to use as few Connection objects as possible.

Filtering Data in a RowSet

Problem

Your application queries the database and returns a large number of rows. The number of rows within the cached ResultSet is too large for the user to work with at one time. You would like to limit the number of rows that are visible so that you can perform different activities with different sets of data that have been queried from the table.

Solution

Use a FilteredRowSet to query the database and store the contents. The FilteredRowSet can be configured to filter the results that are returned from the query so that the only contents visible are the rows that you want to see.

In the following example, a filter class is created that will later be used to filter the results that are returned from a database query.

The filter in the example is used to limit the number of rows that are visible based on an author's last name. The following class contains the implementation of the filter:

```
 style="margin:0;width:971px;height:316px">package org.java9recipes.blog13.recipe13_12;
import java.sql.SQLException;
import javax.sql.RowSet;
import javax.sql.rowset.Predicate;
public class AuthorFilter implements Predicate {
private String[] authors;
private String colName = null;
private int colNumber = -1;
public AuthorFilter(String[] authors, String colName) { this.authors = authors;
this.colNumber = -1;
this.colName = colName;
}
public AuthorFilter(String[] authors, int colNumber) { this.authors = authors;
this.colNumber = colNumber;
this.colName = null;
}
@Override
public boolean evaluate(Object value, String colName) {
if (colName.equalsIgnoreCase(this.colName)) { for (String author : this.authors) {
if (author.equalsIgnoreCase((String)value)) { return true;
}
}
}
return false;
}
@Override
public boolean evaluate(Object value, int colNumber) {
if (colNumber == this.colNumber) {
for (String author : this.authors) {
if (author.equalsIgnoreCase((String)value)) { return true;
}
}
}
return false;
}
@Override
public boolean evaluate(RowSet rs) {
if (rs == null)
return false;
try {
for (int i = 0; i < this.authors.length; i++) {
String authorLast = null;
if (this.colNumber > 0) {
authorLast = (String)rs.getObject(this.colNumber);
} else if (this.colName != null) {
authorLast = (String)rs.getObject(this.colName); } else {
return false;
}
if (authorLast.equalsIgnoreCase(authors[i])) { return true;
}
}
} catch (SQLException e) { return false;
}
return false;
}
}
```

The filter is used by a FilteredRowSet to limit the visible results from a query. As you will see, utilizing a FilteredRowSet provides the capability of filtering data in an object-oriented manner at the application level, rather than doing so at the SQL database level.

The benefit is that you can implement a series of filters and apply them to the same result set, returning the desired result. Using such an option eliminates the requirement to perform multiple database

queries returning different data sets.

The following class demonstrates how to implement a FilteredRowSet. The main() method calls a method that is appropriately named implementFilteredRowSet(), and it contains the code that is used to filter the results of a query on the blog_AUTHOR and AUTHOR_WORK tables so that only results from the authors with the last name of DEA and ThesisScientist are returned:

```
 style="margin:0;width:966px;height:333px">package org.java9recipes.blog13.recipe13_12;
import com.sun.rowset.FilteredRowSetImpl;
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.RowSet;
import javax.sql.rowset.FilteredRowSet;
import org.java9recipes.blog13.recipe13_01.CreateConnection;
public class FilteredRowSetExample {
public static Connection conn = null;
public static CreateConnection createConn;
public static FilteredRowSet frs = null;
public static void main(String[] args) {
boolean successFlag = false;
try {
createConn = new CreateConnection();
conn = createConn.getConnection();
implementFilteredRowSet();
} catch (java.sql.SQLException ex) { System.out.println(ex);
} finally {
if (conn != null) {
try {
conn.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
if (frs != null) {
try {
frs.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
}
public static void implementFilteredRowSet() {
String[] authorArray = {"DEA", "ThesisScientist"};
AuthorFilter authorFilter = new AuthorFilter(authorArray, 2);
try {
frs = new FilteredRowSetImpl();
frs.setCommand("SELECT TITLE, LASTNAME "
"FROM blog_AUTHOR BA, "
"AUTHOR_WORK AW, "
"blog B "
"WHERE AW.AUTHOR_ID = BA.ID "
"AND B.ID = AW.blog_ID");
frs.execute(conn);
System.out.println("Prior to adding filter:");
viewRowSet(frs);
System.out.println("Adding author filter:");
frs.beforeFirst();
frs.setFilter(authorFilter);
viewRowSet(frs);
} catch (SQLException e) { e.printStackTrace();
}
}
public static void viewRowSet(RowSet rs) {
try {
while (rs.next()) {
System.out.println(rs.getString(1) + ", "
```

```
System.out.println(rs.getString(1) + "  "
+ rs.getString(2));
}
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
```

The results of running this code would look similar to the following lines. Notice that only the rows of data corresponding to the authors listed in the filter are returned with the FilteredRowSet.

```
 style="margin:0;width:971px;height:202px">Successfully connected
Prior to adding filter:
Java 7 Thesis - ThesisScientist
Java 7 Thesis - BEATY
Java 7 Thesis - DEA
Java 7 Thesis - GUIME
Java 7 Thesis - OCONNER
Java EE 7 Recipes - ThesisScientist
Java FX 2.0 - Introduction by Example - DEA
Adding author filter:
Java 7 Thesis - ThesisScientist
Java 7 Thesis - DEA
Java EE 7 Recipes - ThesisScientist
```

How it works

Often, the results that are returned from a database query contain a large number of rows. As you probably know, too many rows can create issues when it comes to visually working with data.

It usually helps to limit the number of rows that are returned from a query by using a WHERE clause on a SQL statement so that only relevant data is returned.

However, if an application retrieves data into an in-memory RowSet and then needs to filter the data by various criteria without additional database requests, an approach other than a query needs to be used.

A FilteredRowSet can be used to filter data that is displayed within a populated RowSet so that it can be more manageable to work with.

There are two parts to working with a FilteredRowSet. First, a filter needs to be created which will be used to specify how the data should be filtered. The filter class should implement the Predicate interface.

There may be multiple constructors, each accepting a different set of arguments, and the filter may contain multiple evaluate() methods that each accept different arguments and contain different implementations.

The constructors should accept an array of contents that can be used to filter the RowSet. They should also accept a second argument, either the column name that the filter should be used against or the position of the column that the filter should be used against.

In the solution to this recipe, the filter class is named AuthorFilter, and it is used to filter data per an array of author names. Its constructors each accept an array containing the author names to filter, along with either the column name or position.

Each of the evaluate() methods has the task of determining whether a given row of data matches the specified filter; in this case, the author names that have been passed in via an array.

The first evaluate() method is called if a column name is passed to the filter rather than a position, and the second evaluate() method is called if a column position is passed.

The final evaluate() method accepts the RowSet itself, and it does the work of going through the data and returning a Boolean to indicate whether the corresponding column name/ position values match the filter data.

The second part of the FilteredRowSet implementation is the work of the FilteredRowSet. This can be seen within the implementFilteredRowSet() method of the FilteredRowSetExample class. The FilteredRowSet will actually use the filter class that you've written to determine which rows to display.

You can see that the array of values that will be passed to the filter class is the first declaration within the method. The second declaration is the instantiation of the filter class AuthorFilter. Of course, the array of filter values and the column position that corresponds to the filter values is passed into the filter constructor.

```
 style="margin:0;width:966px;height:86px">String[] authorArray = {"DEA", "ThesisScientist"};

// Creates a filter using the array of authors

AuthorFilter authorFilter = new AuthorFilter(authorArray, 2);
```

To instantiate a FilteredRowSet, create a new instance of the FilteredRowSetImpl class. After it is instantiated, simply set the SQL query that will be used to obtain the results using the setCommand() method and then execute it by calling the executeQuery() method.

```
Instantiate a new FilteredRowSet frs = new FilteredRowSetImpl();
```

Set the query

```
 style="margin:0;width:962px;height:123px">frs.setCommand("SELECT TITLE, LASTNAME "
"FROM blog_AUTHOR BA, "
"AUTHOR_WORK AW, "
"blog B "
"WHERE AW.AUTHOR_ID = BA.ID "
"AND B.ID = AW.blog_ID");
Execute the query
frs.execute(conn);
```

Note You will receive a compile-time warning when using FilteredRowSetImpl because it is an older internal proprietary API produced by Sun Microsystems.

Notice that the filter has not yet been applied. Actually, at this point what you have is a scrollable RowSet that is populated with all the results from the query. The example displays those results before applying the filter.

To apply the filter, use the setFilter() method, passing the filter as an argument. Once that has been done, the FilteredResultSet will display only those rows that match the criteria specified by the filter.

Again, the FilteredRowSet technique has its place, especially when you are working with an application that might not always be connected to a database. It is a powerful tool to use for filtering data, working with it, and then applying different filters and working on the new results. It is similar to applying WHERE clauses to a query without querying the database.

Problem

The application that you are developing requires the storage of Strings of text that can include an unlimited number of characters.

Solution

It is best to use a character large object (CLOB) data type to store text when the size of the Strings that need to be stored is very large. The database diagram for the RECIPE_TEXT table is as follows:

```
 style="margin:0;width:937px;height:321px">RECIPE_TEXT (
id int primary key,
recipe_id int not null,
text clob,
constraint recipe_text_fk
```

```
foreign key (recipe_id)
references recipes(id))
```

The code in the following example demonstrates how to load a CLOB into the database and how to query it:

```
package org.java9recipes.blog13.recipe13_13;
import java.sql.Clob;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import org.java9recipes.blog13.recipe13_01.CreateConnection;
public class LobExamples {
public static Connection conn = null;
public static CreateConnection createConn;
public static void main(String[] args) {
boolean successFlag = false;
try {
createConn = new CreateConnection();
conn = createConn.getConnection();
loadClob();
readClob();
} catch (java.sql.SQLException ex) { System.out.println(ex);
} finally {
if (conn != null) {
try {
conn.close();
} catch (SQLException ex) { ex.printStackTrace();
}
}
}
}
public static void loadClob() {
Clob textClob = null;
String sql = "INSERT INTO RECIPE_TEXT VALUES("
"next value for recipe_text_seq, "
"(select id from recipes where recipe_number = '13-1'), "
"?)";
try (PreparedStatement pstmt = conn.prepareStatement(sql);) { textClob = conn.createClob();
textClob.setString(1, "This will be the recipe text in clob format");
obtain the sequence number in real world
set the clob value
pstmt.setClob(1, textClob);
pstmt.executeUpdate();
} catch (SQLException ex) {
ex.printStackTrace();
}
}
public static void readClob() {
String qry = "select text from recipe_text";
Clob theClob = null;
```

```
Clob theClob = null;
try(PreparedStatement pstmt = conn.prepareStatement(qry); ResultSet rs = pstmt.executeQuery();) {
while (rs.next()) {
theClob = rs.getClob(1);
System.out.println("Clob length: " + theClob.length()); System.out.println(theClob.toString());
}
System.out.println(theClob.toString());
} catch (SQLException ex) {
ex.printStackTrace();
}
}
}
```

How It Works

If your application requires the storage of String values, you need to know how large those Strings might possibly become. Most databases have an upper boundary when it comes to the storage size of VARCHAR fields.

For instance, the Oracle database has an upper boundary of 2,000 characters and anything exceeding that length will be cut off. If you have large amounts of text that need to be stored, use a CLOB field in the database.

A CLOB is handled a bit differently from a String within Java code. In fact, it is actually a bit odd to work

A CLOB is handled a bit differently from a String within Java code. In fact, it is actually a bit odd to work with the first couple of times you use it because you have to create a CLOB from a Connection.

Note In reality, CLOBs and BLOBs (binary large objects) are not stored in the Oracle table where they are defined. Instead, a large object (LOB) locator is stored in the table column.

Oracle might place the CLOB in a separate file on the database server. When Java creates the Clob object, it can be used to hold data for an update to a specific LOB location in the database or to retrieve the data from a specific LOB location within the database.

Let's take a look at the loadClob() method contained in the solution to this recipe. As you can see, a Clob object is created using the Connection createClob() method.

Once the Clob has been created, you set its contents using the setString() method by passing the position indicating where to place the String and the String of text itself:

```
textClob = conn.createClob();

textClob.setString(1, "This will be the recipe text in clob format");
```

Once you have created and populated the Clob, you simply pass it to the database using the PreparedStatement setClob() method. In the case of this example, the PreparedStatement performs a database insert into the RECIPE_TEXT table by calling the executeUpdate() method as usual.

Querying a Clob is fairly straightforward as well. As you can see in the readClob() method that is contained within the solution to this recipe, a PreparedStatement query is set up and the results are retrieved into a ResultSet.

```
The only difference between using a Clob and a String is that you must load the Clob into a Clob type.

Note Calling the Clob getString() object.

 Therefore, calling the Clob object's in the Clob.

the method will pass a funny-looking String of text that denotes a Clob getAsciiStream() method will return the actual data that is stored
```

Although Clobs are fairly easy to use, they take a couple of extra steps to prepare. It is best to plan your applications accordingly and try to estimate whether the database fields you are using might need to be CLOBs due to size restrictions. Proper planning will prevent you from going back and changing the standard String-based code to work with Clobs later.

Invoking Stored Procedures

Problem

Some logic that is required for your application is written as a database-stored procedure. You require the ability to invoke the stored procedure from within your application.

Solution

The following block of code shows the PL/SQL that is required to create the stored procedure that will be called by Java. The functionality of this stored procedure is very minor; it simply accepts a value and assigns that value to an OUT parameter so that the program can display it:

```
create or replace procedure dummy_proc (text IN VARCHAR2, msg OUT VARCHAR2) as

begin

Do something, in this case the IN parameter value is assigned to the OUT parameter msg :=text;

end;
```

The CallableStatement in the following code executes this stored procedure that is contained within the database, passing the necessary parameters. The results of the OUT parameter are then displayed back to the user.

```
 style="margin:0;width:965px;height:125px">try(CallableStatement cs = conn.prepareCall("{call DUMMY_PROC(?,?)}");) { cs.setString(1, "This is a test");
cs.registerOutParameter(2, Types.VARCHAR);
```

```
cs.executeQuery();
System.out.println(cs.getString(2));
} catch (SQLException ex){
ex.printStackTrace();
}
```

Running the example class for this recipe will display the following output, which is the same as the input. This is because the DUMMY_PROC procedure simply assigns the contents if the IN parameter to the OUT parameter.

Successfully connected

This is a test

How It Works

It is not uncommon for an application to use database-stored procedures for logic that can be executed directly within the database.

In order to call a database-stored procedure from Java, you must create a CallableStatement object, rather than use a PreparedStatement. In the solution to this recipe, a CallableStatement invokes a stored procedure named DUMMY_PROC.

The syntax for instantiating the CallableStatement is similar to that of using a PreparedStatement. Use the Connection object's prepareCall() method, passing the call to the stored procedure. The stored procedure call must be enclosed in curly braces {} or the application will throw an exception.

```
cs = conn.prepareCall("{call DUMMY_PROC(?,?)}");
```

Once the CallableStatement has been instantiated, it can be used just like a PreparedStatement for setting the values of parameters. However, if a parameter is registered within the database-stored procedure as an OUT parameter, you must call a special method, registerOutParameter(), passing the parameter position and database type of the OUT parameter that you want to register.

In the solution to this recipe, the OUT parameter is in the second position and it has a VARCHAR type.

```
cs.registerOutParameter(2, Types.VARCHAR);
```

To execute the stored procedure, call the executeQuery() method on the CallableStatement. Once this has been done, you can see the value of the OUT parameter by making a call to the CallableStatement getXXX() method that corresponds to the data type:

```
System.out.println(cs.getString(2));
```

A NOTE REGARDING STORED FUNCTIONS

Calling a stored database function is essentially the same as calling a stored procedure. However, the syntax to prepareCall() is slightly modified. To call a stored function, change the call within the curly

syntax to prepareCall() is slightly modified. To call a stored function, change the call within the curly braces to entail a returned value using a? character.

For instance, suppose that a function named DUMMY_FUNC accepted one parameter and returned a value. The following code would be used to make the call and return the value:

```
cs = conn.prepareCall("{? = call DUMMY_FUNC(?)}"); cs.registerOutParameter(1, Types.VARCHAR); cs.setString(2, "This is a test"); cs.execute();
```

A call to cs.getString(1) would then retrieve the returned value.

Obtaining Dates for Database Use

Problem: You want to convert a LocalDate properly, in order to insert it into a database record.

Solution

Utilize the static java.sql.Date.valueOf(LocalDate) method to convert a LocalDate object to a java.sql.Date object, which can be utilized by JDBC for insertion or querying of the database. In the following example, the current date is inserted into a database column of type Date.

```
 style="margin:0;width:970px;height:234px">private static void insertRecord(
String title,
String publisher) {
String sql = "INSERT INTO PUBLICATION VALUES("
"NEXT VALUE FOR PUBLICATION_SEQ, ?,?,?,?)"; LocalDate pubDate = LocalDate.now();
try (Connection conn = createConn.getConnection(); PreparedStatement pstmt = conn.prepareStatement(sql);) {
pstmt.setInt(1, 100);
pstmt.setString(2, title);
pstmt.setDate(3, java.sql.Date.valueOf(pubDate));
pstmt.setString(4, publisher);
pstmt.executeUpdate();
System.out.println("Record successfully inserted.");
} catch (SQLException ex) { ex.printStackTrace();
}
}
```

How It Works

In Java 8, the new Date-Time API is the preferred API for working with dates and times. Therefore, when working with date values and databases, the JDBC API must convert between SQL dates and new Date-Time LocalDate objects.

The solution to this recipe demonstrates that to obtain an instance of java.sql.Date from a LocalDate object, you simply invoke the static java.sql.Date.valueOf() method, passing the pertinent LocalDate object.

Closing Resources Automatically

Problem: Rather than manually opening and closing resources with each database call, you would prefer to have the application handle such boilerplate code for you.

Solution

Use the try-with-resources syntax to automatically close the resources that you open. The following block of code uses this tactic to automatically close the Connection, Statement, and ResultSet resources when it is finished using them:

```
 style="margin:0;width:960px;height:164px">String qry = "select recipe_number, recipename, description from recipes";
try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement(); ResultSet rs = stmt.executeQuery(qry);) {
while (rs.next()) {
String recipe = rs.getString("RECIPE_NUMBER"); String name = rs.getString("RECIPE_NAME"); String desc = rs.getString("DESCRIPTION");
System.out.println(recipe + "\t" + name + "\t" + desc);
}
} catch (SQLException e) { e.printStackTrace();
}
```

The resulting output from running this code should look similar to the following:

```
 style="margin:0;width:968px;height:85px">Successfully connected
13-1 Connecting to a Database DriverManager and DataSource Implementations - More to Come
13-2 Querying a Database and Retrieving Results Subject to Change
13-3 Handling SQL Exceptions Using SQLException
```

How It Works

Handling JDBC resources has always been a pain in the neck. There is a lot of boilerplate code that is required for closing resources when they are no longer needed.

Since the release of Java SE 7, this has not been the case. Java 7 introduced automatic resource management using try-with-resources. Through the use of this technique, the developer no longer needs to close each resource manually, which is a change that can cut down on many lines of code.

In order to use this technique, you must instantiate all the resources for which you want to have automatic handling enabled within a set of parentheses after a try clause. In the solution to this recipe, the resources that are declared are Connection, Statement, and ResultSet.

```
try (Connection conn = createConn.getConnection(); Statement stmt = conn.createStatement(); Res
ultSet rs = stmt.executeQuery(qry);) {
```

Once those resources are out of scope, they are automatically closed. This means there is no longer a requirement to code a finally block to ensure that resources are closed. The automatic resource handling is available not only to database work but to any resource that complies with the new java. lang.Autocloseable API.

Other operations such as file I/O adhere to the new API as well. There is a single close() method within java.lang.Autoclosable that manages the closing of the resource. Classes that implement the java.io. The closeable interface can adhere to the API.