3.5. Scrollable Result Sets

The single most visible addition to the JDBC API in its 2.0 specification is support for scrollable result sets. When the JDBC specification was first finalized, the specification contributors engaged in serious debate as to whether or not result sets should be scrollable. Those against scrollable result sets—and I was one of them—argued that they were antithetical to object-oriented programming and that they violated the rule that complex functionality should not encumber the most commonly used classes. In addition, requiring all driver vendors to implement scrollable result sets could adversely impact the performance of more mundane result set operations for some database engines. Scrollable result sets, on the other hand, are common in database vendor APIs, and the database vendors thus believed they should be present in JDBC.

3.5.1. Result Set Types

Using scrollable result sets starts with the way in which you create statements. Earlier in the chapter, you learned to create a statement using the `createStatement( )` method. The `Connection` class actually has two versions of `createStatement()`—the zero parameter version you have used so far and a two parameter version that supports the creation of `Statement` instances that generate scrollable `ResultSet` objects. The default call translates to the following call:

```
conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                     ResultSet.CONCUR_READ_ONLY);
```

The first argument is the result set type. The value `ResultSet.TYPE_FORWARD_ONLY` indicates that any `ResultSet` generated by the `Statement` returned from `createStatement()` only moves forward (the JDBC 1.x behavior). The second argument is the result set concurrency. The value `ResultSet.CONCUR_READ_ONLY` specifies that each row from a `ResultSet` is read-only. As you will see in the next chapter, rows from a `ResultSet` can be modified in place if the concurrency specified in the `createStatement()` call allows it.

JDBC defines three types of result sets: `TYPE_FORWARD_ONLY` , `TYPE_SCROLL_SENSITIVE` , and `TYPE_SCROLL_INSENSITIVE` . `TYPE_FORWARD_ONLY` is the only type that is not scrollable. The other two types are distinguished by how they reflect changes made to them. A `TYPE_SCROLL_INSENSITIVE ResultSet` is unaware of in-place edits made to modifiable instances. `TYPE_SCROLL_SENSITIVE`, on the other hand, means that you can see changes made to the results if you scroll back to the modified row at a later time. You should keep in mind that this distinction remains only while you leave the result set open. If you close a `TYPE_SCROLL_INSENSITIVE ResultSet` and then requery, your new `ResultSet` reflects any changes made to the original.

3.5.2. Result Set Navigation

When `ResultSet` is first created, it is considered to be positioned before the first row. Positioning methods such as `next()` point a `ResultSet` to actual rows. Your first call to `next( )`, for example, positions the cursor on the first row. Subsequent calls to `next()` move the `ResultSet` ahead one row at a time. With a scrollable `ResultSet`, however, a call to `next()` is not the only way to position a result set.

The method `previous( )` works in an almost identical fashion to `next( )`. While `next()` moves one row forward, `previous()` moves one row backward. If it moves back beyond the first row, it returns `false`. Otherwise, it returns `true`. Because a `ResultSet` is initially positioned before the first row, you need to move the `ResultSet` using some other method before you can call `previous()`. Example 3.5 shows how `previous( )`, after a call to `afterLast( )`, can be used to move backward through a `ResultSet`.

Example3.5. Moving Backward Through a Result Set

```
import java.sql.*;
import java.util.*;

public class ReverseSelect {
  public static void main(String argv[]) {
```

```
    Connection con = null;

    try {
      String url = "jdbc:msql://carthage.imaginary.com/ora";
      String driver = "com.imaginary.sql.msql.MsqlDriver";
      Properties p = new Properties( );
      Statement stmt;
      ResultSet rs;

      p.put("user", "borg");
      Class.forName(driver).newInstance( );
      con = DriverManager.getConnection(url, "borg", "");
      stmt =
      con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                          ResultSet.CONCUR_READ_ONLY);
       rs = stmt.executeQuery("SELECT * from test ORDER BY test_id");
      // as a new ResultSet, rs is currently positioned
      // before the first row
      System.out.println("Got results:");
      // position rs after the last row
      rs.afterLast( );
      while(rs.previous( )) {
        int a;
        String str;

        a = rs.getInt("test_id");
        if( rs.wasNull( ) ) {
          a = -1;
        }
        str = rs.getString("test_val");
        if( rs.wasNull( ) ) {
          str = null;
        }
        System.out.print("\ttest_id= " + a);
        System.out.println("/str= '" + str + "'");
      }
      System.out.println("Done.");
    }
    catch( Exception e ) {
      e.printStackTrace( );
    }
    finally {
      if( con != null ) {
        try { con.close( ); }
        catch( SQLException e ) { e.printStackTrace( ); }
      }
    }
  }
}
```

This example is really no different than the `SELECT` example from earlier in the chapter. This example simply pulls the results out of the database backward.

Along with `afterLast()` and `previous( )`, JDBC 2.0 provides new methods to navigate around rows in result sets: `beforeFirst( )`, `first( )`, `last( )`, `absolute( )`, and `relative( )`. Except for `absolute()` and `relative()`, the names of the methods say exactly what they do. The `beforeFirst()` method positions the `ResultSet` before the first row—its initial state—and the `first()` and `last()` methods position the `ResultSet` on the first and last rows, respectively.

The methods `absolute()` and `relative( )` each take integer arguments. For `absolute( )`, the argument specifies a row to navigate to. A call to `absolute(5)` moves the `ResultSet` to row 5—unless there are four or fewer rows in the `ResultSet`. If the specified row is beyond the last row in the `ResultSet`, the `ResultSet` is positioned after the last row. A call to `absolute()` with a row number beyond the last row is therefore identical to a call to `afterLast()`.

You can also pass negative numbers to `absolute()`. A negative number specifies absolute navigation backwards from the last row. Where `absolute(1)` is identical to `first()`, `absolute(-1)` is identical to `last()`. Similarly, `absolute(-3)` is the third to last row in the `ResultSet`. If there are fewer than three rows in the `ResultSet`, then `ResultSet` is positioned before the first row.

The `relative()` method handles relative navigation through a `ResultSet`. In other words, it tells the `ResultSet` how many rows to move forward or backward. A value of 1 behaves just like `next( )` and a value of -1 just like `previous()`.

3.5.3. Determining Where You Are

It is hard to get where you want to go if you don't know where you are. Navigating through scrollable result sets is no different. Of course, you do know where a `ResultSet` is positioned when you first create it. While processing the `ResultSet`, however, you may find that you don't know where the `ResultSet` is positioned. The `ResultSet` class fortunately provides these methods to let you check the current `ResultSet` position: `isFirst()`, `isLast()`, `isBeforeFirst()`, `isAfterLast()`, and `getRow( )`. All except `getRow()` return booleans; `getRow()` returns the current row number as an integer.

3.5.4. Helping Your Driver with Scrollable Result Sets

One of the drawbacks of scrollable result sets is that they can be inefficient for some database engines to implement. Specifically, a JDBC driver needs to process rows in an ad hoc fashion, rather than a single, unidirectional fashion. Before scrollable result sets, a JDBC driver can intelligently fetch rows from the database in a just-in-time fashion. While you are getting the first row, it is off retrieving the second and third rows.

JDBC 2.0 gives you the power to help your driver efficiently handle scrollable result sets—to help it avoid having to be ready for random navigation. The method `setFetchDirection( )` lets you tell the driver the direction in which you intend to process a result set. It accepts the values `ResultSet.FETCH_FORWARD`, `ResultSet.FETCH_REVERSE`, or `ResultSet.FETCH_UNKNOWN`. Calling this method may mean absolutely nothing. If, however, the driver can take advantage of knowing the direction in which you intend to process results, then calling this method should improve your performance.

The `setFetchSize( )` method is another method you can use to help the driver be more efficient. The default fetch size is and ignored; the driver makes its best guess as to how many rows to prefetch. If, for example, you know you only want to grab the first row from a result set and no more, you can specify a fetch size of 1. If the driver can optimize based on this information, it can make sure it is not simply returning all the rows when you will only handle 1. By setting the value to 1, however, you do not limit yourself; this value is just a hint to the driver.

When writing a client that intends to use a subset of information in a result set at any given point, you should definitely take advantage of the ability to provide these hints. By indicating to a driver that uses this information that you intend to display only 50 rows at a time in a Swing `JTable`, you prevent it from sending all 1,000 rows of a result set to a client who will likely see at most 100 rows.