Advertisement: Support JavaWorld, click here!

Design and performance improvements with JDBC 4.0

Effectively utilize JDBC'S features to get desired results with less code

Summary
Java Database Connectivity (JDBC) 4.0 is ready for release by mid 2006 as a part of Java Standard Edition 6.0. How can you leverage the new specification to improve the design and performance of database access and interactions in Java applications? This article discusses the new features of JDBC 4.0, illustrates its solutions to some existing problems, and presents its improvements in design and performance through examples. (*2,100 words*; **May 1, 2006**)

By Shashank Tiwari

Java Database Connectivity (JDBC), which has existed from the first public version of the core Java language, has evolved significantly over the last 10 years. In its current version, 4.0, which will be packaged with Java Standard Edition 6.0 (Java SE is Sun's new name for J2SE), it shows significant improvements in design and provides a richer API, with focus on ease of development and improvement in productivity.

This article discusses some of the important changes in the JDBC specification that either improve the design or facilitate better performance. The article does not enlist or survey every single change incorporated as a part of Java Specification Request 221, the JDBC 4.0 initiative.

After reading this article, you should be ready to leverage the new features in your next set of applications.

Annotations and the generic DataSet

I assume you are already aware of annotations and generics, which were introduced in Java with J2SE 5.0. JDBC 4.0 introduces annotations and the generic `DataSet`. This change aims to simplify execution of SQL queries (in scenarios that return a single result set) and SQL DML (data manipulation language) statements (that return either a row count or nothing).

The new API defines a set of `Query` and `DataSet` interfaces. The `Query` interface defines a set of

methods decorated with the JDBC annotations. These decorated methods describe the SQL select and update statements, and specify how the result set should be bound to a `DataSet`. The `DataSet` interface is a parameterized type, as defined by generics. The `DataSet` interface provides a type-safe definition for the result set data.

All `Query` interfaces inherit from the `BaseQuery` interface. A concrete implementation of the interface can be instantiated using either the `Connection.createQueryObject()` or `DataSource.createQueryObject()` methods and passing a `Query` interface type as its parameter.

A `DataSet` interface inherits from `java.util.List`. A data class describing the columns of the result set data, returned by an annotated method of the `Query` interface, is its parameter type. A `DataSet` can be manipulated and operated upon both in a connected and disconnected mode. Thus, the `DataSet` is implemented either as a `ResultSet` or a `CachedRowSet`, depending on its operating mode: connected or disconnected. `DataSet`, being a sub-interface of the `java.util.List`, allows access of its data rows with the Iterator pattern, using the `java.util.Iterator` interface.

The data class or the user-defined class, which is a parameter type of the `DataSet` interface, can be specified in two ways: as a structure or as a JavaBeans object. Either method achieves the goal of binding result set data columns to user-defined class definitions, but the JavaBeans component model is more elegant and facilitates object definition reuse within other frameworks that support the JavaBeans model.

Listing 1 illustrates code snippets for a simple example to show how the new API is used to create and run SQL queries, define result set data using a user-defined class, and bind the returned result set to the user-defined specifications.

Listing 1. Employee user-defined type and employeeQueries

```
pubic class Employee {
   private int employeeId;
   private String firstName;
   private String lastName;

   public int getEmployeeId() {
      return employeeId;
   }

   public setEmployeeId(int employeeId) {
      this.employeeId = employeeId;
   }

   public String getFirstName() {
      return firstName;
   }

   public setFirstName(String firstName) {
      this.firstName = firstName;
   }
```

```
   pubic String lastName() {
      return lastName;
   }

   public setLastName(String lastName) {
      this.lastName = lastName;
   }
}


interface EmployeeQueries extends BaseQuery {
   @Select (sql="SELECT employeeId, firstName, lastName FROM
employee")
   DataSet<Employee> getAllEmployees ();

   @Update (sql="delete from employee")
   int deleteAllEmployees ();
}


Connection con = ...

EmployeeQueries empQueries = con.createQueryObject
(EmployeeQueries.class);

DataSet<Employee> empData = empQueries.getAllEmployees ();
```

Exception-handling enhancements

The exception-handling functionality in the JDBC API prior to version 4.0 is limited and often insufficient. `SQLException` is thrown for all types of errors. There is no classification of exceptions, and no hierarchy defines them. The only way to get some meaningful information is to retrieve and analyze the `SQLState` value. `SQLState` values and their corresponding meanings change from datasource to datasource; hence, getting to the root of the problem and efficiently handling exceptions proves to be a tedious task.

JDBC 4.0 has enhanced exception-handling capability and alleviates some of the mentioned problems. The key changes are as follows:

- Classification of `SQLException` into transient and non-transient types
- Support for chained exceptions
- Implementation of the `Iterable` interface

The `SQLTransientException` is thrown where a previously failed operation may succeed on retrial. The `SQLNonTransientException` is thrown where retrial will not lead to a successful operation unless the cause of the `SQLException` is corrected.

Figure 1 illustrates the subclasses of `SQLTransientException` and `SQLNonTransientException`.

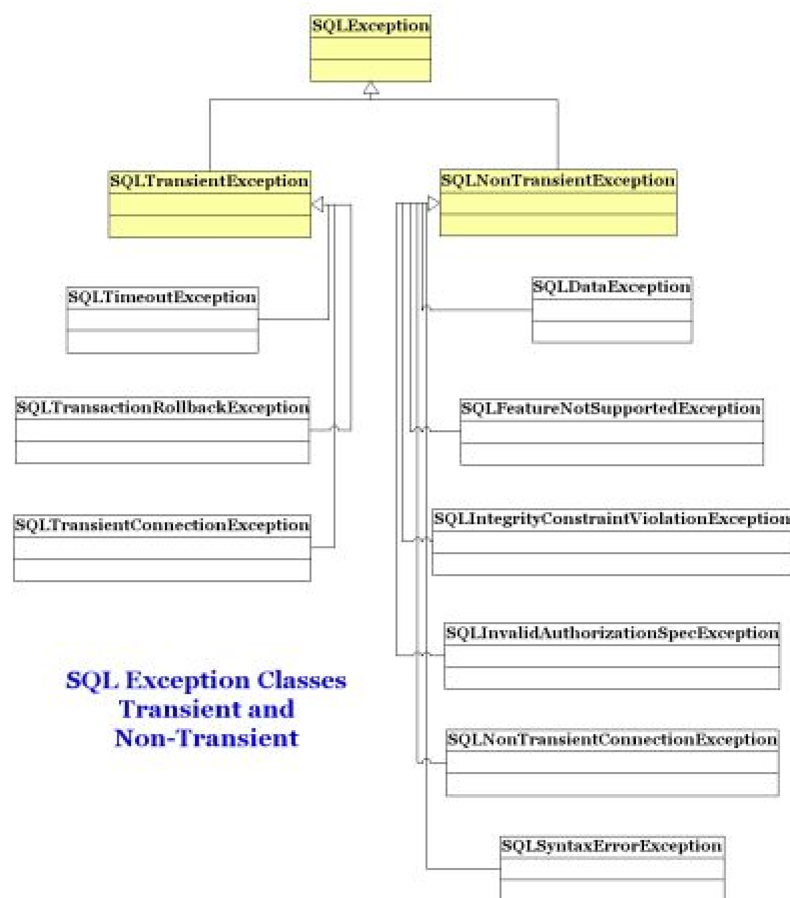


Figure 1. SQL exception classes: Transient and non-transient

Support for chained exceptions are now included. New constructors add extra parameters to capture possible causes for the exception. Multiple `SQLExceptions` could be iterated over in a loop, and `getCause()` could be called to determine the exception's possible cause. The `getCause()` method can return non-`SQLExceptions` if they are the underlying cause of the exceptions.

The `SQLException` class now implements the `Iterable` interface and supports the J2SE 5.0 *for each loop* for easier and more elegant looping.

Listing 2 depicts the usage of the new for-each-loop construct:

Listing 2. For each loop

```
catch(SQLException ex) {
   for(Throwable t : ex) {
      System.out.println("exception:" + t);
   }
}
```

SQL/XML

A large amount of data now exists in the XML format. Databases have extended support for the

XML data type by defining a standard XML type in the SQL 2003 specification. Most database vendors have an implementation of the XML data type in their new releases. With the inclusion of such a type, an XML dataset or document could be one of the fields or column values in a row of a database table. Prior to JDBC 4.0, perhaps the best way to manipulate such data within the JDBC framework is to use proprietary extensions from the driver vendors or access it as a `CLOB` type.

JDBC 4.0 now defines `SQLXML` as the Java data type that maps the database SQL XML type. The API supports processing of an XML type as a string or as a StAX stream. Streaming API for XML, which for Java has been adopted via JSR 173, is based on the Iterator pattern, as opposed to the Simple API for XML Processing (SAX), which is based on the Observer pattern.

Invoking the `Connection` object's `createSQLXML()` method can create a `SQLXML` object. This is an empty object, so the data can be attached to it by either using the `setString()` method or by associating an XML stream using the `createXMLStreamWriter()` method with the object. Similarly, XML data can be retrieved from a `SQLXML` object using `getString()` or associating an XML stream using `createXMLStreamReader()` with the object.

The `ResultSet`, the `PreparedStatement`, and the `CallableStatement` interfaces have `getSQLXML()` methods for retrieving a `SQLXML` data type. `PreparedStatement` and `CallableStatement` also have `setSQLXML()` methods to add `SQLXML` objects as parameters.

The `SQLXML` resources can be released by calling their `free()` methods, which might prove pertinent where the objects are valid in long-running transactions. `DatabaseMetaData`'s `getTypeInfo()` method can be called on a datasource to check if the database supports the `SQLXML` data type, since this method returns all the data types it supports.

Connections and Statements

The `Connection` interface definitions have been enhanced to analyze connection state and usage to facilitate efficiency.

Sometimes database connections are unusable though they may not necessarily be closed and garbage collected. In such situations, the database appears slow and unresponsive. In most of these circumstances, reinitializing the connections is perhaps the only way to resolve the problem. When using the JDBC API prior to version 4.0, there is no way to distinguish between a stale connection and a closed connection. The new API adds an `isValid()` method to the `Connection` interface to query if the connection is still valid.

Also, database connections are often shared among clients, and sometimes some clients tend to use more resources than others, which can lead to starvation-like situations. The `Connection` interface defines a `setClientInfo()` method to define client-specific properties, which could be utilized to analyze and monitor resource utilization by the clients.

RowId

The `RowId` in many databases is a unique way to identify a row in a table. Queries using `RowId` in the search criteria are often the fastest way to retrieve data, especially true in the case of the Oracle and DB2 databases. Since `java.sql.RowId` is now a built-in type in Java, you could utilize the performance benefits associated with its usage. `RowIds` are most useful in identifying unique and specific rows when duplicate data exists and some rows are identical. However, it is important to understand that `RowIds` often are unique only for a table and not for the entire database; they may change and are not supported by all databases. `RowIds` are typically not portable across datasources and thus should be used with caution when working with multiple datasources.

A RowId is valid for the lifetime defined by the datasource and as long as the row is not deleted. The DatabaseMetadata.getRowIdLifetime() method is called to determine the RowId's lifetime. The return type is an enumeration type as summarized in the table below.

RowIdLifetime **enum type**	**Definition**
ROWID_UNSUPPORTED	Datasource does not support RowId type
ROWID_VALID_OTHER	Implementation-dependent lifetime
ROWID_VALID_TRANSACTION	Lifetime is at least the containing transaction
ROWID_VALID_SESSION	Lifetime is at least the containing session
ROWID_VALID_FOREVER	Unlimited lifetime

The ROWID_VALID_TRANSACTION, ROWID_VALID_SESSION, and ROWID_VALID_FOREVER definitions are true as long as the row is not deleted. It is important to understand that a new RowId is assigned if a row is deleted and reinserted, which sometimes could happen transparently at the datasource. As an example, in Oracle, if the "enable row movement" clause is set on a partitioned table and an update of the partition key causes the row to move from one partition to another, the RowId will change. Even without the "enable row movement" flag with the "alter table table_name" move, the RowId could change.

Both the ResultSet and CallableStatement interfaces have been updated to include a method called getRowID(), which returns a javax.sql.RowId type.

Listing 3 shows how RowId could be retrieved from a ResultSet and from a CallableStatement.

Listing 3. Get RowId

```
//The method signatures to retrieve RowId from a ResultSet is as
follows:
   RowId getRowId (int columnIndex)
   RowId getRowId (String columnName)
...

Statement stmt = con.createStatement ();

ResultSet rs = stmt. ExecuteQuery (…);
```

```
while (rs.next ()) {

...

java.sql.RowId rid = rs.getRowId (1);

...

}

//The method signatures to retrieve RowId from a CallableStatement is
as follows:
   RowId getRowId (int parameterIndex)
   RowId getRowId (String parameterName)

Connection con;
...

CallableStatement cstmt = con.prepareCall (…);
...

cstmt.registerOutParameter (2, Types.ROWID);

...

cstmt.executeUpdate ();

...

java.sql.RowId rid = cstmt.getRowId (2);
```

The `RowId` can be used to refer uniquely to a row and thus can be used to retrieve the rows or update the row data. When fetching or updating using `RowId` references, it is important to know the validity of the `RowId`'s lifetime to assure consistent results. It is also advisable to simultaneously use another reference, such as the primary key, to avoid inconsistent results in circumstances where the `RowId` could change transparently.

The `RowId` values can also be set or updated. In the case of an updateable `ResultSet`, the `updateRowId()` method could be used to update the `RowId` for a particular row in a table.

Both the `PreparedStatement` and the `CallableStatement` interfaces support a `setRowId()` method, with different signatures, to set the `RowId` as a parameter value. This value could be used to refer to data rows or to update the `RowId` value for a particular row in a table.

The facility to set or update the `RowId` provides the flexibility to control the unique row identifiers and could be used to make such identifiers unique across the tables used. Perhaps, portability of `RowId` across supporting datasources could also be achieved by explicitly setting consistent values across them. However, because system-generated `RowIds` are often efficient, and transparent tasks could alter `RowIds`, they are best used by an application as a read-only attribute.

Leverage nonstandard vendor implemented resources

The new JDBC API defines a `java.sql.Wrapper` interface. This interface provides the ability to access datasource-vendor-specific resources by retrieving the delegate instance using the corresponding wrapped proxy instance.

This `Wrapper` interface has 17 sub-interfaces as per the current specification and includes `Connection`, `ResultSet`, `Statement`, `CallableStatement`, `PreparedStatement`, `DataSource`, `DatabaseMetaData`, and `ResultSetMetaData`, among others in the list. This is an excellent design as it facilitates datasource-vedor-specific resource implementation at almost all stages of the query-creation and result-set-retrieval lifecycles.

The `unwrap()` method returns the object that implements the given interface to allow access to vendor-specific methods. The `isWrapperFor()` method returns a Boolean value. It returns true if it implements the interface, or if it directly or indirectly is a wrapper for the object.

As an example, when using Oracle, Oracle JDBC drivers provide update batching extensions that are better performing and more efficient as compared to the standard JDBC batch-updating mechanisms. For earlier JDBC versions, this implies using the Oracle-specific definitions, such as `OraclePreparedStatement`, in the code. This compromises code portability. With the new API, many such efficient implementations can be wrapped and exposed within the standard JDBC definitions.

Service provider mechanism for driver loading

In a nonmanaged or standalone program scenario, prior to JDBC 4.0, you would have to load the JDBC driver class explicitly by invoking the `Class.forName` method, as shown in Listing 4:

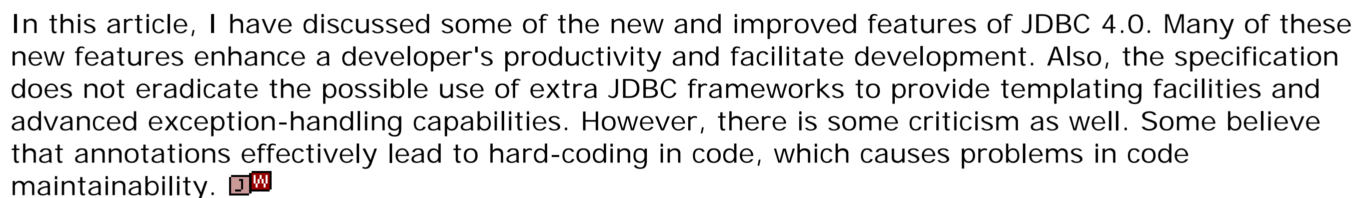
Listing 4. Class.forName

```
Class.forName ("com.driverprovider.jdbc.jdbcDriverImpl");
```

With JDBC 4.0, if the JDBC driver vendors package their drivers as services, defined under the server provider mechanism definitions as per the JAR specification, the `DriverManager` code would implicitly load the driver by searching for it in the classpath. The benefit of this mechanism is that the developer does not need to know about the specific driver class and can write a little less code when using JDBC. Also, since the driver class name is no longer in the code, a name change would not require recompilations. If multiple drivers are specified in the classpath, then `DriverManger` will try and establish a connection using the first driver it encounters in the classpath and iterate further if required.

Conclusion

In this article, I have discussed some of the new and improved features of JDBC 4.0. Many of these new features enhance a developer's productivity and facilitate development. Also, the specification does not eradicate the possible use of extra JDBC frameworks to provide templating facilities and advanced exception-handling capabilities. However, there is some criticism as well. Some believe that annotations effectively lead to hard-coding in code, which causes problems in code maintainability.



About the author

Shashank Tiwari (a.k.a. Shanky) is a software architect at Saven Technologies, an information technology company based out of Illinois. For more than seven years, he has been involved with architecting high-performance distributed applications using J2EE. He advises investment banking

and financial service companies in building robust, quantitative, data intensive, and scalable software applications. He is also an ardent supporter of open source software. He is one of the maintainers of the Python GUI framework called anygui. He lives with his wife and son in New York. More information about him can be accessed at http://www.shanky.org.

Resources

- JSR 221: JDBC 4.0 Specification: http://www.jcp.org/en/jsr/detail?id=221
- Java Database Connectivity at Sun Developer Network: http://java.sun.com/products/jdbc/
- Lance Andersen's Blog (Andersen is the lead for JDBC 4.0): http://weblogs.java.net/blog/lancea/
- Go to AskTom at Oracle.com for concepts related to RowId and OraclePreparedStatements: http://asktom.oracle.com/
- For more articles on JDBC, browse the **Java Database Connectivity (JDBC)** section of *JavaWorld*'s Topical Index: http://www.javaworld.com/channel_content/jw-jdbc-index.shtml
- Also browse the **Java Standard Edition** section of *JavaWorld*'s Topical Index: http://www.javaworld.com/channel_content/jw-j2se-index.shtml?j2se1

Advertisement: Support JavaWorld, click here!

Do you like technology?

Copyright © 2006 JavaWorld.com, an IDG company