Sun Microsystems Inc.

JDBC™ 2.1 API

The JDBC™ API is the Java™ platform standard call-level API for database access. This document contains the final specification of the core JDBC 2.1 API.

Please send technical comments on this specification to:

jdbc@eng.sun.com

Please send product and business questions to:

jdbc-business@eng.sun.com

Version 0.1

Seth White and Mark Hapner

December 16, 1999 5:31 pm

Contents

1 Introduction

1.1 Preface

This document contains the final specification of the JDBC™ 2.1 Core API.

1.2 Target audience

The target audience for this document includes vendors of JDBC technology-based drivers (JDBC drivers) whose products implement the JDBC API, other vendors who want to provide support for the JDBC API in their products, and end-users developing applications using the JDBC API.

1.3 Background

The initial release of the JDBC API has been well received by both end-users developing database applications using Java™ technology, and vendors of database products. Due to its wide acceptance, the JDBC API has become an API that is core to the Java platform. We would like to thank everyone who has worked on or used JDBC technology for helping to make it successful.

Since the first release of the JDBC API, we have received valuable feedback from the members of the community of JDBC technology users regarding new features that would make useful additions to the API. We are continuing to solicit additional input and ideas from everyone who has an interest in JDBC technology, and we encourage everyone to send us their requests and ideas for new features and directions.

1.4 Organization

The JDBC API has been separated into two parts: the core JDBC 2.1 API and the JDBC 2.0 Optional Package API. Chapters 2 and 3 of this document discuss goals and give an overview of the JDBC API as a whole. The remainder of the document contains a detailed specification of the core JDBC 2.1 API. A detailed specification for the JDBC 2.0 Optional Package API is presented in a separate document.

1.5 Terminology

In this document we refer to the previous release of the JDBC API as the JDBC 1.0 API.

1.6 Acknowledgments

The JDBC API design effort has been a broad industry effort that has involved over twenty partner companies. We would like to thank the many reviewers who have contributed to previous versions of the specification. We especially thank Gray Clossman and Stefan Dessloch for making the initial proposal for adding SQL99 types to the JDBC API.

2 Goals

This section outlines the main goals for the JDBC API.

2.1 Leverage the strengths of the JDBC 1.0 and Java platform APIs

One of the important strengths of the JDBC 1.0 API is that it is relatively easy to use. This ease-of-use is in some respects due to the Java programming language, which gives JDBC technology a “hi-level” flavor, despite the fact that it is a call-level interface. For example, users of the JDBC 1.0 API aren’t required to understand low-level memory-management details, or byte-alignment, or data alignment details when transferring data to and from the database.

Despite being easy to use, the JDBC API gives developers using the Java programming language access to a powerful set of database capabilities. JDBC technology can be used to build sophisticated, real-world applications. The JDBC API must continue to strike the proper balance between ease-of-use and providing a rich set of functionality.

2.2 Maintain compatibility with existing applications and drivers

Existing JDBC drivers and the Java programming language applications that use them shall continue to work—unchanged—in an implementation of the Java virtual machine that supports the JDBC 2.1 API. Applications that don’t use any of the new features of the JDBC 2.1 API do not require any changes to continue running. It should be straightforward for existing applications to migrate to the new JDBC API.

2.3 Keep pace with the Java platform

The Java platform has matured since the first release of the JDBC API. Some of the new Java platform APIs that are important for the JDBC API are: the Java Transaction Service (JTS), the Java Naming and Directory Interface™ (JNDI), JavaBeans™, Enterprise JavaBeans™(EJB), and internationalization. The JDBC API must leverage these other Java platform APIs and support them well.

2.4 JavaBeans

The most important new Java platform API for the JDBC API is the JavaBeans API. When the JDBC API was first released there was no component model for the Java platform. The JDBC API should provide a foundation for developers creating data-aware components written to the JavaBeans architecture (JavaBeans components). The JDBC API is a good place to provide this standard component foundation since data access is a ubiquitous aspect of most applications. A new RowSet type has been added to the JDBC Optional Package API to meet this goal. Chapter 4 discusses the differences between the core JDBC 2.1 and JDBC 2.0 Optional Package APIs.

2.5 Advanced Database Features

There are some important features provided by databases that are not supported by the JDBC 1.0 API, like scrollable cursors and advanced data types, such as Binary Large OBjects (BLOBS). The JDBC 2.1 core API must support these advanced features.

The JDBC API should provide a framework that allows developers to access instances of user-defined data types that are stored in a database. The JDBC 2.1 core API supports both databases that provide storage for Java programming language objects (Java objects), and databases that store SQL99 structured types.

The JDBC API should provide some basic support for access to non-SQL tabular data, such as data stored in files. The JDBC Optional Package API provides some limited support for tabular data. See Chapter 4 for a discussion of the differences between the core JDBC 2.1 and JDBC 2.0 Optional Package APIs.

3 Overview of New Features

This chapter contains an overview of the new features that are being added to the JDBC API.

3.1 Result set enhancements

The JDBC 1.0 API provided result sets that had the ability to scroll in a forward direction only. Scrollable result sets allow for more flexibility in the processing of results by providing both forward and backward movement through their contents. In addition, scrollable result sets allow for relative and absolute positioning. For example, it's possible to move to the fourth row in a scrollable result set directly, or to move directly to the third row following the current row, provided the row exists.

The JDBC API allows result sets to be directly updatable, as well.

3.2 Batch updates

The batch update feature allows an application to submit multiple update statements (insert/update/delete) in a single request to the database. This can provide a dramatic increase in performance when a large number of update statements need to be executed.

3.3 Advanced data types

Increased support for storing persistent Java programming language objects (Java objects) and a mapping for SQL99 data types such as binary large objects, and structured types, has been added to the JDBC API. An application may also customize the mapping of SQL99 structured types into Java programming language classes.

3.4 Rowsets

As its name implies, a rowset encapsulates a set of rows. A rowset may or may not maintain an open database connection. When a rowset is 'disconnected' from its data source, updates performed on the rowset are propagated to the underlying database using an optimistic concurrency control algorithm.

Rowsets add support to the JDBC API for the JavaBeans component model. A rowset object is a bean. A rowset implementation may be serializable. Rowsets can be created at design time and used in conjunction with other JavaBeans components in a visual builder tool to construct an application.

3.5 JNDI for naming databases

The Java Naming and Directory Interface (JNDI) API can be used in addition to a JDBC technology-based driver manager (JDBC driver manager) to obtain a connection to a database. When an application uses the JNDI API, it specifies a logical name that identifies a particular database instance and JDBC driver for accessing that database. This has the advantage of making the application code independent of a particular JDBC driver and JDBC technology URL.

3.6 Connection Pooling

The JDBC API contains ‘hooks’ that allow connection pooling to be implemented on top of the JDBC driver layer. This allows for a single connection cache that spans the different JDBC drivers that may be in use. Since creating and destroying database connections is expensive, connection pooling is important for achieving good performance, especially for server applications.

3.7 Distributed transaction support

Support for distributed transactions has been added as an extension to the JDBC API. This feature allows a JDBC driver to support the standard 2-phase commit protocol used by the Java Transaction Service (JTS) API.

3.8 Other new features

Support for character streams has been added. This means that character data can be retrieved and sent to the database as a stream of internationalized Unicode characters. Methods to allow `java.math.BigDecimal` values to be returned with full precision have also been added. Support for time zones has been added.

4 What's Actually Changed

This chapter describes the practical differences between the JDBC 1.0 and JDBC 2.0 APIs.

4.1 A New Package

The JDBC API has been factored into two complementary components. The first component is API that is core to the Java platform (the *core JDBC 2.1 API*) and comprises the updated contents of the `java.sql` package. This document contains the specification for the core JDBC 2.1 API. The second component, termed the *JDBC 2.0 Optional Package API*, comprises the contents of a new package, `javax.sql`, which as its name implies will be delivered as an optional package to the Java platform (formerly Java Standard Extension). The JDBC 2.0 Optional Package API is described in a separate document.

The `java.sql` package contains all of the additions that have been made to the existing interfaces and classes, in addition to a few new classes and interfaces. The new `javax.sql` package has been introduced to contain the parts of the JDBC API which are closely related to other pieces of the Java platform that are themselves Optional Packages, such as the Java Naming and Directory Interface (JNDI), and the Java Transaction Service (JTS). In addition, some advanced features that are easily separable from the core JDBC API, such as connection pooling and rowsets, have also been added to `javax.sql`. Putting these advanced facilities into an optional package instead of into core will help keep the core JDBC API small and focused.

Since optional packages are downloadable, it will always be possible to deploy an application which uses the features in the JDBC Optional Package that will "run anywhere," since if an optional package isn't installed on a client machine, it can be downloaded along with the application that uses it.

4.2 Changes to Classes and Interfaces

The list below contains all of the JDBC 2.1 API core classes and interfaces. Interfaces and classes that are new are listed in bold type. All of the interfaces and classes present in the JDBC 1.0 API are also present in the core JDBC 2.1 API, however, some of the JDBC 1.0 technology interfaces have gained additional methods. The interfaces that contain new methods are listed in italics and those that have not changed are in normal type.

java.sql.Array

java.sql.BatchUpdateException

java.sql.Blob

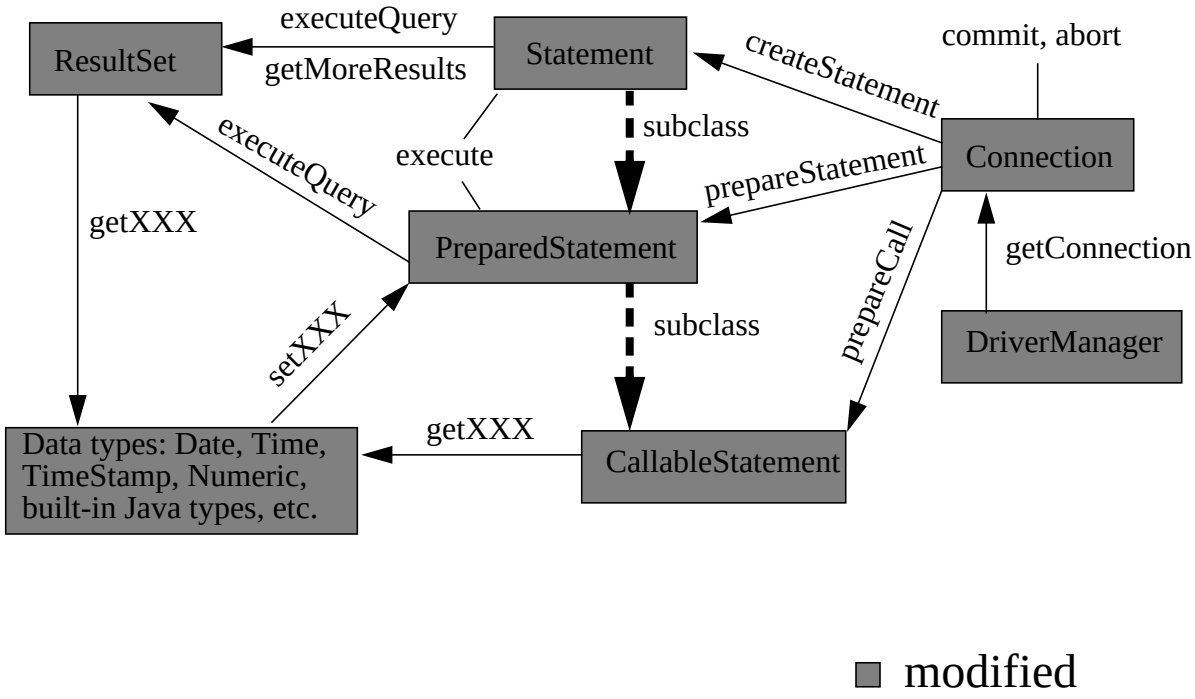
java.sql.CallableStatement

java.sql.Clob

java.sql.Connection

java.sql.DatabaseMetaData
java.sql.DataTruncation
java.sql.Date
java.sql.Driver
java.sql.DriverManager
java.sql.DriverPropertyInfo
java.sql.PreparedStatement
java.sql.Ref
java.sql.ResultSet
java.sql.ResultSetMetaData
java.sql.SQLData
java.sql.SQLException
java.sql.SQLInput
java.sql.SQLOutput
java.sql.SQLWarning
java.sql.Statement
java.sql.Struct
java.sql.Time
java.sql.Timestamp
java.sql.Types

The separate core JDBC 2.1 API documentation contains the Java programming language definitions of the `java.sql` interfaces and classes listed above. The figure below shows the more important core interfaces and their relationships. The important relationships between interfaces have not changed with the introduction of the new JDBC API.



The list below contains the classes and interfaces that comprise the `javax.sql` package. A detailed specification of these new types is contained in a separate document.

javax.sql.ConnectionEvent
javax.sql.ConnectionEventListener
javax.sql.ConnectionPoolDataSurce
javax.sql.DataSource
javax.sql.PooledConnection
javax.sql.RowSet
javax.sql.RowSetEvent
javax.sql.RowSetInternal
javax.sql.RowSetListenerl
javax.sql.RowSetMetaData
javax.sql.RowSetReader
javax.sql.RowSetWriter
javax.sql.XAConnection
javax.sql.XADataSource

5 Result Set Enhancements

This chapter discusses the new functionality that has been added to result sets. The goal of the enhancements is to add two new basic capabilities to result sets: scrolling and updatability. Several methods have also been added to enable a JDBC driver to deliver improved performance when processing results. A variety of examples are included to illustrate the new features.

5.1 Scrolling

A result set created by executing a statement may support the ability to move backward (last-to-first) through its contents, as well as forward (first-to-last). Result sets that support this capability are called scrollable result sets. Result sets that are scrollable also support relative and absolute positioning. Absolute positioning is the ability to move directly to a row by specifying its absolute position in the result set, while relative positioning gives the ability to move to a row by specifying a position that is relative to the current row. The definition of absolute and relative positioning in the JDBC API is modeled on the X/Open SQL CLI specification.

5.2 Result Set types

The JDBC 1.0 API provided one result set type—forward-only. The JDBC 2.1 core API provides three result set types: forward-only, scroll-insensitive, and scroll-sensitive. As their names suggest, the new result set types support scrolling, but they differ in their ability to make changes visible while they are open.

A scroll-insensitive result set is generally **not** sensitive to changes that are made while it is open. A scroll-insensitive result set provides a static view of the underlying data it contains. The membership, order, and column values of rows in a scroll-insensitive result set are typically fixed when the result set is created.

On the other hand, a **scroll-sensitive** result set is sensitive to changes that are made while it is open, and provides a ‘dynamic’ view of the underlying data. For example, when using a scroll-sensitive result set, changes in the underlying column values of rows are visible. The membership and ordering of rows in the result set may be fixed—this is implementation defined.

5.3 Concurrency types

An application may choose from two different concurrency types for a result set: read-only and updatable.

A result set that uses **read-only** concurrency does not allow updates of its contents. This can increase the overall level of concurrency between transactions, since any number of read-only locks may be held on a data item simultaneously.

A result set that is **updatable** allows updates and may use database write locks to mediate access to the same data item by different transactions. Since only a single write lock may be held at a time on a data item, this can reduce concurrency. Alternatively, an optimistic concurrency control scheme may be used if it is thought that conflicting

accesses to data will be rare. Optimistic concurrency control implementations typically compare rows either by value or by a version number to determine if an update conflict has occurred.

5.4 Performance

Two performance hints may be given to a JDBC 2.1 technology-enabled driver to make access to result set data more efficient. Specifically, the number of rows to be fetched from the database each time more rows are needed can be specified, and a direction for processing the rows—forward, reverse, or unknown—can be given as well. These values can be changed for an individual result set at any time. A JDBC driver may ignore a performance hint if it chooses.

5.5 Creating a result set

The example below illustrates creation of a result set that is forward-only and uses read-only concurrency. No performance hints are given by the example, so the driver is free to do whatever it thinks will result in the best performance. The transaction isolation level for the connection is not specified, so the default transaction isolation level of the underlying database is used for the result set that is created. Note that this code is just written using the JDBC 1.0 API, and that it produces the same type of result set that would have been produced by the JDBC 1.0 API.

```
Connection con = DriverManager.getConnection(
        "jdbc:my_subprotocol:my_subname");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
        "SELECT emp_no, salary FROM employees");
```

The next example creates a scrollable result set that is updatable and sensitive to updates. Rows of data are requested to be fetched twenty-five at-a-time from the database.

```
Connection con = DriverManager.getConnection(
        "jdbc:my_subprotocol:my_subname");

Statement stmt = con.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);
stmt.setFetchSize(25);

ResultSet rs = stmt.executeQuery(
        "SELECT emp_no, salary FROM employees");
```

The example below creates a result set with the same attributes as the previous example, however, a prepared statement is used to produce the result set.

```
PreparedStatement pstmt = con.prepareStatement(
        "SELECT emp_no, salary FROM employees where emp_no = ?",
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);

pstmt.setFetchSize(25);
pstmt.setString(1, "100010");
ResultSet rs = pstmt.executeQuery();
```

The method `DatabaseMetaData.supportsResultSetType()` can be called to see which result set types are supported by a JDBC driver. However, an application may still ask a JDBC driver to create a `Statement`, `PreparedStatement`, or `CallableStatement` object using a result set type that the driver does not support. In this case, the driver should issue an `SQLWarning` on the `Connection` that produces the statement and choose an alternative value for the result set type of the statement according to the following rules:

1. If an application asks for a scrollable result set type the driver should use a scrollable type that it supports, even if this differs from the exact type requested by the application.
2. If the application asks for a scrollable result set type and the driver does not support scrolling, then the driver should use a forward-only result set type.

Similarly, the method `DatabaseMetaData.supportsResultSetConcurrency()` can be called to determine which concurrency types are supported by a driver. If an application asks a JDBC driver for a concurrency type that it does not support then the driver should issue an `SQLWarning` on the `Connection` that produces the statement and choose the alternative concurrency type. The choice of result set type should be made first if an application specifies both an unsupported result set type and an unsupported concurrency type.

In some instances, a JDBC driver may need to choose an alternate result set type or concurrency type for a `ResultSet` at statement execution time. For example, a SELECT statement that contains a join over multiple tables may not produce a `ResultSet` that is updatable. The JDBC driver should issue an `SQLWarning` in this case on the `Statement`, `PreparedStatement`, or `CallableStatement` that produces the `ResultSet` and choose an appropriate result set type or concurrency type as described above. An application may determine the actual result set type and concurrency type of a `ResultSet` by calling the `ResultSet.getType()` and `getConcurrency()` methods, respectively.

5.6 Updates

A result set is updatable if its concurrency type is `CONCUR_UPDATABLE`. Rows in an updatable result set may be updated, inserted, and deleted. The example below updates the first row of a result set. The `ResultSet.updateXXX()` methods are used to modify the value of an individual column in the current row, but do not update the underlying database. When the `ResultSet.updateRow()` method is called the database is updated. Columns may be specified by name or number.

```
rs.first();
rs.updateString(1, "100020");
rs.updateFloat(“salary”, 10000.0f);
rs.updateRow();
```

The updates that an application makes must be discarded by a JDBC driver if the application moves the cursor from the current row before calling `updateRow()`. In addition, an application can call the `ResultSet.cancelRowUpdates()` method to explicitly cancel the updates that have been made to a row. The `cancelRowUpdates()` method must be called after calling `updateXXX()` and before calling `updateRow()`, otherwise it has no effect.

The following example illustrates deleting a row. The fifth row in the result set is deleted from the database.

```
rs.absolute(5);
rs.deleteRow();
```

The example below shows how a new row may be inserted into a result set. The JDBC API defines the concept of an *insert row* that is associated with each result set and is used as a staging area for creating the contents of a new row before it is inserted into the result set itself. The `ResultSet.moveToInsertRow()` method is used to position the result set’s cursor on the insert row. The `ResultSet.updateXXX()` and `ResultSet.getXXX()` methods are used to update and retrieve individual column values from the insert row. The contents of the insert row is undefined immediately after calling `ResultSet.moveToInsertRow()`. In other words, the value returned by calling a `ResultSet.getXXX()` method is undefined after `moveToInsertRow()` is called until the value is set by calling `ResultSet.updateXXX()`.

Calling `ResultSet.updateXXX()` while on the insert row does not update the underlying database or the result set. Once all of the column values are set in the insert row, `ResultSet.insertRow()` is called to update the result set and the database simultaneously. If a column is not given a value by calling `updateXXX()` while on the insert row, or a column is missing from the result set, then that column must allow a null value. Otherwise, calling `insertRow()` throws an `SQLException`.

```
rs.moveToInsertRow();
rs.updateString(1, "100050");
rs.updateFloat(2, 1000000.0f);
rs.insertRow();
rs.first();
```

A result set remembers the current cursor position “in the result set” while its cursor is temporarily positioned on the insert row. To leave the insert row, any of the usual cursor positioning methods may be called, including the special method `ResultSet.moveToCurrentRow()` which returns the cursor to the row which was the current row before `ResultSet.moveToInsertRow()` was called. In the example above, `ResultSet.first()` is called to leave the insert row and move to the first row of the result set.

Due to differences in database implementations, the JDBC API does not specify an exact set of SQL queries which must yield an updatable result set for JDBC drivers that support updatability. Developers can, however, generally expect queries which meet the following criteria to produce an updatable result set:

1. The query references only a single table in the database.
2. The query does not contain any join operations.
3. The query selects the primary key of the table it references.

In addition, an SQL query should also satisfy the conditions listed below if inserts are to be performed.

4. The query selects all of the non-nullable columns in the underlying table.
5. The query selects all columns that don’t have a default value.

5.7 Cursor movement examples

A result set maintains an internal pointer called a *cursor* that indicates the row in the result set that is currently being accessed. A result set cursor is analogous to the cursor on a computer screen which indicates the current screen position. The cursor maintained by a forward-only result set can only move forward through the contents of the result set. Thus, rows are accessed sequentially beginning with the first row.

Iterating forward through a result set is done by calling the `ResultSet.next()` method, as with the JDBC 1.0 API. In addition, scrollable result sets—any result set whose type is not forward only—implement the method, `beforeFirst()`, which may be called to position the cursor before the first row in the result set.

The example below positions the cursor before the first row and then iterates forward through the contents of the result set. The `getXXX()` methods, which are JDBC 1.0 API methods, are used to retrieve column values.

```
rs.beforeFirst();
while ( rs.next()) {
        System.out.println(rs.getString("emp_no") +
                                " " + rs.getFloat("salary"));
}
```

Of course, one may iterate backward through a scrollable result set as well, as is shown below.

```
rs.afterLast();
while (rs.previous()) {
        System.out.println(rs.getString(“emp_no”) +
                “ “ + rs.getFloat(“salary”));
}
```

In this example, the `ResultSet.afterLast()` method positions the scrollable result set’s cursor after the last row in the result set. The `ResultSet.previous()` method is called to move the cursor to the last row, then the next to last, and so on. `ResultSet.previous()` returns `false` when there are no more rows, so the loop ends after all of the rows have been visited.

After examining the `ResultSet` interface, the reader will no doubt recognize that there is more than one way to iterate through the rows of a scrollable result set. It pays to be careful, however, as is illustrated by the following example, which shows one alternative that is incorrect.

```
// incorrect!!!
while (!rs.isAfterLast()) {
        rs.relative(1);
        System.out.println(rs.getString("emp_no") +
                " " + rs.getFloat("salary"));
}
```

This example attempts to iterate forward through a scrollable result set and is incorrect for several reasons. One error is that if `ResultSet.isAfterLast()` is called when the result set is empty, it will return a value of false since there is no last row, and the loop body will be executed, which is not what is wanted. An additional problem occurs when the cursor is positioned before the first row of a result set that contains data. In this case calling `rs.relative(1)` is erroneous since there is no current row.

The code sample below fixes the problems in the previous example. Here a call to `ResultSet.first()` is used to distinguish the case of an empty result set from one which contains data. Since `ResultSet.isAfterLast()` is only called when the result set is non-empty the loop control works correctly, and `ResultSet.relative(1)` steps through the rows of the result set since `ResultSet.first()` initially positions the cursor on the first row.

```
if (rs.first()) {
        while (!rs.isAfterLast()) {
                System.out.println(rs.getString("emp_no") +
                        " " + rs.getFloat("salary"));
                rs.relative(1);
        }
}
```

5.8 Detecting and viewing changes

So far, we have introduced the different result set types and shown a few examples of how a result set of a particular type can be created, updated, and traversed. This section goes into more detail on the differences between result set types, and what these differences mean for an application that uses result sets.

The different result set types—forward-only, scroll-insensitive, and scroll-sensitive—provided by the JDBC API vary greatly in their ability to make changes in the underlying data visible to an application. This aspect of result sets is particularly interesting for the result set types which support scrolling, since they allow a particular row to be visited multiple times while a result set is open.

5.8.1 Visibility of changes

We begin the discussion of this topic by describing the visibility of changes at the transaction level. First, note the seemingly obvious fact that all of the updates that a transaction makes are visible to itself. However, the changes (updates, inserts, and deletes) made by other transactions that are visible to a particular transaction are determined by the transaction isolation level. The isolation level for a transaction can be set by calling

```
con.setTransactionIsolation(TRANSACTION_READ_COMMITTED);
```

where the variable `con` has type `Connection`. If all transactions in a system execute at the `TRANSACTION_READ_COMMITTED` isolation level or higher, then a transaction will only see the committed changes of other transactions. The changes that are visible to a result set's enclosing transaction when a result set is opened are always visible through the result set. In fact, this is what it means for an update made by one transaction to be visible to another transaction.

But what about changes made while a result set is open? Are they visible through the result set by, for example, calling `ResultSet.getXXX()`? Whether a particular result set exposes changes to its underlying data made by other transactions, other result sets that are part of the same transaction (We refer to these two types of changes collectively as 'other's changes'.), or itself while the result set is open depends on the result set type.

5.8.2 Other's changes

A scroll-insensitive result set does not make any changes visible that are made by others—other transactions and other result sets in the same transaction—once the result set is opened. The content of a scroll-insensitive result set with respect to changes made by others is static—the membership, ordering, and row values are fixed. For example, if another transaction deletes a row that is contained in a static result set while it is open, the row remains visible. One way to implement a scroll-insensitive result set is to create a private copy of the result set's data.

Scroll-sensitive result sets lie at the opposite end of the spectrum. A scroll-sensitive result set makes all of the updates made by **others** that are visible to its enclosing transaction visible. Inserts and deletes may not be visible, however.

Let us define carefully what it means for updates to be visible. If an update made by another transaction affects where a row should appear in the result set—this is in effect a delete followed by an insert—the row may not move until the result set is reopened. If an update causes a row to fail to qualify for membership in a result set—this is in effect a delete—the row may remain visible until the result set is reopened. If a row is explicitly deleted by another transaction, a scroll-sensitive result set may maintain a placeholder for the row to permit logical fetching of rows by absolute position. Updated column values are always visible, however.

The `DatabaseMetaData` interface provides a way to determine the exact capabilities that are supported by a result set. For example, the new methods: `othersUpdatesAreVisible`, `othersDeletesAreVisible`, and `othersInsertsAreVisible` may be used for this purpose.

A forward-only result set is really a degenerate case of either a scroll-insensitive or scroll-sensitive result set— depending on how the DBMS evaluates the query that produces the result set. Most DBMSs have the ability to materialize query results incrementally for some queries. If a query result is materialized incrementally, then data values aren't actually retrieved until they are needed from the DBMS and the result set will behave like a sensitive result set. For some queries, however, incremental materialization isn't possible. For example, if the result set is sorted, the entire result set may need to be produced a priori before the first row in the result set is returned to the application by the DBMS. In this case a forward-only result set will behave like an insensitive result set.

For a `TYPE_FORWARD_ONLY` result set the `othersUpdatesAreVisible`, `othersDeletesAreVisible`, and `othersInsertsAreVisible` methods determine whether inserts, updates, and deletes are visible when the result set is materialized incrementally by the DBMS. If the result of a query is sorted then incremental materialization may not be possible and changes will not be visible, even if the methods above return true.

5.8.3 A result set's own changes

We have pointed out that the visibility of changes made by others generally depends on a result set's type. A final point that concerns the visibility of changes via an open result set is whether a result set can see its own changes (inserts, updates, and deletes). A JDBC technology application can determine if the changes made by a result set are visible to the result set itself by calling the `DatabaseMetaData` methods: `ownUpdatesAreVisible`, `ownDeletesAreVisible`, and `ownInsertsAreVisible`. These methods are needed since this capability can vary between DBMSs and JDBC drivers.

One's own updates are visible if an updated column value can be retrieved by calling `getXXX()` following a call to `updateXXX()`. Updates are **not** visible if `getXXX()` still returns the initial column value after `updateXXX()` is called. Similarly, an inserted row is visible if it appears in the result set following a call to `insertRow()`. An inserted row is not visible if it does not appear in the result set immediately after `insertRow()` is called—without closing and reopening the result set. Deletions are visible if deleted rows are either removed from the result set or if deleted rows leave a *hole* in the result set.

The following example, shows how an application may determine whether a TYPE_SCROLL_SENSITIVE result set can see its own updates.

```
DatabaseMetaData dmd;
...
if  (dmd.ownUpdatesAreVisible(ResultSet.TYPE_SCROLL_INSENSITIVE))
{
        // changes are visible
}
```

5.8.4 Detecting changes

The ResultSet.wasUpdated(), wasDeleted(), and wasInserted() methods can be called to determine whether a row has been effected by a visible update, delete, or insert respectively since the result set was opened. The ability of a result set to detect changes is orthogonal to its ability to make changes visible. In other words, visible changes are not automatically detected.

The DatabaseMetaData interface provides methods that allow an application to determine whether a JDBC driver can detect changes for a particular result set type. For example,

```
boolean bool = dmd.deletesAreDetected(
        ResultSet.TYPE_SCROLL_SENSITIVE);
```

If deletesAreDetected returns true, then ResultSet.wasDeleted() can be used to detect 'holes' in a TYPE_SCROLL_SENSITIVE result set.

5.9 Refetching a row

Some applications may need to see up-to-the-second changes that have been made to a row. Since a JDBC driver can do prefetching and caching of data that is read from the underlying database (see ResultSet.setFetchSize()), an application may not see the very latest changes that have been made to a row, even when a sensitive result set is used and updates are visible. The ResultSet.refreshRow() method is provided to allow an application to request that a driver refresh a row with the latest values stored in the database. A JDBC driver may actually refresh multiple rows at once if the fetch size is greater than one. Applications should exercise restraint in calling refreshRow(), since calling this method frequently will likely slow performance.

5.10 JDBC API compliance

Although we expect most JDBC drivers to support scrollable result sets, we have made them optional to minimize the complexity of implementing JDBC drivers for data sources that do not support scrollability. The goal is that it be possible for a JDBC driver to implement scrollable result sets using the support provided by the underlying database system for systems that have such support. If the DBMS associated with a driver does not support scrollability then this feature may be omitted, or a JDBC driver may

implement scrollability as a layer on top of the DBMS. Its important to note that JDBC technology rowsets, which are part of the JDBC Optional Package API, always support scrollability, so a rowset can be used when the underlying DBMS doesn't support scrollable results.

6 Batch Updates

The batch update facility allows multiple update operations to be submitted to a data source for processing at once. Submitting multiple updates together, instead of individually, can greatly improve performance. `Statement`, `PreparedStatement`, and `CallableStatement` objects can be used to submit batch updates.

6.1 Description of batch updates

6.1.1 Statements

The batch update facility allows a `Statement` object to submit a set of heterogeneous update commands together as a single unit, or batch, to the underlying DBMS. In the example below all of the update operations required to insert a new employee into a fictitious company database are submitted as a single batch.

```
// turn off autocommit
con.setAutoCommit(false);

Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, 260)");

// submit a batch of update commands for execution
int[] updateCounts = stmt.executeBatch();
```

In the example, autocommit mode is disabled to prevent the driver from committing the transaction when `Statement.executeBatch()` is called. Disabling autocommit allows an application to decide whether or not to commit the transaction in the event that an error occurs and some of the commands in a batch cannot be processed successfully. For this reason, autocommit should always be turned off when batch updates are done. The commit behavior of `executeBatch` is always implementation defined when an error occurs and autocommit is true.

To keep our discussion of batch updates general, we define the term *element* to refer to an individual member of a batch. As we have seen, an element in a batch is just a simple command when a `Statement` object is being used. Although we are focusing on using `Statement` objects to do batch updates in this section, the discussion that follows applies to `PreparedStatment` and `CallableStatement` objects, as well.

In the new JDBC API, a `Statement` object has the ability to keep track of a list of commands—or batch—that can be submitted together for execution. When a `Statement` object is created, its associated batch is empty—the batch contains no elements. The `Statement.addBatch()` method adds an element to the calling statement's batch. The method `Statement.clearBatch()` (not shown above) can be called to reset a batch if

the application decides not to submit a batch of commands that has been constructed for a statement.

Successful execution

The `Statement.executeBatch()` method submits a statement's batch to the underlying data source for execution. Batch elements are executed serially (at least logically) in the order in which they were added to the batch. When all of the elements in a batch execute successfully, `executeBatch()` returns an integer array containing one entry for each element in the batch. The entries in the array are ordered according to the order in which the elements were processed (which, again, is the same as the order in which the elements were originally added to the batch). An entry in the array may have the following values:

1. If the value of an array entry is greater than or equal to zero, then the batch element was processed successfully and the value is an update count indicating the number of rows in the database that were effected by the element's execution.
2. A value of -2 indicates that a element was processed successfully, but that the number of effected rows is unknown.

Calling `executeBatch()` closes the calling `Statement` object's current result set if one is open. The statement's internal list of batch elements is reset to empty once `executeBatch()` returns. The behavior of the `executeQuery`, `executeUpdate`, or `execute` methods is implementation defined when a statement's batch is non-empty.

`ExecuteBatch()` throws a `BatchUpdateException` if any of the elements in the batch fail to execute properly, or if an element attempts to return a result set. Only DDL and DML commands that return a simple update count may be executed as part of a batch. When a `BatchUpdateException` is thrown, the `BatchUpdateException.getUpdateCounts()` method can be called to obtain an integer array of update counts that describes the outcome of the batch execution.

Handling failures during execution

A JDBC driver may or may not continue processing the remaining elements in a batch once execution of an element in a batch fails. However, a JDBC driver must always provide the same behavior when used with a particular DBMS. For example, a driver cannot continue processing after a failure for one batch, and not continue processing for another batch.

If a driver stops processing after the first failure, the array returned by `BatchUpdateException.getUpdateCounts()` will always contain fewer entries than there were elements in the batch. Since elements are executed in the order that they are added to the batch, if the array contains N elements, this means that the first N elements in the batch were processed successfully when `executeBatch()` was called.

When a driver continues processing in the presence of failures, the number of elements, N, in the array returned by `BatchUpdateException.getUpdateCounts()`is always equal to the number of elements in the batch. The following additional array value is

returned when a `BatchUpdateException` is thrown and the driver continues processing after a failure:

3. A value of -3 indicates that the command or element failed to execute successfully. This value is also returned for elements that could not be processed for some reason—such elements fail implicitly.

JDBC drivers that do not continue processing after a failure never return -3 in an update count array. Drivers of this type simply return a status array containing an entry for each command that was processed successfully.

A JDBC technology based application can distinguish a JDBC driver that continues processing after a failure from one that does not by examining the size of the array returned by `BatchUpdateException.getUpdateCounts()`. A JDBC driver that continues processing always returns an array containing one entry for each element in the batch. A JDBC driver that does not continue processing after a failure will always return an array whose number of entries is less than the number of elements in the batch.

6.1.2 PreparedStatements

An element in a batch consists of a parameterized command and an associated set of parameters when a `PreparedStatement` is used. The batch update facility is used with a `PreparedStatement` to associate multiple sets of input parameter values with a single `PreparedStatement` object. The sets of parameter values together with their associated parameterized update command can then be sent to the underlying DBMS engine for execution as a single unit.

The example below inserts two new employee records into a database as a single batch. The `PreparedStatement.setXXX()` methods are used to create each parameter set (one for each employee), while the `PreparedStatement.addBatch()` method adds a set of parameters to the current batch.

```
// turn off autocommit
con.setAutoCommit(false);

PreparedStatement stmt = con.prepareStatement(
        "INSERT INTO employees VALUES (?, ?)");

stmt.setInt(1, 2000);
stmt.setString(2, "Kelly Kaufmann");
stmt.addBatch();

stmt.setInt(1, 3000);
stmt.setString(2, "Bill Barnes");
stmt.addBatch();

// submit the batch for execution
int[] updateCounts = stmt.executeBatch();
```

Finally, `PreparedStatement.executeBatch()` is called to submit the updates to the DBMS. Calling `PreparedStatement.executeBatch()` clears the statement's associated list of batch elements. The array returned by `PreparedStatement.executeBatch()` contains an element for each set of parameters in the batch, similar to the case for `Statement`. Each element either contains an update count or the generic 'success' indicator (-2).

Error handling in the case of `PreparedStatement` objects is the same as error handling in the case of `Statement` objects. Some drivers may stop processing as soon as an error occurs, while others may continue processing the rest of the batch. As for `Statement`, the number of elements in the array returned by `BatchUpdateException.getUpdateCounts()` indicates whether or not the driver continues processing after a failure. The same three array element values are possible, as for `Statement`. The order of the entries in the array is the same order as the order in which elements were added to the batch.

6.1.3 Callable Statements

The batch update facility works the same with `CallableStatement` objects as it does with `PreparedStatement` objects. Multiple sets of input parameter values may be associated with a callable statement and sent to the DBMS together. Stored procedures invoked using the batch update facility with a callable statement must return an update count, and may not have out or inout parameters. The `CallableStatement.executeBatch()` method should throw an exception if this restriction is violated. Error handling is analogous to `PreparedStatement`.

6.2 What's required

Support for batch updates is optional. If a JDBC driver supports batch updates, then the `DatabaseMetaData.supportsBatchUpdates()` method must return true, else it must return false. In addition, to preserve backward compatibility, JDBC drivers that do not continue processing after a failure are not required to return a value of -2 as described in Section 6.1, however, this is encouraged. JDBC drivers that continue processing are required to support both of the negative return values.

Note: In the future, the JDBC API shall define symbolic constants for the negative array entry values described in Section 6.1. These values have been added as an addendum to the original JDBC 2.0 API specification.

7 Persistence for Java Objects

The JDBC 1.0 API provided some support for storing objects and retrieving Java objects from a database via the `getObject()` and `setObject()` mechanism. The new JDBC API enhances the ability of a JDBC driver to implement persistence for Java objects in general, by providing new metadata capabilities that can be used to retrieve a description of the Java objects that a data source contains. Instances of a Java programming language class (Java class) can be stored in a database as serialized Java objects, or in some other vendor specific format. If object serialization is used then references between objects can be treated according to the rules specified by Java object serialization.

The new JDBC API features described in this chapter are intended to support a new generation of Java-aware database management systems, termed *Java-relational DBMSs*. A Java-relational DBMS extends the type system of a database with Java object types and allows users to write queries that reference these types. Several database vendors are creating products with Java-relational capabilities. The mechanisms described in this chapter are optional. JDBC drivers that do not support the capabilities described in this chapter are not required to implement them.

Lets take a look at how a typical application written in the Java programming language (Java application) can make use of the JDBC API to store and retrieve Java objects.

7.1 Retrieving Java objects

The example below shows how objects can be retrieved using the JDBC API. The example query references a table, `PERSONNEL`, that contains a column called `Employee` containing instances of the Java class `Employee`. Here, the column name, `Employee`, and the Java class name are the same, but this is not required by the JDBC API. In fact, since there is currently not a standard, agreed upon syntax for SQL queries that reference Java programming language types (Java types), the JDBC API does not mandate the use of any particular query syntax.

```
ResultSet rs = stmt.executeQuery(
        “SELECT Employee FROM PERSONNEL“);
rs.next();
Employee emp = (Employee)rs.getObject(1);
```

The example selects all of the `Employee` instances from the `PERSONNEL` table. The `ResultSet.next()` method is called to position the result set to the first row containing an `Employee`. The example application then obtains an `Employee` instance by calling `ResultSet.getObject()`. This causes the JDBC driver to construct an instance of the `Employee` class, possibly by deserializing a serialized object instance, and return the instance as a `java.lang.Object` which the application then narrows to an `Employee`.

Note that the example above does not contain any additions to the JDBC 1.0 API aside from possibly requiring some form of extended SQL query syntax which is not specified by the JDBC API. As an aside, we note that the JDBC technology based code shown above can also be used to retrieve data of an SQL user-defined type that is being mapped to a Java class. The details on how this is done are specified in a later chapter.

7.2 Storing Java objects

The following example code illustrates the process of updating a Java object and making the updated copy of the object persistent using JDBC technology.

```
emp.setSalary(emp.getSalary() * 1.5);
PreparedStatement pstmt = con.preparedStatement(
  “UPDATE PERSONNEL SET Employee = ? WHERE Employee.no = 1001”);
pstmt.setObject(1, emp);
pstmt.executeUpdate();
```

The example gives an employee a 50% raise. First, the `Employee.setSalary()` method is called to update the value of the employee’s salary. Note that the semantics of methods on the `Employee` class are not specified by the JDBC API. Here, we assume that the `Employee` class is an ordinary Java class, so calling `Employee.setSalary()` just changes the value of some private data field contained in the `Employee` instance. Calling `Employee.setSalary()` does not update the database, for example, although an alternative implementation could do this, in effect making database updates ‘transparent’ to applications that use the `Employee` class.

Next, a `PreparedStatement` object is created using an extended SQL UPDATE command—the query syntax used in the example is again not mandated by the JDBC API. The UPDATE command specifies that the `Employee` column in the `PERSONNEL` table is to be changed for a specified row. `PreparedStatement.setObject()` is used to pass the `Employee` object to the prepared statement, and the `executeUpdate()` method updates the `Employee` value stored in the database.

Note once again that the example above does not involve any syntactic additions to the JDBC 1.0 API. In addition, the same code could be used if the Employee class was being mapped to an SQL user-defined type.

7.3 Additional metadata

The new JDBC API contains new metadata support that allows an application to obtain a complete description of the Java objects that are stored in a data source.

7.3.1 Identifying Java objects

A new type code, `JAVA_OBJECT`, has been added to `java.sql.Types` to denote a Java object type. The `JAVA_OBJECT` type code is returned by methods such as `DatabaseMetaData.getTypeInfo()` and `DatabaseMetaData.getColumns()`. For example, if a DBMS supports types that can be a Java class,

`DatabaseMetaData.getTypeInfo()` would return a result set containing the following entry:

1. **TYPE_NAME** String => data source specific name (may be null)
2. **DATA_TYPE** short => `java.sql.Types.JAVA_OBJECT`
3. etc.

The TYPE_NAME column contains the data source specific term for a Java object, such as “JavaObject”, “Serialized” etc. TYPE_NAME may be null.

7.3.2 Retrieving schema-specific Java type descriptions

A Java class is typically registered with a particular database schema before it is used in defining the schema’s tables. Information on schema-specific user-defined types—of which `JAVA_OBJECT` types are one particular kind—can be retrieved by calling the `DatabaseMetaData.getUDTs()` method. For example,

```
int[] types = {Types.JAVA_OBJECT};
ResultSet rs = dmd.getUDTs("catalog-name", "schema-name",
        "%", types);
```

returns descriptions of all the Java object types **defined** in the `catalog-name.schema-name` schema. If the driver does not support UDTs or no matching UDTs are found then an empty result set is returned.

Each type description has the following columns:

TYPE_CAT	String => the type's catalog (may be null)
TYPE_SCHEM	String => the type's schema (may be null)
TYPE_NAME	String => the database type name
JAVA_CLASS	String => a Java classname
DATA_TYPE	short => value defined in `java.sql.Types`, e.g. JAVA_OBJECT
REMARKS	String => explanatory comment on the type

The **TYPE_CAT**, **TYPE_SCHEM**, **DATA_TYPE**, and **REMARKS** columns should be self-explanatory. The **TYPE_NAME** is, in effect, the SQL type name. This is the name used in a CREATE TABLE statement to specify a column of this type.

When **DATA_TYPE** is `JAVA_OBJECT`, the **JAVA_CLASS** is the fully qualified Java class name of the Java class associated with **TYPE_NAME**. All values actually stored in a **TYPE_NAME** column must be instances of this class or one of its subclasses. Instances of this class or a subclass are materialized by the JDBC driver when values are fetched from a **TYPE_NAME** column by an application that uses JDBC technology.

The `DatabaseMetaData.getUDTs()` method also accepts a fully qualified SQL name as its third parameter. In this case the catalog and schema pattern parameters are ignored. The fully qualified SQL name may contain wildcards. For example, the code sample below is equivalent to the previous example,

```
int[] types = {Types.JAVA_OBJECT};
ResultSet rs = dmd.getUDTs(null, null,
        "catalog-name.schema-name.%", types);
```

Here we have assumed that the '.' character is used to separate the elements of a fully qualified name. Note that since the format of fully qualified names may vary between database systems, one should generally not hardcode fully qualifed names as in the example above. The `DatabaseMetaData` interface provides information about the format of fully qualified names that is supported by a particular JDBC driver.

7.3.3 Retrieving the Java class object

The JDBC API doesn't provide any special support for loading the Java class files that correspond to Java objects being stored in a database. An application should be able to obtain the class object that corresponds to an object in the database by calling `Class.forName()` and passing the class name as a parameter. In other words, the JDBC API assumes that the bytecodes for objects stored in the database are loaded via the usual Java programming language mechanism.

8 New SQL Types

The next two chapters discuss additions to the JDBC API that allow a Java application to access new SQL data types, such as binary large objects and structured types. JDBC drivers that do not support the new SQL types need not implement the methods and interfaces described in these chapters.

8.1 Taxonomy of SQL Types

The latest version of the ANSI/ISO SQL standard is commonly referred to as *SQL99*. The JDBC API incorporates a model of the new SQL99 types that includes only those properties that are essential to exchanging data between Java applications and databases. The JDBC API should not be affected if some details of the syntax and server-side semantics of the new SQL99 types are altered before the draft becomes an official standard.

The SQL99 draft specifies these data types:

- SQL2 built-in types—the familiar SQL ‘column types’
  - CHAR
  - FLOAT
  - DATE
  - etc.
- New built-in types—new types added by SQL99
  - BLOB—a Binary Large OBject
  - CLOB—a Character Large OBject
- Structured types, for example:
  - CREATE TYPE PLANE_POINT (X FLOAT, Y FLOAT)
- Distinct types—based on the representation of a built-in type, for example:
  - CREATE TYPE MONEY AS NUMERIC(10,2)
- Constructed types—based on a given base type:
  - REF(structured-type)—designates row containing a structured type instance
  - base-type ARRAY[n]—an array of n base-type elements
- Locator types—designate a datum that resides on the server
  - LOCATOR(structured-type)—locator to structured instance in server
  - LOCATOR(array)—locator to array in server
  - LOCATOR(blob)—locator to Binary Large Object in server
  - LOCATOR(clob)—locator to Character Large Object in server

A `REF` value persistently denotes an instance of a structured type that resides in the database. A `LOCATOR` exists only in the client environment and is a transient, logical pointer to data that resides on the database server. A locator typically refers to data that is

too large to materialize on the client, for example, images or audio. There are operators defined at the SQL level to retrieve random-access pieces of the data denoted by the locator.

The remainder of this chapter discusses the default mechanism provided by the JDBC API for accessing each of the new SQL types mentioned above. The JDBC API also provides a means for customizing the mapping of SQL distinct and structured types into Java classes. This mechanism is discussed in the Chapter 9.

8.2 Blobs and clobs

8.2.1 Retrieving blobs and clobs

The binary large object (blob) and character large object (clob) data types are treated similarly to the existing, built-in types defined in the JDBC API. Values of these types can be retrieved by calling the `getBlob()` and `getClob()` methods that appear on the `ResultSet` and `CallableStatement` interfaces. For example,

```
Blob blob = rs.getBlob(1);
Clob clob = rs.getClob(2);
```

retrieves a blob value from the first column of the result set and a clob value from the second column. The `Blob` interface contains operations for returning the length of the blob, a specific range of bytes contained in the blob, etc. The `Clob` interface contains corresponding operations that are character based. See the accompanying API documentation for more details.

An application does not deal directly with the LOCATOR(blob) and LOCATOR(clob) types that are defined in SQL. By default, a JDBC driver should implement the `Blob` and `Clob` interfaces using the appropriate locator type. Also, by default `Blob` and `Clob` objects only remain valid during the **transaction** in which they are created. A JDBC driver may allow these defaults to be changed. For example, the lifetime of `Blob` and `Clob` objects could be changed to session-scoped. However, the JDBC API does not specify how this is done.

8.2.2 Storing blobs and clobs

A `Blob` or `Clob` value can be passed as an input parameter to a `PreparedStatement` object just like other data types by calling the `setBlob()` and `setClob()` methods respectively. The `setBinaryStream()`, and `setObject()` methods may be used to input a stream value as a blob. The `setAsciiStream()`, `setUnicodeStream()`, and `setObject()` methods may be used to input a stream as a clob value.

8.2.3 Metadata additions

Two new type codes, `BLOB` and `CLOB`, have been added to `java.sql.Types`. These values are returned by methods such as `DatabaseMetaData.getTypeInfo()` and `DatabaseMetaData.getColumns()` when a JDBC driver supports these data types.

8.3 Arrays

8.3.1 Retrieving arrays

Data of type SQL array can be retrieved by calling the `getArray()` method of the `ResultSet` and `CallableStatement` interfaces. For example,

```
Array a = rs.getArray(1);
```

retrieves an `Array` value from the first column of the result set. By default, a JDBC driver should implement the `Array` interface using an SQL LOCATOR(array) internally. Also, by default `Array` objects only remain valid during the **transaction** in which they are created. These defaults may be changed as for the `Blob` and `Clob` types, but the JDBC API does not specify how this is done.

The `Array` interface provides several methods which return the contents of the array to the client as a materialized Java programming language array (Java array) or `ResultSet` object. These methods are `getArray()`and `getResultSet()`, respectively. See the separate API documentation for details.

8.3.2 Storing arrays

The `PreparedStatement.setArray()` method may be called to pass an `Array` value as an input parameter to a prepared statement. A Java array may be passed as an input parameter by calling `PreparedSatement.setObject()`.

8.3.3 Metadata additions

A new type code, `ARRAY`, has been added to `java.sql.Types`. This value is returned by methods such as `DatabaseMetaData.getTypeInfo()` and `DatabaseMetaData.getColumns()` when a JDBC driver supports the `Array` data type.

8.4 Refs

8.4.1 Retrieving refs

An SQL reference can be retrieved by calling the `getRef()` method of the `ResultSet` and `CallableStatement` interfaces. For example,

```
Ref ref = rs.getRef(1);
```

retrieves a `Ref` value from the first column of the result set. By default, retrieving a `Ref` value does not materialize the data to which the `Ref` refers. Also, by default a Ref value remains valid while the **session** or connection on which it is created is open. These defaults may be overridden, but again the JDBC API does not specify how this is done.

The `Ref` interface does **not** provide methods for dereferencing. Instead, a `Ref` can be passed as an input parameter to an appropriate SQL statement that fetches the object that it references. See the separate JDBC API documentation for details.

8.4.2 Storing refs

The `PreparedStatement.setRef()` method may be called to pass a `Ref` as an input parameter to a prepared statement.

8.4.3 Metadata additions

A new type code, `REF`, has been added to `java.sql.Types`. This value is returned by methods such as `DatabaseMetaData.getTypeInfo()` and `DatabaseMetaData.getColumns()` when a JDBC driver supports the `Ref` data type.

8.5 Distinct types

8.5.1 Retrieving distinct types

By default, a datum of SQL type DISTINCT is retrieved by calling any `getXXX()` method that is appropriate to the underlying type that the distinct type is based on. For example, given the following type declaration:

```
CREATE TYPE MONEY AS NUMERIC(10,2)
```

a value of type MONEY could be retrieved as follows:

```
java.math.BigDecimal bd = rs.getBigDecimal(1);
```

since the underlying SQL NUMERIC type is mapped to the `java.math.BigDecimal` type.

8.5.2 Storing distinct types

Any `PreparedStatement.setXXX()` method that is appropriate to the underlying type of an SQL DISTINCT type may be used to pass an input parameter of that distinct type to a prepared statement. For example, given the definition of type MONEY above `PreparedStatement.setBigDecimal()` would be used.

8.5.3 Metadata additions

A new type code, `DISTINCT`, has been added to `java.sql.Types`. This value is returned by methods such as `DatabaseMetaData.getTypeInfo()` and `DatabaseMetaData.getColumns()` when a JDBC driver supports this data type.

An SQL DISTINCT type must be defined as part of a particular database schema before it is used in a schema table definition. Information on schema-specific user-defined

types—of which `DISTINCT` types are one particular kind—can be retrieved by calling the `DatabaseMetaData.getUDTs()` method. For example,

```
int[] types = {Types.DISTINCT};
ResultSet rs = dmd.getUDTs("catalog-name", "schema-name",
        "%", types);
```

returns descriptions of all the SQL DISTINCT types **defined** in the `catalog-name.schema-name` schema. If the driver does not support UDTs or no matching UDTs are found then an empty result set is returned.

Each type description has the following columns:

TYPE_CAT	String => the type's catalog (may be null)
TYPE_SCHEM	String => the type's schema (may be null)
TYPE_NAME	String => the database type name
JAVA_CLASS	String => a Java class or interface name
DATA_TYPE	short => value defined in `java.sql.Types`, e.g. DISTINCT
REMARKS	String => explanatory comment on the type

Most of the columns above should be self-explanatory. The **TYPE_NAME** is the SQL type name given to the DISTINCT type—MONEY in the example above. This is the name used in a CREATE TABLE statement to specify a column of this type.

When **DATA_TYPE** is `Types.DISTINCT`, the **JAVA_CLASS** column contains a fully qualified Java class name. Instances of this class will be created if `getObject()` is called on a column of this DISTINCT type. For example, JAVA_CLASS would default to `java.math.BigDecimal` in the case of MONEY above. The JDBC API does not prohibit a driver from returning a subtype of the class named by JAVA_CLASS. The JAVA_CLASS value reflects a custom type mapping when one is used. See Chapter 9 for details.

8.6 Structured types

8.6.1 Retrieving structured types

A value of an SQL structured type is always retrieved by calling method `getObject()`. By default, `getObject()` returns a value of type `Struct` for a structured type. For example,

```
Struct struct = (Struct)rs.getObject(1);
```

retrieves a `Struct` value from the first column of the current row of result set `rs`. The `Struct` interface contains methods for retrieving the attributes of a structured type as an array of `java.lang.Object` values. By default, a JDBC driver should materialize the contents of a `Struct` prior to returning a reference to it to the application. Also, by default a `Struct` object is considered valid as long as the Java application maintains a reference to it. A JDBC driver may allow these defaults to be changed—to allow an SQL LOCATOR to be used, for example—but the JDBC API does not specify how this is done.

8.6.2 Storing structured types

The `PreparedStatement.setObject()` method may be called to pass a `Struct` as an input parameter to a prepared statement.

8.6.3 Metadata additions

A new type code, `STRUCT`, has been added to `java.sql.Types`. This value is returned by methods such as `DatabaseMetaData.getTypeInfo()` and `DatabaseMetaData.getColumns()` when a JDBC driver supports structured data types.

A structured SQL type must be defined as part of a particular database schema before it can be used in a schema table definition. Information on schema-specific user-defined types—of which `STRUCT` types are one particular kind—can be retrieved by calling the `DatabaseMetaData.getUDTs()` method. For example,

```
int[] types = {Types.STRUCT};
ResultSet rs = dmd.getUDTs("catalog-name", "schema-name",
        "%", types);
```

returns descriptions of all the structured SQL types **defined** in the `catalog-name.schema-name` schema. If the driver does not support UDTs or no matching UDTs are found then an empty result set is returned. See section 8.5.3 for a description of the result set returned by `getUDTs()`.

When the **DATA_TYPE** returned by `getUDTs()` is `Types.STRUCT`, the **JAVA_CLASS** column contains the fully qualified Java class name of a Java class. Instances of this class are manufactured by the JDBC driver when `getObject()` is called on a column of this STRUCT type. Thus, **JAVA_CLASS** defaults to `java.sql.Struct` for structured types. Chapter 9 discusses how this default can be modified by a Java application. We note here only that the JDBC API does not prohibit a driver from returning a subtype of the class named by JAVA_CLASS.

9 Customizing SQL Types

This chapter describes the support that the JDBC API provides for customizing the mapping of SQL structured and distinct types into Java classes. The customization mechanism involves minimal extensions to the JDBC API. The new functionality is an extension of the existing `getObject()` and `setObject()` mechanism.

9.1 The type mapping

An instance of `java.util.Map` is used to hold a custom mapping between SQL user-defined types—structured and distinct types—and Java classes. The `java.util.Map` interface is new in the JDK 1.2 and replaces `java.util.Dictionary`. Such an object is termed a *type-map* object. A type-map object implements a function from SQL names of user-defined types to objects of type `java.lang.Class`. A type-map object determines the class from which to construct an object to contain data of a given SQL user-defined type.

Each `Connection` has an associated type-map object. The type-map object contains type-mappings for translating data of SQL user-defined types in operations on that connection. Methods are provided for getting and setting a connection's type map. For example,

```
java.util.Map map = con.getTypeMap();
con.setTypeMap(map);
```

The `Connection.getTypeMap()` method returns the type-map object associated with a connection, while `Connection.setTypeMap()` can be used to set a new type mapping.

The mapping mechanism is quite flexible. If a connection's type mapping is not explicitly initialized by the JDBC application, then the default mappings described in Chapter 8 are used by operations on the connection. If a custom mapping is inserted into the type-map for SQL type `type-name`, then all operations on the connection will use this custom mapping for values of type `type-name`. Finally, we note that type-map objects may even be provided explicitly when calling certain `getXXX()` and `setXXX()` methods to override the custom or default mapping associated with a `Connection`.

9.2 Java class conventions

A Java class which appears in a custom type-map must implement a new interface—`java.sql.SQLData`. The `SQLData` interface contains methods that convert instances of SQL user-defined types to Java class instances, and vice versa. For example, the method `SQLData.readSQL()` reads a stream of data values and builds a Java object, while method `SQLData.writeSQL()` writes a sequence of values from a Java object to a stream. We anticipate that these methods will typically be generated by a tool which understands the database schema.

This stream-based approach for exchanging data between SQL and the Java programming language is conceptually similar to Java object Serialization. The data are read from and written to an SQL data stream provided by the JDBC driver. The SQL data stream may be implemented on various network protocols and data-formats. It may be implemented on any logical data-representation in which the leaf SQL data items (of which SQL structured types are composed) can be read from (written to) the data stream in a "depth-first" traversal of the structured types. That is, the attributes of an SQL structured type appear in the stream in the order in which they are declared in that type, and each (perhaps structured) attribute value appears fully (its structure recursively elaborated) in the stream before the next attribute. For data of SQL structured types that use inheritance, the attributes must appear in the stream in the order that they are inherited. That is, the attributes of a super-type must appear before attributes of a subtype. If multiple inheritance is used, then the attributes of super-types should appear in the stream in the order in which the super-types are listed in the type declaration. This protocol does not require the database server to have any knowledge of the Java programming language.

9.3 Streams of SQL data

This section describes the stream interfaces, SQLInput and SQLOutput, which support customization of the SQL to Java type mapping.

9.3.1 Retrieving data

When data of SQL structured and distinct types are retrieved from the database, they "arrive" in a stream implementing the `SQLInput` interface. The `SQLInput` interface contains methods for reading individual data values sequentially from the stream. The example below illustrates how a `SQLInput` stream can be used to provide values for the fields of an `SQLData` object. The `SQLData` object—the `this` object in the example—contains three persistent fields: a `String s`, a `Blob blob`, and an `Employee emp`.

```
this.str = sqlin.readString();
this.blob = sqlin.readBlob();
this.emp = (Employee)sqlin.readObject();
```

The `SQLInput.readString()` method reads a `String` value from the stream. The `SQLInput.readBlob() method` can be used to retrieve a `Blob` value from the stream. By default, the `Blob` interface is implemented using an SQL locator, so calling `readBlob()` doesn’t materialize the blob contents on the client. The `SQLInput.readObject()` method can be used to return an object reference from the stream. In the example, the `Object` returned is narrowed to an `Employee`.

There are a number of additional `readXXX()` methods defined on the `SQLInput` interface for reading each of the types. The `SQLInput.wasNull()` method can be called to check if the value returned by a `readXXX()` method was null.

9.3.2 Storing data

When an `SQLData` object is passed to a driver as an input parameter via a `setXXX()` method, the JDBC driver calls the object's `SQLData.writeSql()` method to obtain a stream representation of the contents of the object. Method `writeSQL()` writes data from the object to an `SQLOutput` stream as the representation of an SQL user-defined type. Method `writeSQL()` will typically have been generated by some tool from an SQL type definition. The example below illustrates use of the `SQLOutput` stream object.

```
sqlout.writeString(this.str);
sqlout.writeBlob(this.blob);
sqlout.writeObject(this.emp);
```

The example shows how the contents of an `SQLData` object can be written to an `SQLOutput` stream. The `SQLData` object—the `this` object in the example—contains three persistent fields: a `String s`, a `Blob blob`, and an `Employee emp`. Each field is written in turn to the `SQLOutput` stream, `sqlout`. The `SQLOutput` interface contains additional methods for writing each of the types defined in the JDBC API.

9.4 Examples

9.4.1 Example of SQL structured type

The following SQL example defines structured types PERSON, FULLNAME, and RESIDENCE. It defines tables with rows of types PERSON and RESIDENCE, and inserts a row into each, so that one row references another. Finally, it queries the table.

```
CREATE TYPE RESIDENCE
(
        DOOR NUMERIC(6),
        STREET VARCHAR(100),
        CITY VARCHAR(50),
        OCCUPANT REF(PERSON)
);

CREATE TYPE FULLNAME
(
        FIRST VARCHAR(50),
        LAST VARCHAR(50)
);

CREATE TYPE PERSON
(
        NAME FULLNAME,
```

```
        HEIGHT NUMERIC,
        WEIGHT NUMERIC,
        HOME REF(RESIDENCE)
);


CREATE TABLE HOMES OF RESIDENCE (OID REF(RESIDENCE)
        VALUES ARE SYSTEM GENERATED);

CREATE TABLE PEOPLE OF PERSON (OID REF(PERSON)
        VALUES ARE SYSTEM GENERATED);

INSERT INTO PEOPLE (SURNAME, HEIGHT, WEIGHT) VALUES
(
        FULLNAME('DAFFY', 'DUCK'),
        4,
        58
);

INSERT INTO HOMES (DOOR, STREET, CITY, OCCUPANT) VALUES
(
        1234,
        'CARTOON LANE',
        'LOS ANGELES',
        (SELECT OID FROM PEOPLE P WHERE P.NAME.FIRST = 'DAFFY')
);

UPDATE PEOPLE SET HOME = (SELECT OID FROM HOMES H WHERE
        H.OCCUPANT->NAME.FIRST = 'DAFFY') WHERE
        FULLNAME.FIRST = 'DAFFY'
```

The example above constructs three structured type instances, one each of types PERSON, FULLNAME, and RESIDENCE. A FULLNAME attribute is embedded in a PERSON. The PERSON and RESIDENCE instances are stored as rows of tables, and reference each other via Ref attributes.

The Java classes below represent the SQL structured types given above. We expect that such classes will typically be generated by a tool that reads the definitions of those structured types from the catalog tables, and, subject to customizations that a user of the tool may provide for name-mappings and type-mappings of primitive fields, will generate Java classes like those shown below.

Note: The JDBC API does not provide a standard API for accessing the metadata needed by a mapping tool. Providing this type of metadata introduces many subtle dependencies on the SQL99 type model, so it has been left out for now.

In each class below, method `SQLData.readSQL()` reads the attributes in the order that they appear in the definition of the corresponding structured types in the database (i.e., in "row order, depth-first" order, where the complete structure of each attribute is read, recursively, before the next attribute is read). Similarly, `SQLData.writeSQL()` writes the data to the stream in that order.

```
public class Residence implements SQLData {
    public int door;
    public String street;
    public String city;
    public Ref occupant;

    private String sql_type;
    public String getSQLTypeName() { return sql_type; }

    public void readSQL (SQLInput stream, String type)
        throws SQLException {
      sql_type = type;
      door = stream.readInt();
      street = stream.readString();
      city = stream.readString();
      occupant = stream.readRef();
    }

    public void writeSQL (SQLOutput stream) throws SQLException {
      stream.writeInt(door);
      stream.writeString(street);
      stream.writeString(city);
      stream.writeRef(occupant);
    }
}

public class Fullname implements SQLData {
    public String first;
    public String last;

    private String sql_type;
    public String getSQLTypeName() { return sql_type; }

    public void readSQL (SQLInput stream, String type)
        throws SQLException {
      sql_type = type;
      first = stream.readString();
      last = stream.readString();
    }

    public void writeSQL (SQLOutput stream) throws SQLException {
      stream.writeString(first);
      stream.writeString(last);
    }
}

public class Person implements SQLData {
    Fullname name;
    float height;
    float weight;
    Ref home;
```

```
    private String sql_type;
    public String getSQLTypeName() { return sql_type; }

    public void readSQL (SQLInput stream, String type)
        throws SQLException {
      sql_type = type;
      name = (Fullname)stream.readObject();
      height = stream.readFloat();
      weight = stream.readFloat();
      home = stream.readRef();
    }

    public void writeSQL (SQLOutput stream)
        throws SQLException {
      stream.writeObject(name);
      stream.writeFloat(height);
      stream.writeFloat(weight);
      stream.writeRef(home);
    }
}
```

The following method uses those classes to materialize data from the tables HOMES and PEOPLE that were defined earlier:

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

public void demo (Connection con) throws SQLException {

  // setup mappings for the connection
  try {
    java.util.Map map = con.getTypeMap();
    map.put("S.RESIDENCE", Class.forName("Residence"));
    map.put("S.FULLNAME", Class.forName("Fullname"));
    map.put("S.PERSON", Class.forName("Person"));
  }
  catch (ClassNotFoundException ex) {}

  PreparedStatement pstmt;
  ResultSet rs;

  pstmt = con.prepareStatement("SELECT OCCUPANT FROM HOMES");
  rs = pstmt.executeQuery();
  rs.next();
  Ref ref = rs.getRef(1);

  pstmt = con.prepareStatement(
            "SELECT FULLNAME FROM PEOPLE WHERE OID = ?");
  pstmt.setRef(1, ref);
  rs = pstmt.executeQuery();
```

```
    rs.next();
    Fullname who = (Fullname)rs.getObject(1);

    // prints "Daffy Duck"
    System.out.println(who.first + " " + who.last);
  }
```

9.4.2 Mirroring SQL inheritance in the Java programming language

SQL structured types may be defined to form an inheritance hierarchy. For example, consider SQL type STUDENT that inherits from PERSON:

```
CREATE TYPE PERSON AS OBJECT (NAME VARCHAR(20), BIRTH DATE);

CREATE TYPE STUDENT AS OBJECT EXTENDS PERSON (GPA NUMERIC(4,2));
```

The following Java classes can represent data of those SQL types. Class `Student` extends `Person`, mirroring the SQL type hierarchy. Methods `SQLData.readSQL()` and `SQLData.writeSQL()` of the subclass cascades each call to the corresponding method in its super-class, in order to read or write the super-class attributes before reading or writing the subclass attributes.

```
import java.sql.*;
...
public class Person implements SQLData {
  public String name;
  public Date birth;

  private String sql_type;
  public String getSQLTypeName() { return sql_type; }

  public void readSQL (SQLInput data, String type)
      throws SQLException {
    sql_type = type;
    name = data.readString();
    birth = data.readDate();
  }

  public void writeSQL (SQLOutput data)
      throws SQLException {
    data.writeString(name);
    data.writeDate(birth);
  }
}

public class Student extends Person {
  public float GPA;

  private String sql_type;
  public String getSQLTypeName() { return sql_type; }
```

```
    public void readSQL (SQLInput data, String type)
        throws SQLException {
      sql_type = type;
      super.readSQL(data, type);
      GPA = data.readFloat();
    }

    public void writeSQL (SQLOutput data)
    throws SQLException {
      super.writeSQL(data);
      data.writeFloat(GPA);
    }
  }
```

The Java class hierarchy need not mirror the SQL inheritance hierarchy. For example, class `Student` above could have been declared without a super-class. In this case, `Student` could contain fields to hold the inherited attributes of the SQL type STUDENT as well as the attributes declared by STUDENT itself..

9.4.3 Example mapping of SQL distinct type

An SQL distinct type, MONEY, and a Java class Money that represents it:

```
-- SQL definition
CREATE TYPE MONEY AS NUMERIC(10,2);

// definition
public class Money implements SQLData {

  public java.math.BigDecimal value;

  private String sql_type;
  public String getSQLTypeName() { return sql_type; }

  public void readSQL (SQLInput stream, String type)
        throws SQLException {
    sql_type = type;
    value = stream.readBigDecimal();
  }

  public void writeSQL (SQLOutput stream) throws SQLException {
    stream.writeBigDecimal(value);
  }
}
```

9.5 Generality of the approach

Users have great flexibility to customize the Java classes that represent SQL structured and distinct types. They control the mappings of built-in SQL attribute types to Java

field types. They control the mappings of SQL names (of types and attributes) to Java names (of classes and fields). Users may add (to Java classes that represent SQL types) fields and methods that implement domain-specific functionality. Users can generate beans as the classes that represent SQL types.

A user can even map a single SQL type to different Java classes, depending on arbitrary conditions. To do that, the user must customize the implementation of `SQLData.readSQL()` to construct and return objects of different classes under different conditions.

Similarly, the user can map a single SQL value to a graph of Java objects. Again, that is accomplished by customizing the implementation of `SQLData.readSQL()` to construct multiple objects and distribute the SQL attributes into fields of those objects.

A customization of the `SQLData.readSQL()` method could populate the type-map object incrementally. And so on. We believe that these kinds of flexibility will allow users to map SQL types appropriately for different kinds of applications.

9.6 NULL data

An application uses the existing `getObject()` and `setObject()` mechanism to retrieve and store `SQLData` values. We note that when the second parameter, `x`, of method `PreparedStatement.setObject()` has the value `null`, then the driver executes the SQL statement as if the SQL literal NULL had appeared in place of that parameter of the statement:

```
void setObject (int i, Object x) throws SQLException;
```

When parameter `x` is null, there is no enforcement that the corresponding argument expression is of a Java type that could successfully be passed to that SQL statement if its value were non-null. The Java programming language null carries no type information. For example, a null Java programming language variable of class `AntiMatter` could be passed as an argument to an SQL statement that requires a value of SQL type `MATTER`, and no error would result, even though the relevant type-map object did not permit the translation of `MATTER` to `AntiMatter`.

9.7 Summary

Chapters 8 and 9 presented extensions to support new categories of SQL types. The extensions have these properties:

- All of the new SQL types are handled with uniform and extensible interfaces, which may be staged into the JDBC API piecemeal.
- Minimal mechanism is added to the API. An implementation does little more than transfer control to methods `SQLData.readSQL()` and `SQLData.writeSQL()` of classes that have been generated to represent the SQL types.

- The extensions are based on existing interfaces `java.io.Serializable`, `java.io.DataInput`, `java.io.DataOutput`, `java.sql.ResultSet`, and `java.sql.PreparedStatement`
- Great flexibility is given to writers of database tools to add value by customizing the Java classes that represent SQL data.

10 Other New Features and Changes

This chapter describes additional changes that have been made in the JDBC API.

10.1 Changes to java.sql.ResultSet

A version of the `ResultSet.getBigDecimal()` method that returns full precision has been added.

10.2 Changes to java.sql.ResultSetMetaData

The `ResultSetMetaData.getColumnType()` method may now return the new SQL type codes: `STRUCT`, `DISTINCT`, `BLOB`, etc. The `STRUCT` and `DISTINCT` type codes are always returned for structured and distinct values, independent of whether the default or a custom type mapping is being used.

The `ResultSetMetaData.getColumnTypeName()` method should return the following for the new SQL types.

Column Type	Column Type Name
JAVA_OBJECT	the SQL name of the Java type
DISTINCT	the SQL name of the distinct type
STRUCT	the SQL name of the structured type
ARRAY	data source dependent type name
BLOB	data source dependent type name
CLOB	data source dependent type name
REF	data source dependent type name

A `ResultSetMetaData.getColumnClassName()` method has been added to return the fully qualified name of the Java class whose instances are manufactured if ResultSet.getObject() is called to retrieve a value from the column. See the separate API documentation for details.

The `ResultSetMetaData.getColumnTypeName()` method returns a fully qualified SQL type name when the type code is STRUCT, DISTINCT, or JAVA_OBJECT.

10.3 Changes to DatabaseMetaData

The `DatabaseMetaData.getColumns()` method may now return DATA_TYPE values of the new SQL99 types: BLOB, CLOB, etc. The `DatabaseMetaData.getColumns()` method returns the same type names as those listed in Section 10.2 for the SQL99 data types.

Added method `DatabasemetaData.getConnection()` to return the `Connection` object that produced the metadata object.

Added method `DatabasemetaData.getUDTs()`. See the separate API documentation for details.

Added methods to support the new `ResultSet` and batch update functionality: `supportsResultSetConcurrency()`, `supportsBatchUpdates()`, etc. See the separate API documentation for details.

10.4 Changes to java.sql.DriverManager

A `DriverManager.setLogWriter()` method that takes a `java.io.PrintWriter` object as input has been added. A new `DriverManager.getLogWriter()` method returns a `PrintWriter` object. The `set/getLogStream()` methods have been deprecated.

10.5 Date, Time, and Timestamp

The JDBC API follows the Java platform's approach of representing dates and times as a millisecond value relative to January 1, 1970 00:00:00 GMT. Since most databases don't support the notion of a time zone, the JDBC 2.1 core API adds new methods to allow a JDBC driver to get/set `Date`, `Time`, and `Timestamp` values for a particular time zone using a `Calendar`. For example,

```
ResultSet rs;
...
Date date1 = rs.getDate(1);
```

returns a `Date` object that wraps a millisecond value which denotes a particular date, like January 3, 1999, and a normalized time 00:00:00 in the default time zone. The time component of the Date is set to zero in the default time zone since SQL DATE values don't have a time component. Since a `Calendar` was not supplied explicitly to `getDate()`, the default time zone (really the default `Calendar`) is used by the JDBC driver internally to create the appropriate millisecond value assuming that the underlying database doesn't store time zone information.

The following example retrieves a date value in GMT—Greenwich Mean Time.

```
ResultSet rs;
...

TimeZone.setDefault(TimeZone.getTimeZone("GMT"));
Calendar cal = Calendar.getInstance();
Date date2 = rs.getDate(1, cal);
```

In the example above, a `Calendar` is passed explicitly to `getDate()` to inform the JDBC driver how to calculate the appropriate millisecond value. Note that the same result could have been achieved by simply changing the default time zone, and not pass-

ing the `Calendar` explicitly since the JDBC driver will use the default time zone by default.

Note that the two `Date` objects created above will not compare as equal assuming that the default time zone is not GMT, even if they represent the ‘same’ date.

```
if (date1.equals(date2))
        //never get here
```

This is because each Java language `Date` object really just wraps a normalized millisecond time value and these millisecond values will differ across time zones. If an application wishes to compare dates in different time zones it should first convert them to a `Calendar`.

An application should create a `Date` object using a `Calendar`. The application is responsible for specifying the time as 00:00:00 on the desired date when using the `Calendar` since the JDBC API uses this convention. In addition when creating a `Time` value the application must specify a date of January 1, 1970 to the `Calendar` used to create the millisecond value for the `Time` as this is the convention specified for time.

10.6 Refinement to the *Drop Table* requirement

The JDBC 1.0 API specification required that JDBC 1.0 compliant drivers provide full support for the SQL92, Transitional Level, DROP TABLE command, including full support for the CASCADE and RESTRICT options of DROP TABLE. Because some popular databases currently do not fully support DROP TABLE as it is defined in the SQL92 specification, this requirement has been relaxed in the JDBC API specification.

A JDBC compliant driver is required to support the DROP TABLE command as specified by SQL92, Transitional Level. However, support for the CASCADE and RESTRICT options of DROP TABLE is optional for a JDBC compliant driver. In addition, the behavior of DROP TABLE is implementation defined when there are views or integrity constraints defined that reference a table that is being dropped.

11 Clarifications

We have gotten several requests to clarify some aspects of the JDBC API. This chapter contains additional explanation of some features.

11.1 Connection.isClosed()

The `Connection.isClosed()` method is only guaranteed to return true after `Connection.closed()` has been called. `Connection.isClosed()` cannot be called, in general, to determine if a database connection is valid or invalid. A typical client can determine that a connection is invalid by catching the exception that is thrown when an operation is attempted.

11.2 Statement.setCursorName()

The `Statement.setCursorName()` method provides a way for an application to specify a cursor name for the cursor associated with the next result set produced by a statement. A result set's cursor name can be retrieved by calling `ResultSet.getCursorName()`. If `Statement.setCursorName()` is called prior to creating a result set, then `ResultSet.getCursorName()` should always return the value specified in `Statement.setCursorName()`.

We note that calling `Statement.setCursorName()` prior to creating a result set does not mean that the result set is updatable, in other words, positioned update or delete may not be allowed on a result set even if `Statement.setCursorName()` was called. By default, a result set is read-only.

The only use for a cursor name is to embed it in a SQL statement of the form

```
UPDATE ... WHERE CURRENT OF <cursor>
```

The cursor name provides a way to do a positioned update or delete. To enable positioned update and delete on a result set, a select query of the form

```
SELECT FOR UPDATE ... FROM ... WHERE ...
```

should be used to create the result set. If `Statement.setCursorName()` is not called to specify a cursor name, then the JDBC driver or underlying DBMS must generate a cursor name when a `SELECT FOR UPDATE` statement is executed, if positioned update/delete is supported. `ResultSet.getCursorName()`should return `null` if the result set is read-only and `Statement.setCursorName()` was not called to specify a cursor name.

11.3 Character conversion

JDBC driver implementations are expected to automatically convert the Java programming language unicode encoding of strings and characters to and from the character en-

coding of the database being accessed. The JDBC API does not define how to override the character encoding of a database. For example, the API does not define how to store unicode characters in an ASCII database.

11.4 Streams as input parameters

When an application passes a stream as an input value via a setXXX() or updateXXX() method, the application is responsible for maintaining the stream in a readable state until one of the following methods is called: PreparedStatement.execute(), executeQuery(), executeUpdate(), or executeBatch(), and ResultSet.insertRow() or updateRow(). A JDBC driver is not required to wait until one of these methods is called to read the stream value.

11.5 Result sets not created by a `Statement`

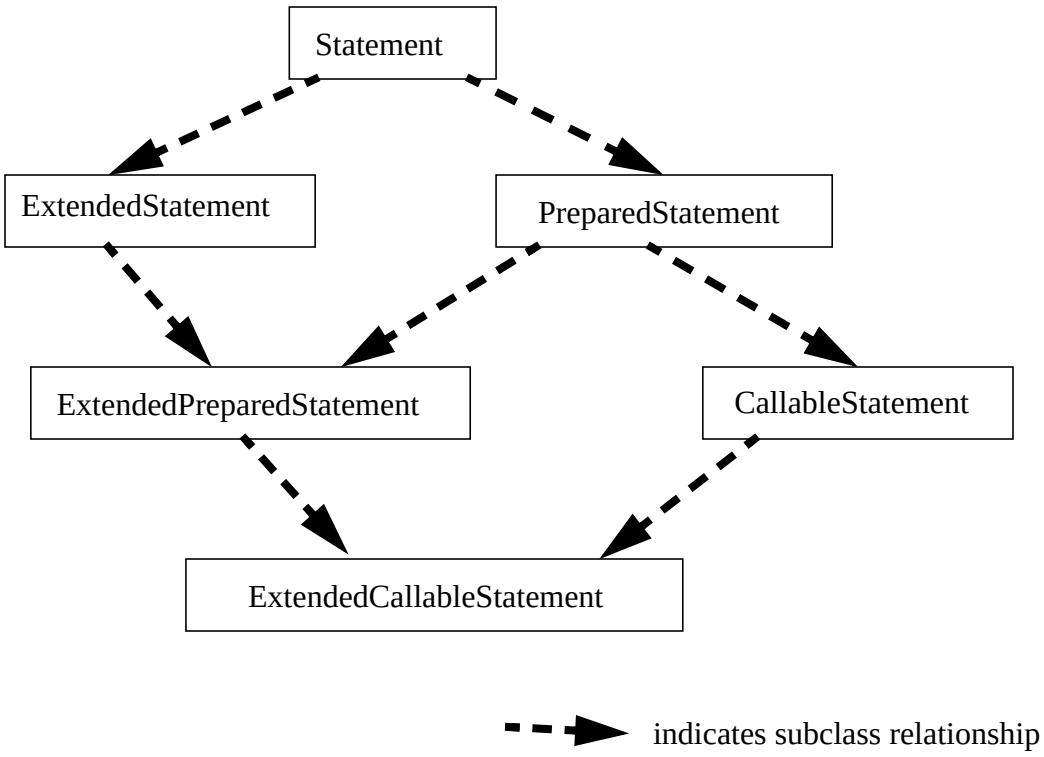
A `ResultSet` object that is created by a metadata operation is only required to be forward-only. Scrollability is not required for result sets produced by `DatabaseMetaData` operations.

Appendix A: Rejected Design Choices

A.1 Design Alternative: Create new subtypes of the existing java.sql types.

We have chosen to add new methods to existing interfaces and classes in order to add new functionality to the JDBC API.

An alternative design, which we considered, was instead to extend the existing java.sql types with new subtypes that contained any new methods. This approach was judged to be too unwieldy. For example, since we needed to add methods to the `java.sql.Statement` interface, such as `Statement.setFetchSize()`, it would have been necessary to create three new statement interfaces related to the old interfaces in a fairly complex inheritance hierarchy (see below). This approach seemed to add too much complexity to the JDBC API.



Appendix B: Additional Suggestions

This section contains a list of some of the suggested additions that we received that have **not** been added to the JDBC API. They are listed simply as a record of some of the things that have been left out. We welcome input concerning the items listed here.

B.1 Other Suggestions for v0.1

Processing Results - Add a way to determine the value of an auto-incremented key after an insert is done.

Enhanced Meta-Data - Add meta-data for prepared statement parameters and for prepared statement result set. The JDBC API does not provide metadata describing a prepared statements parameters; and, it does not provide metadata describing its results without executing the statement (equivalent to SQL92 DESCRIBE OUTPUT and ODBC SQLDescribeParam.

New Data Types - Add the SQL time interval data type

Security - Allow the application to choose underlying transport properties, e.g., SSL. Provide SSL Socket implementation. A JDBC technology based application must be able to select driver-supported mechanisms for securing the wirelevel protocol (e.g., encryption). Relative to SSL, one option is to allow the application to specify a specific Cyphersuite (key-exchange algorithm, bulk-encryption, MAC [message authentication algorithm])

Security-Authentication - Allow an application to select driver-supported mechanisms for performing authentication. The following mechanisms should be supported: Username, password; Kerberos token; Digital Certificates

Command Complete event - Support a user provided event object that is fired when a Command completes (both current command and regular command, sync or async). - - Various events: Connect Event; Disconnect Event; Before Connect Event

Cursor Implementation Location - Support client-side vs server-side.

Parameter Management - Support: Append, GetCount, GetItem (by name/position), Delete and Refresh.

Hybrid SQL/Java programming language Integration - Provide a mechanism for defining Java classess and provide a hybrid SQL/Java query mechanism over Tables whose columns may be SQL atomics or Java classes.

Specialty Data Types - Provide extensions for OLAP, Spatial, TimeSeries and other Specialty Data Types.

Serializing data, time, timestamps - Allow these types to be serializable.

Async Requests - Allow the caller to request that a Statement execute asynchronously.

Java classes - Introduce the notion of a SQL specialization of Java classes/beans that introduces SQL99 concepts useful for dealing with Java objects in the context of databases and business applications. For example, it is useful for a database system to un-

derstand which method(s) definitions in a Class may be used to perform operations on objects such as comparisons, etc. One approach would be to introduce "generic" method names. Those could also be used outside of the database by regular Business Applications.

Add support for SQL PSM.

Add additional SQL language functionality e.g. various forms of join.

Add *levels* of JDBC compatibility, as opposed to individual API calls to see if individual features are supported by a drvier.

Add an API call that describes the format of the URL understood by a driver.

Add a row object that encapsulates database data in its native format.

Add immutability for Date, Time, Timestamp.

B.2 Additional suggestions for v0.7

- It was suggested that as an alternative to providing individual methods for each new property on the `Statement` interface such as result set type, concurrency type, etc., we could introduce a new Class, `ResultSetProperties`, that itself contained all methods for getting and setting these properties. Statement would then just contain two new methods for getting and setting a `ResultSetProperties` property. This approach would help to simplify the `Statement` interface.
- Add a new CursorResultSet interface which extends ResultSet and adds method `CursorStatement prepareUpdate() throws SQLException`; OR add the `prepareUpdate` method to the existing `ResultSet` interface - and define that it may fail if there is no cursor associated with it. Add a new `CursorStatement` interface which extends `PreparedStatement` and adds the methods: `void update() throws SQLException` and `void delete() throws SQLException`. It would be helpful to introduce another intermediate `CursorResultSet` which would sit in the interitance hierarchy between `ResultSet` and `ScrollableResultSet`. The motivation for the CursorStatement is to avoid the need to parse every query to look for cursor operations. The reason to have CursorStatement extend PreparedStatement is to get access to the setXXX methods. The setXXX() methods would be used to provide new values to the corresponding columns of the current row in the CursorResultSet. The executeUpdate() method would perform the actual update (with the parameter values that had been set) or delete (parameter values are ignored). The execute() and executeQuery() methods would be overridden so that they always throw SQLException. Additionally, the CursorStatement would be "bound" to the CursorResultSet which created it such that whenever that CursorResultSet was repositioned (next, relative, absolute, first, last, etc.) that the CursorStatement would automatically track this and update/delete methods affect the right row. ** It is tempting to do away with the CursorStatement and just add that functionality to the CursorResultSet because these things are likely to be in 1:1 relationship.

- Add to the `Statement` interface: `void setCursorProperties(CursorProperties props) throws SQLException;` Define a new `java.sql.CursorProperties` class. I like the idea of adding statement properties so that appropriate subclasses of `ResultSet` are returned when the statement is executed. But, rather than adding a bunch of individual accessors/mutators for all these properties to `Statement`, I would recommend defining a `CursorProperties` class with public members and then add just a single new method to `Statement`: `void setCursorProperties(CursorProperties props) throws SQLException;` This would remove the need for the new `Statement.setFetchSize` and `getFetchSize` methods for example. `CursorProperties` would have a public constructor which returns a `CursorProperties` object with well defined default values (TBD). Members of this class would include: 0. boolean useCursors - if true the statement should return a `CursorResultSet` from executeQuery. 1. String cursorName - `Statement.setCursorName()` would be depricated. 2. boolean scrollable - if true the statement should return a `ScrollableResultSet` from execute query. 3. boolean readonly - if true this cursor is READONLY. 4. int rowCacheSize - hint to driver on how many rows to retrieve from the database at a time. 5. boolean closeOnEndTransaction - in ANSI SQL '92 cursors are automatically closed on commit/rollback, but many databases allow cursors to remain open for efficiency 6. String[] updatableColumns - list of columns which in SQL '92 grammar would be in the "FOR UPDATE OF <column list>" clause. 7. boolean sensitive - if true committed changes to the underlying tables which happened while the cursor was open may be seen by the application as it scrolls over those rows.
- The result of any SQL query can be thought of as defining a simple structured type. The fields of the structured type correspond to the columns of the query result, and each row in the result set returned by the query represents an instance of the type. The JDBC API could allow a mapping from a type that is defined implicitly by an SQL query to a Java class, or even a bi-directional mapping between regular relational tables and Java classes.

Appendix C: Change History

C.1 Changes between 0.10 and 0.70:

- Removed references to the `java.sql2` package. Substituted a proposal that splits the JDBC API into two packages: `javax.sql` and `java.sql`.
- Removed descriptions of the classes and intefaces in `javax.sql`. We plan to add them in again later, or create a separate document for the extended JDBC API.
- Removed the `ScrollableResultSet` interface and associated methods on other interfaces. Added methods for scrolling to the `ResultSet` interface
- Added result set type, concurrency type, keyset size, fetch size, and fetch direction properties to `Connection`, `Statement`, and `ResultSet` interfaces.
- Deprecated Statement.setCursorName and ResultSet.getCursorName. Use of “SELECT FOR UPDATE” is also deprecated. The new result set update methods can be used instead.
- Added SQL99 APIs.
- Added new metadata for persistent Java objects.

C.2 Changes between 0.70 and 0.80

- The Struct interface no longer extends the SQLData interface.
- All occurrences of the SQLType interface have been removed and replaced with 'String'.
- The Array.getArray() method now returns Object instead of Object[]. This allows an int[] array to be returned, for example.
- The specification now states that a Java programming language array may be passed as an input parameter via PreparedStatement.setObject().
- The semantics of ResultSet.isDeleted(), etc. is clarified.
- The DatabaseMetaData.getClass() method has been dropped due to the difficulties involved in loading classes directly from a database.
- The ResultSet.getBigDecimal() and CallableStatement.getBigDecimal() methods which take an ‘int scale’ parameter have been deprecated.
- Statement.getResultSetType() now throws SQLException
- Added method Array.getBaseTypeName(). Array.getBaseType() now returns an int type code.
- DatabaseMetaData.getUDTs() now allows a fully qualified SQL name which may contain wildcard characters in the typeNamePattern parameter.

C.3 Changes between 0.80 and 0.90

- Section 9.4: Compiled code examples and removed syntax errors.
- Section 9.4.2: Added note on the independence of the Java programming language and SQL99 inheritance hierarchies.
- Section 6.1: Removed the requirement that indirect updates, such as those performed by a trigger that is fired, be included in the update counts returned by a batch update.
- Section 5.6: Clarified some technical details concerning inserting new rows in a result set.
- Section 7.3.1: Noted that the TYPE_NAME returned by getTypeInfo() may contain a vendor specific type name when the type code is JAVA_ODBECT. Previously, the TYPE_NAME had to be null.
- Removed method SQLInput.readStruct() and SQLOutput.writeStruct(). Read/writeObject() should be used instead.
- Chapter 5: Revised and simplified the scrollable result set model.
- Section 11.3: Added note on character conversion.
- Section 10.3: Added description of time zone support.

C.4 Changes between 0.90 and 0.91

- Section 5.5: Added rules for selecting a supported result set type and concurrency type.
- Section 5.6: Added method ResultSet.cancelRowUpdates.
- Section 5.6: Added more description on the semantics of the update methods.
- Added Section 5.9.

C.5 Changes between 0.91 and 0.95

- Section 5.6: Added general description of the queries that yield updatable result sets.
- Section 8.2: Clarified rules for passing blob and clob values as input parameters.
- Sections 11.4 and 11.5: Added these sections to the document.

C.6 Changes between 0.95 and 1.0

- no changes

C.7 Changes between 1.0 and 1.1

- Chapter 6: Extended the semantics of executeBatch. Batch processing is now allowed to continue after an element in a batch has failed to execute successfully.

- Added Section 10.6.

Appendix D: Motivation for the SQL99 Proposal

The following observations and requirements have shaped this proposal:

- A Java technology program will retrieve a value of an SQL type as a single Java programming language data item. For example, an instance of a structured SQL type can be materialized as a single object, by a single method call (e.g., to method getObject()).
- A Java program will retrieve an SQL value as a "strongly typed" Java object. For example, a structured type instance will be materialized as an object of a class with fields or methods that correspond to the attributes of that particular structured type, and that allow the program to access the attributes of the structured type.
- A client-side Java program can retrieve values of user-defined SQL types from a database that is "unaware" of Java technology. No Java programming language support is required in that database. For example, no special definitions must be made in the database to allow structured type data to be retrieved by Java programs.
- In particular, structured type values are retrieved into Java programs by means of SELECT statements; are posted to the database by INSERT and UPDATE statements; and so on.
- The SQL methods of a structured type execute in the server, and are not relevant to the issue of retrieving structured type data from SQL into Java programs.
- Definitions of user-defined SQL types are visible in SQL catalog tables. Similarly, definitions of Java classes are programmatically available thru the reflection API. Therefore, tools that generate Java class definitions for SQL data, or vice versa, can be written by database tools vendors, independent of any particular relational database vendor. Our mappings of SQL types to Java classes must permit third-parties to write such tools.
- We assume that the mappings between SQL user-defined types any Java classes will usually be generated by tools. It is not necessary to design the mappings so that human programmers can easily write them "by hand".
- There is no reason to define exactly one mapping of an SQL type to a class as the only "correct" mapping. Mapping tools may define different Java classes, corresponding to a given SQL user-defined type, to serve different application domains or execution environments. Such different classes may have different type mappings for the primitive attributes, or auxilliary fields that are relevant to a particular application. Our mappings of SQL user-defined types to Java classes should allow such variations.
- We want to support Java programs that fetch data of SQL user-defined types in a "dynamic SQL" style, without "knowing" in advance the number and kinds of attributes of those data.

- An SQL structured type may contain an attribute of a REF(structured-type) type that references a row containing another structured type instance. Despite a superficial similarity, that SQL REF is not analogous to a field containing another lightweight Java object. An SQL REF is a key value that designates a row of a table; it does not designate an object in transient memory.
- Users will often want a 1-1 mapping between SQL types and Java classes. Java class-to-SQL mapping tools will generate a class for each distinct type, for each structured type, for each REF(structured-type), for each array type, for each Locator type, and so on.
- We want to allow flexibility in the mappings of SQL types to Java classes, so that tools builders, and implementers of domain-specific "application service layers" can add value by customizing the Java classes that implement SQL types. In particular, the mapping of leaf SQL attributes and elements to fields and variables permits all conversions that the JDBC API currently permits. SQL names may be mapped to Java programming language names in arbitrary ways. An SQL value may be represented by one Java object, or by a graph of objects.