JDBC™ 4.0 Specification

JSR 221

Lance Andersen, Specification Lead
December 14, 2005

PUBLIC DRAFT

Specification: JSR-221, JDBC 4.0 API Specification ("Specification")

Status: Pre-FCS, Public Review

Release: 14 December 2005

Copyright 2005 Sun Microsystems, Inc.

4150 Network Circle, Santa Clara, California 95054, U.S.A

All rights reserved.

Please Recycle

Adobe PostScript™

You will hold Sun (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND: If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Specification and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT: You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS: Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

Neither party may assign or otherwise transfer any of its rights or obligations under this Agreement, without the prior written consent of the other party, except that Sun may assign this Agreement to an affiliated company.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

(Sun.pSpec.license.11.14.2003)

Sun Microsystems, Inc.
www.sun.com

Contents

Preface

This document supersedes and consolidates the content of these predecessor specifications:

- "JDBC: A Java SQL API"
- "JDBC 2.1 API"
- "JDBC 2.0 Standard Extension API"
- "JDBC 3.0 Specification"

This document introduces a range of new features for the JDBC API and is combined with various specification improvements that focus on features introduced in or before the JDBC 3.0 API. Where possible, any adjustment to the JDBC 3.0 API is marked for easy identification - look for the JDBC 4.0 API demarcation for specific features introduced in this revised and updated specification.

Readers can also download the API specification (Javadoc^{TM} API and comments) for a complete and precise definition of JDBC classes and interfaces. This documentation is available from the download page at

http://java.sun.com/products/jdbc

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Submitting Feedback

Please send any comments and questions concerning this specification to:

`jsr-221-comments@jcp.org`

Introduction

1.1 The JDBC API

The JDBC™ API provides programmatic access to relational data from the Java™ programming language. Using the JDBC API, applications written in the Java programming language can execute SQL statements, retrieve results, and propagate changes back to an underlying data source. The JDBC API can also be used to interact with multiple data sources in a distributed, heterogenous environment.

The JDBC API is based on the X/Open SQL CLI, which is also the basis for ODBC. JDBC provides a natural and easy-to-use mapping from the Java programming language to the abstractions and concepts defined in the X/Open CLI and SQL standards.

Since its introduction in January 1997, the JDBC API has become widely accepted and implemented. The flexibility of the API allows for a broad range of implementations.

1.2 Platforms

The JDBC API is part of the Java platform, which includes the Java™ Standard Edition (Java™ SE) and the Java™ Enterprise Edition (Java™ EE). The JDBC 4.0 API is divided into two packages: `java.sql` and `javax.sql`. Both packages are included in the Java SE and Java EE platforms.

1.3 Target Audience

This specification is targeted primarily towards the vendors of these types of products:

- drivers that implement the JDBC API
- application servers providing middle-tier services above the driver layer
- tools that use the JDBC API to provide services such as application generation

This specification is also intended to serve the following purposes:

- an introduction for end-users whose applications use the JDBC API
- a starting point for developers of other APIs layered on top of the JDBC API

1.4 Acknowledgements

The JDBC 4.0 specification work is being conducted as part of JSR-221 under the Java Community Process. This specification is the result of the collaborative efforts involving many individuals,. We would like to thank the following members of the expert group for their invaluable contributions to this document:

Lance Andersen, Sun Microsystems (Specification Lead)

Mark Biamonte, DataDirect Technologies

P.G. Binod, Sun Microsystems

Volker Berlin

Paul Bonfanti

Christopher Farrar, IBM

John Goodson, DataDirect Technologies

Fei Luo, BEA Systems

Ashish Mahahan, Sybase

Mark Matthews, MySQL Inc.

Sudhanshu Pant, Nokia Corporation

Ara Aravamudhan, Datamirror

Goals

2.1 History

The JDBC API is a mature technology, having first been specified in January 1997. In its initial release, the JDBC API focused on providing a basic call-level interface to SQL databases. The JDBC 2.1 specification and the 2.0 Optional Package specification then broadened the scope of the API to include support for more advanced applications and for the features required by application servers to manage use of the JDBC API on behalf of their applications.

The JDBC 3.0 specification operated with the stated goal to round out the API by filling in smaller areas of missing functionality. With JDBC 4.0, our goals are two fold: Improve the Ease-of-Development experience for all developers working with SQL in the Java platform. Secondly, provide a range of enterprise level features to expose JDBC to a richer set of tools and APIs to manage JDBC resources.

2.2 Overview of Goals

The following list outlines the goals and design philosophy for the JDBC API in general and the JDBC 4.0 API in particular:

1. Fit into the Java EE and Java SE platforms

   The JDBC API is a constituent technology of the Java platform. The JDBC 4.0 API should be aligned with the overall direction of the Java 2 Enterprise Edition and Java 2 Standard Edition platforms. In addition, recent developments with the Java 5.0 platform have exposed a range of new features and language improvements that are extensively used in this specification.

2. Be consistent with SQL:2003

The JDBC API provides programmatic access from applications written in the Java programming language to standard SQL. JDBC 3.0 sought to ensure it's support for a subset of the SQL:99 features that were likely to be widely supported by the industry. Similarly for JDBC 4.0, support for SQL:2003 is focused on the major components of this specification that we anticipate will be supported for the foreseeable future.

3. Consolidate predecessor specifications

This document incorporates content from four prior JDBC specifications to provide a single standalone specification of the JDBC API.

4. Offer vendor-neutral access to common features

The JDBC API strives to provide high-bandwidth access to features commonly supported across different vendor implementations. The goal is to provide a degree of feature access comparable to what can be achieved by native applications. However, the API must be general and flexible enough to allow for a wide range of implementations.

5. Maintain the focus on SQL

The focus of the JDBC API has always been on accessing relational data from the Java programming language. This goal, previously stated in the JDBC 3.0 API remains core to the principles on which this specification is built. The provision of ease of development themed improvements including APIs and utilities continue to focus on the needs for the SQL based software development from the Java platform. Similarly to previous specifications, this does not preclude interacting with additional technologies such as XML, CORBA and non-relational data.

6. Provide a foundation for tools and higher-level APIs

The JDBC API presents a standard API to access a wide range of underlying data sources or legacy systems. Implementation differences are made transparent through JDBC API abstractions, making it a valuable target platform for tools vendors who want to create portable tools and applications.

Because it is a "call-level" interface from the Java programming language to SQL, the JDBC API is also suitable as a base layer for higher-level facilities such as Enterprise JavaBeans (EJB) 3.0 container-managed persistence, SQLJ and the JDBC RowSet implementation.

7. Keep it simple

The JDBC API is intended to be a simple-to-use, straight forward interface upon which more complex entities can be built. This goal is achieved by defining many compact, single-purpose methods instead of a smaller number of complex, multipurpose ones with control flag parameters.

8. Enhance reliability, availability, and scalability

   Reliability, availability, and scalability are the themes of the Java EE and Java SE platforms, as well as the direction for future Java platforms. The JDBC 4.0 API stays true to these themes by enhancing support in several areas, including resource management, the reuse of prepared statements across logical connections, and error handling.

9. Maintain backward compatibility with existing applications and drivers

   Existing JDBC technology-enabled drivers (JDBC drivers) and the applications that use them must continue to work in an implementation of the Java virtual machine that supports the JDBC 4.0 API. Applications that use only features defined in earlier releases of the JDBC API will not require changes to continue running. It should be straightforward for existing applications to migrate to JDBC 4.0 technology.

10. Close Association with JDBC RowSet implementations

    J2SE 5.0 contains a standard JDBC RowSet implementation as specified in JDBC RowSet Implementations (JSR-114). This specification will provide a set of utilities described at both the utility class level and the Meta Data language level. This will allow developers to easily migrate JDBC-technology enabled applications towards the JDBC RowSet model that enables disconnected data source access in addition to the ability to manage relational data stores from an XML stand-point.

11. Allow forward compatibility with Connectors

    The Connector architecture defines a standard way to package and deploy a resource adapter that allows a Java EE container to integrate its connection, transaction, and security management with those of an external resource. The JDBC 4.0 API provides the migration path for JDBC drivers to the Connector architecture. It should be possible for vendors whose products use JDBC technology to move incrementally towards implementing the Connector API. The expectation is that JDBC driver vendors will write resource manager wrappers around their existing data source implementations so that they can be reused in a Connector framework.

12. Specify requirements unambiguously

    The requirements for JDBC compliance need to be unambiguous and easy to identify. The JDBC 4.0 specification and the API documentation (Javadoc) will clarify which features are required and which are optional.

Summary of New Features

3.1 Overview of changes

The JDBC 4.0 API introduces new material and changes in the following areas:

- Automatic loading of `java.sql.Driver`

  `DriverManager.getConnection` has been modified to utilize the Java SE
  Service Provider mechanism to automatically load JDBC Drivers. This removes
  the need to invoke `Class.forName`.

- Ease of Development API

  Added various standard JDBC Annotations and support for generic DataSet to
  ease SQL interactions between a Java application and a SQL data source.

- ROWID data type

  Added the data type `java.sql.RowId`. This allows JDBC programs to access
  a SQL ROWID.

- National Character Set Conversion Support

  The JDBC types `NCHAR`, `NVARCHAR`, `LONGNVARCHAR`, and `NCLOB` have been
  added. The methods `setNString`, `setNCharacterStream`, `setNClob` have
  been added to the `PreparedStatement` interface and the methods
  `getNClob`, `getNString`, `getNCharacterStream` were added to the
  `CallableStatement` and `ResultSet` interface. The methods `updateNClob`,
  `updateNString`, `updateNCharacterStream` were added to the
  `ResultSet` interface.

- Enhanced Support for `BLOB` and `CLOB`

  Additional methods have been added to the `Connection` interface, allowing
  for the creation of a `BLOB`, `CLOB` and `NCLOB` object. The `PreparedStatement`
  interface has had methods added to insert `BLOB` object using an InputStream,

and to insert `CLOB`, `NCLOB` objects using a `Reader` object. The `Blob`, `Clob` and `NClob` interfaces have added a `free` method that can be used to release resources that these objects hold.

■ SQL/XML and XML Support

SQL:2003 introduces the notion of how to express XML data in a SQL data store. Additional APIs have been added to allow applications access to this data.

■ Wrapper Pattern

Added the ability to unwrap implementation of JDBC classes to make use of non-standard JDBC methods provided by vendor implementations.

■ `SQLException` Enhancements

Support for Java SE chained Exceptions has been added. SQLExceptions now support the Iterable interface allowing for their use in a for-each loop. Two categories of SQLExceptions have been added: `SQLTransientException` and `SQLNonTransientExeption`. Each category provides new `SQLException` subclasses that map to common SQLState class values.

■ Connection Management

Additional enhancements to `Connection` and `Statement` interfaces to permit improved connection state tracking, and greater flexibility when managing Statement objects in pool environments.

■ JDBC API changes

The following changes were made to existing JDBC interfaces.

■ `Connection`

Added the methods `createBlob`, `createClob`, `createNClob`, `createQueryObject`, `isValid`, `createSQLXML`, `getClientInfo`, `setClientInfo`.

■ `CallableStatement`

Added the methods `getRowId`, `setRowId`, `getNClob`, `getNString`, `getCharacterStream`, `getNCharacterStream`, `setNString`, `setNCharacterStream`, `setNClob`, `getSQLXML`, `setSQLXML`. Overloaded the `setClob` and `setBlob` methods.

■ `DatabaseMetaData`

Added the methods `getRowIdLifetime`, `autoCommitFailureClosesAllResultSets`, `getFunctions`, `getFunctionParameters`, `providesQueryObjectGenerator`, `getClientInfoProperties` and `supportsStoredFunctionsUsingCallSyntax`. Overloaded the method `getSchemas`.

■ `PreparedStatement`

Added the methods `setRowId`, `setNString`, `setNCharacterStream`, `setSQLXML`, `isPoolable`, `setPoolable` and `setNClob`. Overloaded the methods `setClob` and `setBlob`.

■ `ResultSet`

Added the methods `getHoldability`, `getRowId`, `updateRowId`, `getNClob`, `isClosed`, `getNString`, `getNCharacterStream`, `updateNCharacterStream`, `updateNString`, `getSQLXML`, `updateSQLXML` and `updateNClob`. Modified `updateObject`.

■ `Statement`

Added the methods `isClosed` and `getResultSetHoldability`.

■ `DataSource`

Added the method `createQueryObject`.

■ `PooledConnection`

Added the methods `addStatementEventListener` and `removeStatementEventListener`.

■ `SQLInput`

Added the methods `readNClob`, `readNString`, `readRowId`, and `readSQLXML`.

■ `SQLOutput`

Added the methods `writeNClob`, `writeNString`, `writeRowId`, and `writeSQLXML`.

See Chapter 5 "Classes and Interfaces" for a list of the classes and interfaces affected by these changes.

Overview

The JDBC API provides a way for Java programs to access one or more sources of data. In the majority of cases, the data source is a relational DBMS, and its data is accessed using SQL. However, it is also possible for JDBC technology-enabled drivers to be implemented on top of other data sources, including legacy file systems and object-oriented systems. A primary motivation for the JDBC API is to provide a standard API for applications to access a wide variety of data sources.

This chapter introduces some of the key concepts of the JDBC API. In addition, it describes two common environments for JDBC applications, with a discussion of how different functional roles are implemented in each one. The two-tier and three-tier models are logical configurations that can be implemented on a variety of physical configurations.

4.1 Establishing a Connection

The JDBC API defines the `Connection` interface to represent a connection to an underlying data source.

In a typical scenario, a JDBC application will connect to a target data source using one of two mechanisms:

- `DriverManager` — this fully implemented class was introduced in the original JDBC 1.0 API. When an application first attempts to connect to a data source by specifying a URL, `DriverManager` will automatically load any JDBC 4.0 drivers found within the CLASSPATH (any drivers that are pre-JDBC 4.0 must be explicitly loaded by the application).

- `DataSource` — this interface was introduced in the JDBC 2.0 Optional Package API. It is preferred over `DriverManager` because it allows details about the underlying data source to be transparent to the application. A `DataSource` object's properties are set so that it represents a particular data source. When its `getConnection` method is invoked, the `DataSource` instance will return a

connection to that data source. An application can be directed to a different data source by simply changing the `DataSource` object's properties; no change in application code is needed. Likewise, a `DataSource` implementation can be changed without changing the application code that uses it.

The JDBC API also defines two important extensions of the `DataSource` interface to support enterprise applications. These extensions are the following two interfaces:

- `ConnectionPoolDataSource` — supports caching and reusing of physical connections, which improves application performance and scalability
- `XADataSource` — provides connections that can participate in a distributed transaction

4.2 Executing SQL Statements and Manipulating Results

Once a connection has been established, an application using the JDBC API can execute queries and updates against the target data source. The JDBC API provides access to the most commonly implemented features of SQL:2003. Because different vendors vary in their level of support for these features, the JDBC API includes the `DatabaseMetadata` interface. Applications can use this interface to determine whether a particular feature is supported by the data source they are using. The JDBC API also defines escape syntax to allow an application to access non-standard vendor-specific features. The use of escape syntax has the advantage of giving JDBC applications access to the same feature set as native applications and at the same time maintaining the portability of the application.

Applications use methods in the `Connection` interface to specify transaction attributes and create `Statement`, `PreparedStatement`, or `CallableStatement` objects. These statements are used to execute SQL statements and retrieve results. The `ResultSet` interface encapsulates the results of an SQL query. Statements may also be batched, allowing an application to submit multiple updates to a data source as a single unit of execution.

The JDBC API extends the `ResultSet` interface with the `RowSet` interface, thereby providing a container for tabular data that is much more versatile than a standard result set. A `RowSet` object is a JavaBeans™ component, and it may operate without being connected to its data source. For example, a `RowSet` implementation can be serializable and therefore sent across a network, which is particularly useful for small-footprint clients that want to operate on tabular data without incurring the overhead of a JDBC driver and data source connection. Another feature of a `RowSet` implementation is that it can include a custom reader for accessing any data in tabular format, not just data in a relational database. Further, a `RowSet` object can

update its rows while it is disconnected from its data source, and its implementation can include a custom writer that writes those updates back to the underlying data source.

4.2.1 Support for SQL Advanced Data Types

The JDBC API defines standard mappings to convert SQL data types to JDBC data types and back. This includes support for SQL:2003 advanced data types such as `BLOB`, `CLOB`, `ARRAY`, `REF`, `STRUCT`, `XML`, and `DISTINCT`. JDBC drivers may also implement one or more customized type mappings for user-defined types (UDTs), in which the UDT is mapped to a class in the Java programming language. The JDBC API also provides support for externally managed data, for example, data in a file outside the data source.

4.3 Two-tier Model

A two-tier model divides functionality into a client layer and a server layer, as shown in FIGURE 4-1.
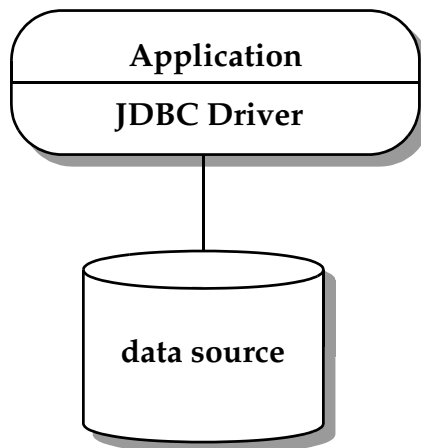
FIGURE 4-1   Two-tier Model

The client layer includes the application(s) and one or more JDBC drivers, with the application handling these areas of responsibility:

- presentation logic

- business logic

- transaction management for multiple-statement transactions or distributed transactions

- resource management

In this model, the application interacts directly with the JDBC driver(s), including establishing and managing the physical connection(s) and dealing with the details of specific underlying data source implementations. The application may use its knowledge of a specific implementation to take advantage of nonstandard features or do performance tuning.

Some drawbacks of this model include:

- mingling presentation and business logic with infrastructure and system-level functions. This presents an obstacle to producing maintainable code with a well-defined architecture.

- making applications less portable because they are tuned to a particular database implementation. Applications that require connections to multiple databases must be aware of the differences between the different vendors' implementations.

- limiting scalability. Typically, the application will hold onto one or more physical database connections until it terminates, limiting the number of concurrent applications that can be supported. In this model, issues of performance, scalability and availability are handled by the JDBC driver and the corresponding underlying data source. If an application deals with multiple drivers, it may also need to be aware of the different ways in which each driver/data source pair resolves these issues.

4.4 Three-tier Model

The three-tier model introduces a middle-tier server to house business logic and infrastructure, as shown in FIGURE 4-2.
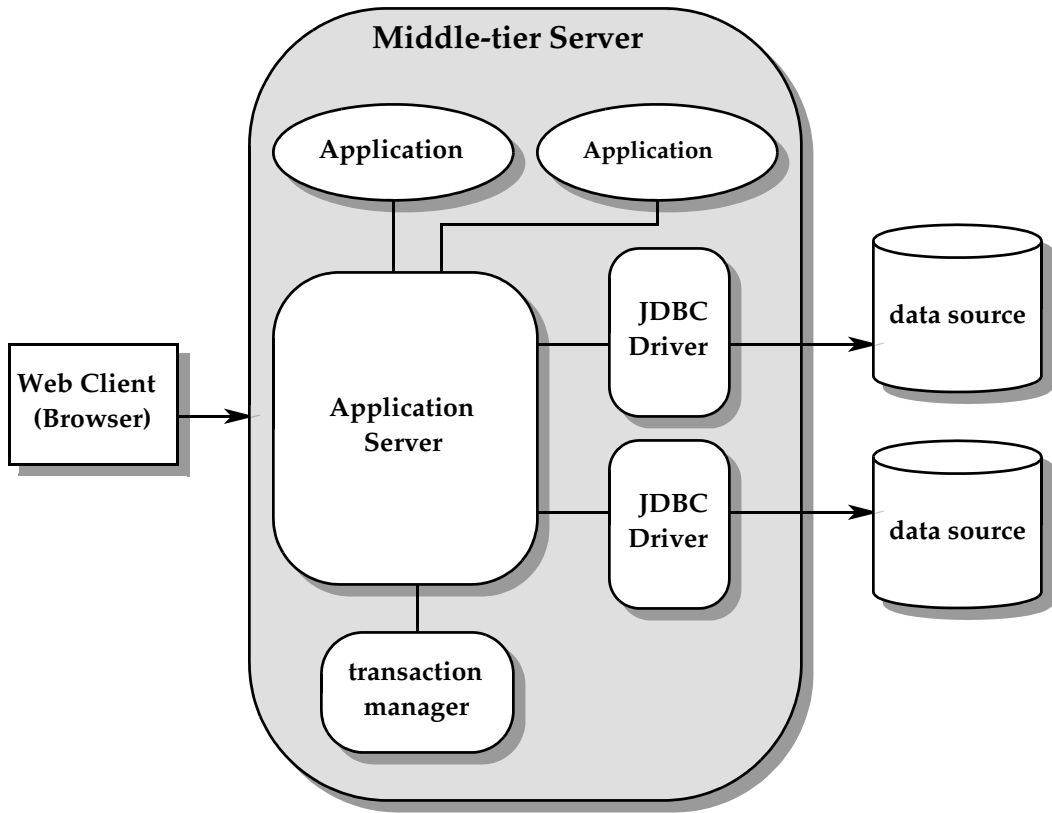
FIGURE 4-2   Three-tier Model

This architecture is designed to provide improved performance, scalability and availability for enterprise applications. Functionality is divided among the tiers as follows:

1. Client tier — a thin layer implementing presentation logic for human interaction. Java programs, web browsers and PDAs are typical client-tier implementations. The client interacts with the middle-tier application and does not need to include any knowledge of infrastructure or underlying data source functions.

2. Middle-tier server — a middle tier that includes:

   ■ *Applications* to interact with the client and implement business logic. If the application includes interaction with a data source, it will deal with higher-level abstractions, such as DataSource objects and logical connections rather than lower-level driver API.

- *An application server* to provide supporting infrastructure for a wide range of applications. This can include management and pooling of physical connections, transaction management, and the masking of differences between different JDBC drivers. This last point makes it easier to write portable applications. The application server role can be implemented by a Java EE server. Application servers implement the higher-level abstractions used by applications and interact directly with JDBC drivers.

- *JDBC driver(s)* to provide connectivity to the underlying data sources. Each driver implements the standard JDBC API on top of whatever features are supported by its underlying data source. The driver layer may mask differences between standard SQL:2003 syntax and the native dialect supported by the data source. If the data source is not a relational DBMS, the driver implements the relational layer used by the application server.

3. Underlying data source — the tier where the data resides. It can include relational DBMSs, legacy file systems, object-oriented DBMSs, data warehouses, spreadsheets, or other means of packaging and presenting data. The only requirement is a corresponding driver that supports the JDBC API.

4.5 JDBC in the Java EE Platform

Java EE components, such as JavaServer™ Pages, Servlets, and Enterprise Java Beans™ (EJB™) components, often require access to relational data and use the JDBC API for this access. When Java EE components use the JDBC API, the container manages their transactions and data sources. This means that Java EE component developers do not directly use the JDBC API's transaction and datasource management facilities. See the Java EE Platform Specification for further details.

Classes and Interfaces

The following classes and interfaces make up the JDBC API.

5.1 The `java.sql` Package

The core JDBC API is contained in the package `java.sql`. The annotations, enums, classes and interfaces in `java.sql` are listed below. Annotations, enums and classes are bold type; interfaces are in standard type.

java.sql.AutoGeneratedKeys

java.sql.Array

java.sql.BaseQuery

java.sql.BatchUpdateException

java.sql.Blob

java.sql.CallableStatement

java.sql.ClientInfoException

java.sql.Clob

java.sql.ConflictingRow

java.sql.Connection

java.sql.DataSet

java.sql.DataTruncation

java.sql.DatabaseMetaData

java.sql.Date

java.sql.DataSetResolver

java.sql.Driver

java.sql.DriverManager

java.sql.DriverPropertyInfo

java.sql.NClob

java.sql.ParameterMetaData

java.sql.PreparedStatement

java.sql.QueryObjectFactory

java.sql.QueryObjectGenerator

java.sql.Ref

java.sql.ResultColumn

java.sql.ResultSet

java.sql.ResultSetMetaData

java.sql.RowId

java.sql.RowIdLifeTime

java.sql.Savepoint

java.sql.Select

java.sql.SQLData

java.sql.SQLDataException

java.sql.SQLDataSetSyncException

java.sql.SQLException

java.sql.SQLFeatureNotSupportedException

java.sql.SQLInput

java.sql.SQLIntegrityConstraintViolationException

java.sql.SQLInvalidAuthorizationSpecException

java.sql.SQLNonTransientConnectionException

java.sql.SQLNonTransientException

java.sql.SQLOutput

java.sql.SQLPermission

java.sql.SQLRuntimeException

java.sql.SQLSyntaxErrorException

java.sql.SQLTimeoutException

java.sql.SQLTransactionRollbackException

java.sql.SQLTransientConnectionException

java.sql.SQLTransientException

java.sql.SQLXML

java.sql.SQLWarning

java.sql.Statement

```
java.sql.Struct
```
java.sql.Time

java.sql.Timestamp

java.sql.Types

java.sql.Update
```
java.sql.Wrapper
```

The following classes and interfaces are either new or updated in the JDBC 4.0 API. New classes and interfaces are highlighted in bold.

java.sql.AutoGeneratedKeys

java.sql.BaseQuery
```
java.sql.Blob
```
```
java.sql.CallableStatement
```
java.sql.ClientInfoException
```
java.sql.Clob
```
java.sql.ConflictingRow
```
java.sql.Connection
```
java.sql.DataSetResolver
```
java.sql.DatabaseMetaData
```
java.sql.NClob
```
java.sql.PreparedStatement
```
java.sql.QueryObjectFactory

java.sql.QueryObjectGenerator

java.sql.ResultColumn
```
java.sql.ResultSet
```
java.sql.RowId

java.sql.RowIdLifeTime

java.sql.Select

java.sql.SQLDataException

java.sql.SQLDataSetSyncException
```
java.sql.SQLException
```
java.sql.SQLFeatureNotSupportedException
```
java.sql.SQLInput
```
java.sql.SQLIntegrityConstraintViolationException

java.sql.SQLInvalidAuthorizationSpecException

java.sql.SQLNonTransientConnectionException

java.sql.SQLNonTransientException

java.sql.SQLOutput

java.sql.SQLRuntimeException

java.sql.SQLSyntaxErrorException

java.sql.SQLTimeoutException

java.sql.SQLTransactionRollbackException

java.sql.SQLTransientConnectionException

java.sql.SQLTransientException

java.sql.SQLXML

java.sql.SQLWarning

java.sql.Statement

java.sql.Types

java.sql.Update

java.sql.Wrapper

javax.sql.CommonDataSource

javax.sql.StatementEvent

javax.sql.StatementEventListener

FIGURE 5-1 shows the interactions and relationships between the key classes and interfaces in the java.sql package. The methods involved in creating statements, setting parameters and retrieving results are also shown.
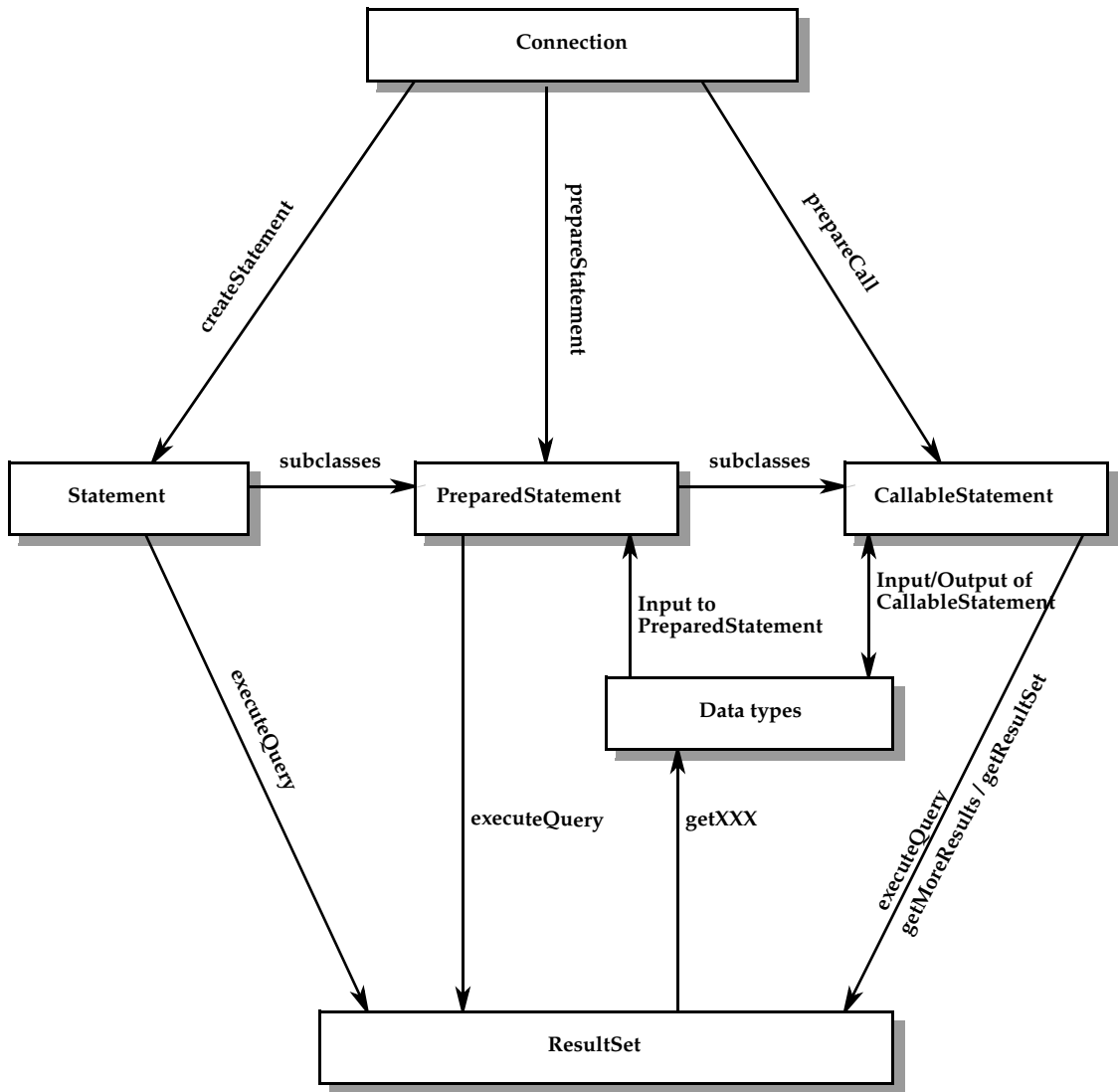
FIGURE 5-1   Relationships between major classes and interface in the `java.sql` package

5.2 The `javax.sql` Package

The following list contains the classes and interfaces that are contained in the `javax.sql` package. Classes are highlighted in bold; interfaces are in normal type.

```
javax.sql.CommonDataSource
```
`javax.sql.ConnectionEvent`
```
javax.sql.ConnectionEventListener
```
```
javax.sql.ConnectionPoolDataSource
```
```
javax.sql.DataSource
```
```
javax.sql.PooledConnection
```
```
javax.sql.RowSet
```
`javax.sql.RowSetEvent`
```
javax.sql.RowSetInternal
```
```
javax.sql.RowSetListener
```
```
javax.sql.RowSetMetaData
```
```
javax.sql.RowSetReader
```
```
javax.sql.RowSetWriter
```
`javax.sql.StatementEvent`
```
javax.sql.StatementEventListener
```
```
javax.sql.XAConnection
```
```
javax.sql.XADataSource
```

Note – The classes and interfaces in the `javax.sql` package were first made available as the JDBC 2.0 Optional Package. This optional package was previously separate from the `java.sql` package, which was part of J2SE 1.2. Both packages (`java.sql` and `javax.sql`) are now part of Java SE as of J2SE 1.4.

FIGURE 5-2, FIGURE 5-3, FIGURE 5-4, and FIGURE 5-5 show the relationships between key classes and interfaces in these areas of functionality: `DataSource` objects, connection pooling, distributed transactions, and rowsets.
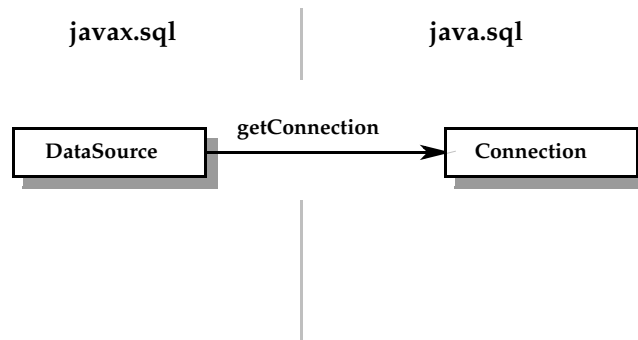
FIGURE 5-2 Relationship between `javax.sql.DataSource` and `java.sql.Connection`
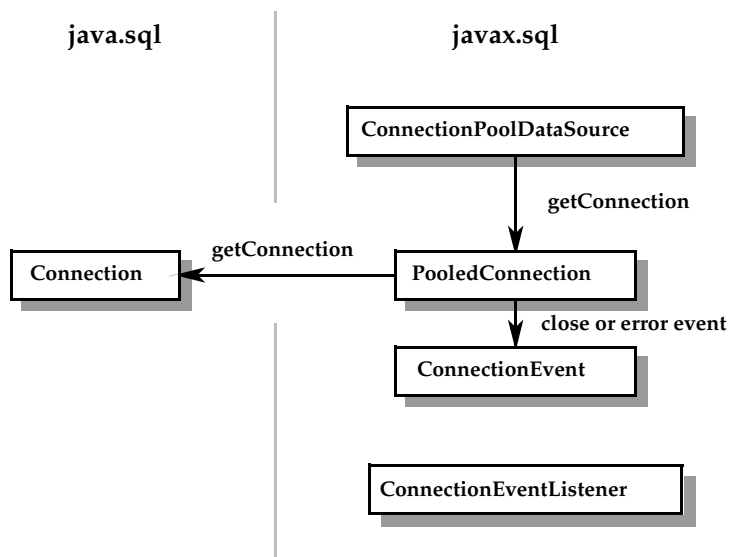
FIGURE 5-3 Relationships involved in connection pooling
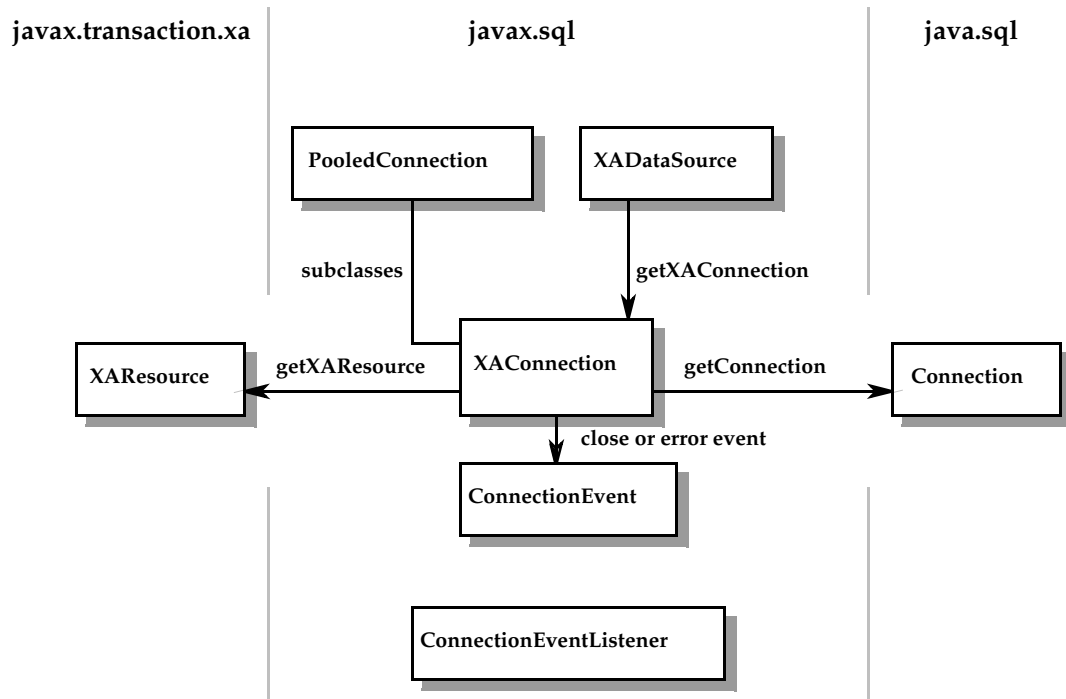
FIGURE 5-4   Relationships involved in distributed transaction support
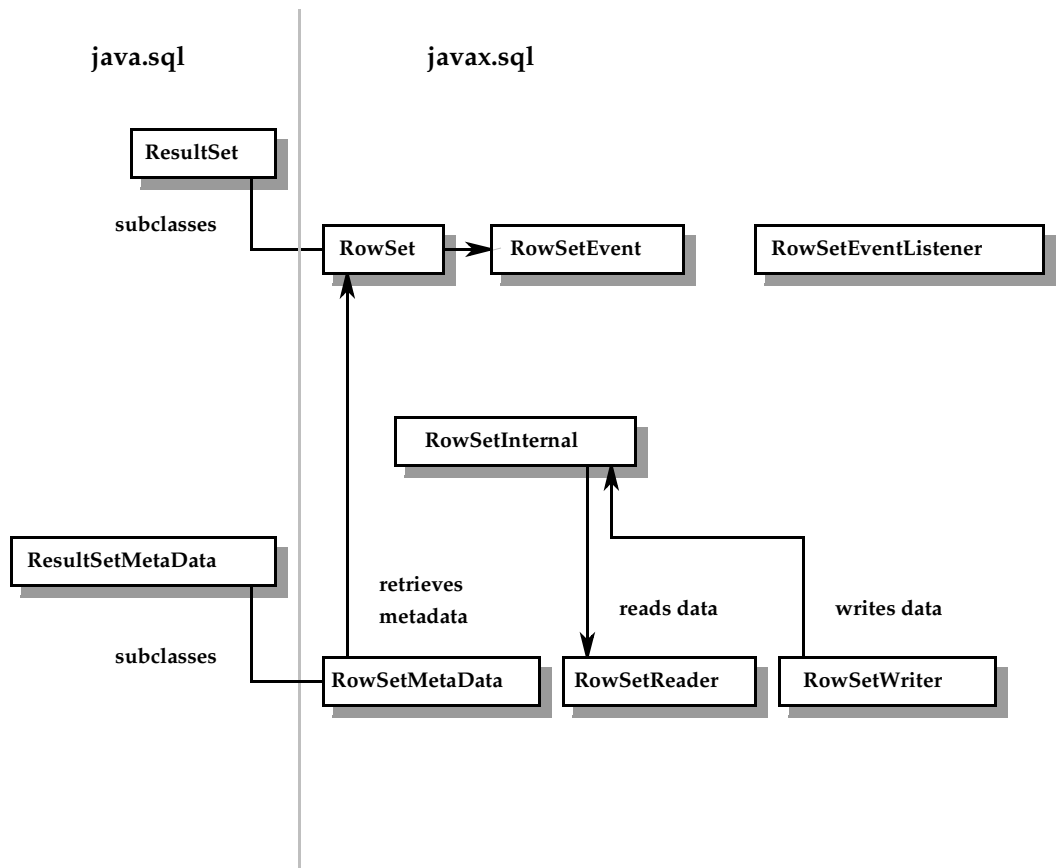
FIGURE 5-5 `RowSet` relationships

Compliance

This chapter identifies the features that a JDBC API implementation is required to support for each level of compliance. A JDBC API implementation includes a JDBC technology-enabled driver and its underlying data source. Therefore, compliance is defined in terms of what features are available above the driver layer.

Any features not identified here are optional. In general, a driver is not required to implement any feature that its underlying data source does not support.

6.1 Definitions

To avoid ambiguity, we will use these terms in our discussion of compliance:

- **JDBC API implementation** — a JDBC technology-enabled driver and its underlying data source. The driver may provide support for features that are not implemented by the underlying data source. It may also provide the mapping between standard syntax/semantics and the native API implemented by the data source.

- **Relevant specifications** — this document, the API specification, and the relevant SQL specification. This is also the order of precedence if a feature is described in more than one of these documents. For the JDBC 1.0 API, the relevant SQL specification is SQL92 and X/Open SQL CLI. For the JDBC 2.0 and 3.0 APIs, it is SQL92 plus the relevant sections of SQL99 and X/Open SQL CLI. For the JDBC 4.0 API, it is SQL92 plus the relevant sections of SQL:2003 and X/Open SQL CLI.

- **Supported feature** — a feature for which the JDBC API implementation supports standard syntax and semantics for that feature as defined in the relevant specifications.

- **Extension** — a feature that is not covered by any of the relevant specifications or a non-standard implementation of a feature that is covered.

- **Fully implemented** — a term applied to an interface that has all of its methods implemented to support the semantics defined in the relevant specifications.
- **Required interface** — an interface that must be included although it might not be fully implemented. Methods that are not implemented should throw an `SQLException` to indicate that the corresponding feature is not supported.

6.2 Guidelines and Requirements

The following guidelines apply to all levels of compliance:

- A JDBC API implementation must support Entry Level SQL92 plus the SQL command `Drop Table` (see note.)

    Entry Level SQL92 represents a "floor" for the level of SQL that a JDBC API implementation must support. Access to features based on SQL99 or SQL:2003 should be provided in a way that is compatible with the relevant part of the SQL99 or SQL:2003 specification.

- Drivers must support escape syntax. Escape syntax is described in Chapter 13 "Statements".
- Drivers must support transactions. See Chapter 9 "Transactions" for details.
- Drivers should provide access to every feature implemented by the underlying data source, including features that extend the JDBC API. When a feature is not supported, the corresponding methods throw an `SQLFeatureNotSupportedException`. The intent is for applications using the JDBC API to have access to the same feature set as native applications.
- If a `DatabaseMetaData` method indicates that a given feature is supported, it must be supported via standard syntax and semantics as described in the relevant specifications. This may require the driver to provide the mapping to the data source's native API or SQL dialect if it differs from the standard.
- If a feature is supported, all of the relevant metadata methods must be implemented. For example, if a JDBC API implementation supports the `RowSet` interface, it must also implement the `RowSetMetaData` interface.
- If a feature is not supported, the corresponding `DatabaseMetaData` method must say so. Attempting to access the unsupported feature causes an `SQLException` to be thrown.

> **Note –** A JDBC API implementation 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. In addition, the behavior of DROP TABLE is implementation-defined when there are views or integrity constraints defined that reference the table being dropped.

6.3 JDBC 1.0 API Compliance

A driver that is compliant with the JDBC 1.0 API must do the following:

- Adhere to the preceding guidelines and requirements
- Fully implement the following interfaces:
  - `java.sql.Driver`
  - `java.sql.DatabaseMetaData` (excepting those methods introduced in subsequent JDBC API releases)
  - `java.sql.ResultSetMetaData` (excepting those methods introduced in subsequent JDBC API releases)
- Include the following required interfaces:
  - `java.sql.CallableStatement`
  - `java.sql.Connection`
  - `java.sql PreparedStatement`
  - `java.sql.ResultSet`
  - `java.sql.Statement`

6.4 JDBC 2.0 API Compliance

A driver that is compliant with the JDBC 2.0 API must do the following:

- Comply with the JDBC 1.0 API requirements
- Implement the following additional `DatabaseMetaData` methods:
  - `deletesAreDetected`
  - `getConnection`
  - `getUDTs`
  - `insertsAreDetected`

- othersDeletesAreVisible
- othersInsertsAreVisible
- othersUpdatesAreVisible
- ownDeletesAreVisible
- ownInsertsAreVisible
- ownUpdatesAreVisible
- supportsBatchUpdates
- supportsResultSetConcurrency
- supportsResultSetType
- updatesAreDetected
- Implement the following additional `ResultSetMetaData` methods:
  - getColumnClassName
  - getColumnType
  - getColumnTypeName

6.5   JDBC 3.0 API Compliance

A driver that is compliant with the JDBC 3.0 API must do the following:

- Comply with the JDBC 2.0 API requirements
- Include the following required interfaces:
  - `java.sql.ParameterMetaData`
  - `java.sql.Savepoint`
- It must implement the following additional `DatabaseMetaData` methods:
  - supportsSavepoints
  - supportsNamedParameters
  - supportsMultipleOpenResults
  - supportsGetGeneratedKeys
  - getSuperTypes
  - getSuperTables
  - getAttributes
  - getResultSetHoldability
  - supportsResultSetHoldability
  - getSQLStateType

- getDatabaseMajorVersion
- getDatabaseMinorVersion
- getJDBCMajorVersion
- getJDBCMinorVersion

6.6 JDBC 4.0 API Compliance

A driver that is compliant with the JDBC 4.0 API must do the following:

- Comply with the JDBC 3.0 API requirements
- Include the following required interfaces
  - java.sql.BaseQuery
  - java.sql.RowId
  - java.sql.QueryObjectGenerator
  - java.sql.StatementEvent
  - java.sql.StatementEventListener
  - java.sql.DataSet<T>
  - java.sql.SQLXML
- Include the following required annotations
  - java.sql.AutoGeneratedKeys
  - java.sql.ResultColumn
  - java.sql.Select
  - java.sql.Update
- Include the following required enums
  - java.sql.GeneratedKeys
  - java.sql.RowIdLifetime
- It must implement the following additional Connection methods
  - createQueryObject
  - isValid
  - getClientInfo
  - setClientInfo
- It must implement the following additional DataSource method
  - createQueryObject
- It must implement the following additional DatabaseMetaData methods:

- getRowIdLifetime
- supportsStoredFunctionsUsingCallSyntax
- providesQueryObjectGenerator
- getSchemas(String catalog, String schemaPattern)
- autoCommitFailureClosesAllResultSets
- getClientInfoProperties

- It must implement the following additional `ResultSet` methods
  - getHoldability
  - isClosed

6.7 Determining Compliance Level

The JDBC API is a constituent technology of the Java platform. Compliance with the JDBC API specification is determined as a subset of evaluating compliance with the overall platform.

Note – As of this writing, there is no separate evaluation of compliance level for the JDBC API.

6.8 Deprecated APIs

Deprecation refers to a class, interface, constructor, method or field that is no longer recommended and may cease to exist in a future version.

The following constructors and methods were deprecated in the JDBC 2.0 API:

java.sql.CallableStatement.getBigDecimal(int, int)

java.sql.Date(int, int, int)

java.sql.Date.getHours()

java.sql.Date.getMinutes()

java.sql.Date.getSeconds()

java.sql.Date.setHours(int)

java.sql.Date.setMinutes(int)

java.sql.Date.setSeconds(int)

java.sql.DriverManager.getLogStream()

java.sql.DriverManager.setLogStream(PrintStream)

java.sql.PreparedStatement.setUnicodeStream(int, InputStream, int)

java.sql.ResultSet.getBigDecimal(int, int)

java.sql.ResultSet.getBigDecimal(String, int)

java.sql.ResultSet.getUnicodeStream(int)

java.sql.ResultSet.getUnicodeStream(String)

java.sql.Time(int, int, int)

java.sql.Time.getDate()

java.sql.Time.getDay()

java.sql.Time.getMonth()

java.sql.Time.getYear()

java.sql.Time.setDate(int)

java.sql.Time.setMonth(int)

java.sql.Time.setYear(int)

java.sql.Timestamp(int, int, int, int, int, int, int)

Exceptions

The SQLException class and its subtypes provide information about errors and warnings that occur while a data source is being accessed.

7.1 SQLException

An instance of SQLException is thrown when an error occurs during an interaction with a data source. The exception contains the following information:

- a textual description of the error. The String containing the description can be retrieved by calling the method SQLException.getMessage.

- a SQLState. The String containing the SQLState can be retrieved by calling the method SQLException.getSQLState.

  The value of the SQLState string will depend on the underlying data source setting the value. Both X/Open and SQL:2003 define SQLState values and the conditions in which they should be set. Although the sets of values overlap, the values defined by SQL:2003 are not a superset of X/Open.

  The DatabaseMetaData method getSQLStateType allows an application to determine if the SQLStates being returned by a data source are X/Open or SQL:2003.

- an error code. This is an integer value identifying the error that caused the SQLException to be thrown. Its value and meaning are implementation specific and may be the actual error code returned by the underlying data source. The error code can be retrieved using the SQLException.getErrorCode method.

- a cause. This is another Throwable which caused this SQLException to occur.

■ a reference to any "chained" exceptions. If more than one error occurs the exceptions are referenced via this chain. All chained exception can be recursively retrieved by calling the SQLException.getNextException method on the exception that was thrown. If no more exceptions are chained, the getNextException method returns null.

There are multiple subclasses that extend SQLException. These subclasses are described in the following sections.

7.1.1 Support for the Java SE Chained Execeptions

The SQLException class and its subclasses have been enhanced to provide support for the Java SE chained exception facility, also known as the cause facility. The changes to support this functionality are:

■ The addition of four Constructors in order to provide support for the cause parameter.

■ Added support to the SQLException class to support the enchanced For-Each Loop, introduced in J2SE 5.0, allowing the navigation of SQLExceptions and their cause relationship without having to call getCause after each invocation of getNextException.

■ The getCause method may return Non-SQLExceptions in addition to SQLExceptions

Please refer to the JDBC 4.0 API specification for additional information.

7.1.2 Navigating SQLExceptions

It is possible that during the execution of a SQL statement that one or more Exceptions could occur, each with their own potential causal relationship. This means that when a JDBC application catches a SQLException, there is a possibility that there may be additional SQLExceptions chained to the original thrown SQLException. To access the additional chained SQLExceptions, an application would recursively invoke getNextException until a null value is returned.

A SQLException may have a causal relationship, which consists of one or more Throwables which caused the SQLException to be thrown. You may recursively call the method SQLException.getCause, until a null value is returned, to navigate the chain of causes.

The following code demonstrates how an application could navigate SQLExceptions and their causes.

```
catch(SQLException ex) {
    while(ex != null) {
        System.out.println("SQLState:" + ex.getSQLState());
        System.out.println("Error Code:" + ex.getErrorCode());
        System.out.println("Message:" + ex.getMessage());
        Throwable t = ex.getCause();
        while(t != null) {
            System.out.println("Cause:" + t);
            t = t.getCause();
        }
        ex = ex.getNextException();
    }
}
```

CODE EXAMPLE 7-1   Navigating SQLException and causes

7.1.2.1   Using a For-Each Loop with SQLExceptions

JDBC applications may use the Java SE For-Each Loop enhancement to navigate the
SQLExceptions and their causal relationship.

The following code demonstrates how to use the For-Each Loop with
SQLExceptions.

```
catch(SQLException ex) {
    for(Throwable e : ex ) {
        System.out.println("Error encountered: " + e);
    }
}
```

CODE EXAMPLE 7-2   Using SQLExceptions with a For-Each Loop

7.2   SQLWarning

`SQLWarning` is a subclass of `SQLException`. The methods in the following
interfaces will generate an `SQLWarning` object if they cause a database access
warning:

- `BaseQuery`
- `Connection`
- `DataSet`
- `Statement`
- `ResultSet`

When a method generates an `SQLWarning` object, the caller is not informed that a
data access warning has occurred. The method `getWarnings` must be called on the
appropriate object to retrieve the `SQLWarning` object. However, the
`DataTruncation` sub-class of `SQLWarning` may be thrown in some circumstances,
see "DataTruncation " in section 8.3 for more details.

If multiple data access warnings occur, they are chained to the first one and can be
retrieved by recursively calling the `SQLWarning.getNextWarning` method. If
there are no more warnings in the chain, `getNextWarning` returns `null`.

Subsequent `SQLWarning` objects continue to be added to the chain until the next
statement is executed or, in the case of a `ResultSet` object, when the cursor is re-
positioned, at which point all `SQLWarning` objects in the chain are removed.

7.3   DataTruncation

The `DataTruncation` class, a sub-class of `SQLWarning`, provides information
when data is truncated. When data truncation occurs on a write to the data source, a
`DataTruncation` object is thrown. The data value that has been truncated may
have been written to the data source even if a warning has been generated. When
data truncation occurs on a read from the data source, a `SQLWarning` is reported.

A `DataTruncation` object contains the following information:

- the descriptive `String` "Data truncation"
- the SQLState "01004" when data truncation occurs on a read from the data source
- the SQLState "22001" when data truncation occurs on a write to the data source

- a `boolean` to indicated whether a column value or a parameter was truncated. The method `DataTruncation.getParameter` returns `true` if a parameter was truncated and `false` if a column value was truncated.

- an `int` giving the index of the column or parameter that was truncated. If the index of the column or parameter is unknown, the method `DataTruncation.getIndex` returns `–1`. If the index is unknown, the values returned by the methods `DataTruncation.getParameter` and `DataTruncation.getRead` are undefined.

- a `boolean` to indicate whether the truncation occurred on a read or a write operation. The method `DataTruncation.getRead` returns `true` if the truncation occurred on a read and `false` if the truncation occurred on a write.

- The method `DataTruncation.getDataSize` returns an `int`, representing the number of bytes of data that should have been transferred. This number may be approximate if data conversions were being performed. The value may be -1 if the size is unknown.

- an `int` indicating the actual number of bytes that were transferred. The method `DataTruncation.getTransferSize` returns the number of bytes actually transferred or `–1` if the number of bytes is unknown.

7.3.1   Silent Truncation

The `Statement.setMaxFieldSize` method allows a maximum size (in bytes) to be set. This limit applies only to the `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR` , `LONGVARCHAR`, `NCHAR`, `NVARCHAR`, and `LONGNVARCHAR` data types.

If a limit has been set using `setMaxFieldSize` and there is an attempt to read or write data that exceeds the limit, any truncation that occurs as a result of exceeding the set limit will *not* be reported.

7.4   `BatchUpdateException`

A `BatchUpdateException` object provides information about errors that occur while a batch of statements is being executed. This exception's behavior is described in Chapter 14 "Batch Updates".

7.5 Categorized SQLExceptions

JDBC 4.0 introduces new `SQLException` subclasses which provide a standard mapping to common SQLStates class values and to common error states which are not associated with a specific SQLState class value. The SQLState class values are defined in the SQL:2003 specification. A JDBC driver may also throw a Categorized `SQLException` for JDBC driver detected errors. The new `SQLException` subclasses will provide a means for JDBC programmers to write more portable, error-handling code.

The new `SQLExceptions` will be broken into two exception categories: `SQLNonTransientException` and `SQLTransientException`.

Any SQLState class values which are currently not mapped to either a `SQLNonTransientException` or a `SQLTransientException` will result in a `SQLException` being thrown.

7.5.1 NonTransient SQLExceptions

A NonTransient `SQLException` must extend the class `SQLNonTransientException`. A NonTransient `SQLException` would be thrown in instances where a retry of the same operation would fail unless the cause of the `SQLException` is corrected.

Table 8-1 specifies which `NonTransientSQLException` subclass must be thrown for a a given SQLState class value:

TABLE 7-1   NonTransientSQLExeceptions Subclasses

SQL State Class	SQLNonTransientException Subclass
0A	SQLFeatureNotSupportedException
08	SQLNonTransientConnectionException
22	SQLDataException
23	SQLIntegrityConstraintViolationException
N/A	SQLInvalidAuthorizationException
42	SQLSyntaxErrorException

7.5.2 Transient SQLExceptions

A Transient `SQLException` must extend the class `SQLTransientException`. A Transient `SQLException` will be thrown in situations where a previously failed operation might be able to succeed when the operation is retried without any intervention by application-level functionality.

Table 8-2 specifies which `SQLTransientException` subclass must be thrown for a a given SQLState code:

TABLE 7-2   TransientSQLExeceptions Subclasses

SQL State Class	SQLTransientException Subclass
08	SQLTransientConnectionException
40	SQLTransactionRollbackException
N/A	SQLTimeoutException

7.6   SQLRuntimeException

A `SQLRuntimeException`, a sub-class of `RuntimeException`, is thrown by the Ease of Development methods, such as `DataSet.insert`. The `SQLRuntimeException.getCause` method can be invoked to determine the `Throwable` which caused the `SQLRuntimeException` to occur.

7.7   SQLDataSetSyncException

A `SQLDataSetSyncException`, a sub-class of `SQLRuntimeException`, is thrown by the method `DataSet.sync`. when failures occur propagating a `DataSet` to the data store. This exception's behavior is described in Chapter 19 "Ease of Development".

Database Metadata

The `DatabaseMetaData` interface is implemented by JDBC drivers to provide information about their underlying data sources. It is used primarily by application servers and tools to determine how to interact with a given data source. Applications may also use `DatabaseMetaData` methods to get information about a data source, but this is less typical.

The `DatabaseMetaData` interface includes over 150 methods, which can be categorized according to the types of information they provide:

- general information about the data source
- whether or not the data source supports a given feature or capability
- data source limits
- what SQL objects the data source contains and attributes of those objects
- transaction support offered by the data source

The `DatabaseMetaData` interface also contains over 40 fields, which are constants used as return values for various `DatabaseMetaData` methods.

This chapter presents an overview of the `DatabaseMetaData` interface, gives examples to illustrate the categories of metadata methods, and introduces some new methods. For a comprehensive listing, however, the reader should consult the JDBC 4.0 API specification.

JDBC also defines the `ResultSetMetaData` interface, which is discussed in Chapter 16 "Result Sets".

8.1 Creating a `DatabaseMetadata` Object

A `DatabaseMetaData` object is created with the `Connection` method `getMetaData`. Once created, it can be used to dynamically discover information about the underlying data source. CODE EXAMPLE 8-1 creates a `DatabaseMetadata` object and uses it to determine the maximum number of characters allowed for a table name.

```
// con is a Connection object
DatabaseMetaData dbmd = con.getMetadata();
int maxLen = dbmd.getMaxTableNameLength();
```

CODE EXAMPLE 8-1   Creating and using a `DatabaseMetadata` object

8.2 Retrieving General Information

Some `DatabaseMetaData` methods are used to dynamically discover general information about a data source as well as some details about its implementation. Some of the methods in this category are:

- `getURL`
- `getUserName`
- `getDatabaseProductVersion`, `getDriverMajorVersion` and `getDriverMinorVersion`
- `getSchemaTerm`, `getCatalogTerm` and `getProcedureTerm`
- `nullsAreSortedHigh` and `nullsAreSortedLow`
- `usesLocalFiles` and `usesLocalFilePerTable`
- `getSQLKeywords`

8.3 Determining Feature Support

A large group of `DatabaseMetaData` methods can be used to determine whether a given feature or set of features is supported by the driver or underlying data source. Beyond this, some of the methods describe what *level* of support is provided. Some of the methods that describe support for individual features are:

- `supportsAlterTableWithDropColumn`
- `supportsBatchUpdates`
- `supportsTableCorrelationNames`
- `supportsPositionedDelete`
- `supportsFullOuterJoins`
- `supportsStoredProcedures`
- `supportsMixedCaseQuotedIdentifiers`

Methods to describe a level of feature support include:

- `supportsANSI92EntryLevelSQL`
- `supportsCoreSQLGrammar`

8.4 Data Source Limits

Another group of methods provides the limits imposed by a given data source. Some of the methods in this category are:

- `getMaxRowSize`
- `getMaxStatementLength`
- `getMaxTablesInSelect`
- `getMaxConnections`
- `getMaxCharLiteralLength`
- `getMaxColumnsInTable`

Methods in this group return the limit as an `int`. A return value of zero means that there is no limit or the limit is unknown.

8.5 SQL Objects and Their Attributes

Some `DatabaseMetaData` methods provide information about the SQL objects that populate a given data source. This group also includes methods to determine the attributes of those objects. Methods in this group return `ResultSet` objects in which each row describes a particular object. For example, the method `getUDTs` returns a `ResultSet` object in which there is a row for each UDT that has been defined in the data source. Examples of this category are:

- `getSchemas`
- `getCatalogs`
- `getTables`
- `getPrimaryKeys`
- `getProcedures`
- `getProcedureColumns`
- `getUDTs`
- `getFunctions`
- `getFunctionParameters`

8.6 Transaction Support

A small group of methods provides information about the transaction semantics supported by the data source. Examples of this category include:

- `supportsMultipleTransactions`
- `getDefaultTransactionIsolation`

8.7 New Methods

The JDBC 4.0 API introduces the following new `DatabaseMetaData` methods:

- `getSchemas` — Overloaded the existing `getSchemas()` method to take a catalog and schema as arguments.

- `supportsStoredFunctionsUsingCallSyntax` — Method used to indicate whether user-defined or vendor functions may be called using the stored procedure escape syntax.

- `autoCommitFailureClosesAllResultSets` — Method used to indicate whether a `SQLException` while autoCommit is `true` indicates that all open ResultSets are closed, even ones that are holdable.

- `providesQueryObjectGenerator` — Method used to indicate whether a JDBC driver provides its own `QueryObjectGenerator` implementation.

- `getClientInfoProperties` — Method used to indicate the list of client info properties that the driver supports.

- `getFunctions` — Method used determine the list of user functions for a given catalog.

- `getFunctionParameters` — Method used determine the parameters and return types for user functions in a given catalog.

A complete definition of these methods may be found in the JDBC 4.0 API specification (javadoc).

8.8 Modified Methods

The JDBC 4.0 API modifies the definitions of these existing `DatabaseMetaData` methods:

- `getTypeInfo` — Clarified to indicate that if the database supports SQL distinct types, then getTypeInfo() will return a single row with a TYPE_NAME of DISTINCT and a DATA_TYPE of Types.DISTINCT. If the database supports SQL structured types, then getTypeInfo() will return a single row with a TYPE_NAME of STRUCT and a DATA_TYPE of Types.STRUCT. Support has also been added for the new data types SQLXML and ROWID.

- `getCrossReference` — Clarified that the method retrieves a description of the foreign key columns in the given foreign key table that reference the primary key or the unique constraint columns of the parent table (could be the same or a different table). The number of columns returned from the parent table must match the number of columns that make up the foreign key. They are ordered by FKTABLE_CAT, FKTABLE_SCHEM, FKTABLE_NAME, and KEY_SEQ.

- `getColumns` — Clarified that the return value for COLUMN_DEF that the default value for the column, should be interpreted as a string when the value is enclosed in quotes.

- `getProcedures` — Added the column SPECIFIC_NAME. The ordering of the returned rows is now by PROCEDURE_SCHEM, PROCEDURE_NAME and SPECIFIC_ NAME.

- `getProcedureColumns` — Added the columns COLUMN_DEF, SQL_DATATYPE, SQL_DATETIME_SUB, CHAR_OCTET_LENGTH, ORDINAL_POSITION, IS_NULLABLE and SPECIFIC_NAME. The ordering of the returned rows is now by PROCEDURE_SCHEM, PROCEDURE_NAME and SPECIFIC_ NAME.

The JDBC 4.0 API specification includes updated definitions of these methods.

Transactions

Transactions are used to provide data integrity, correct application semantics, and a consistent view of data during concurrent access. All JDBC compliant drivers are required to provide transaction support. Transaction management in the JDBC API mirrors the SQL:2003 specification and includes these concepts:

- Auto-commit mode
- Transaction isolation levels
- Savepoints

This chapter describes transaction semantics associated with a single `Connection` object. Transactions involving multiple `Connection` objects are discussed in Chapter 12 "Distributed Transactions".

9.1 Transaction Boundaries and Auto-commit

When to start a new transaction is a decision made implicitly by either the JDBC driver or the underlying data source. Although some data sources implement an explicit "begin transaction" statement, there is no JDBC API to do so. Typically, a new transaction is started when the current SQL statement requires one and there is no transaction already in place. Whether or not a given SQL statement requires a transaction is also specified by SQL:2003.

The `Connection` attribute *auto-commit* specifies when to end transactions. Enabling auto-commit causes a transaction commit after each individual SQL statement as soon as that statement is complete. The point at which a statement is considered to be "complete" depends on the type of SQL statement as well as what the application does after executing it:

- For Data Manipulation Language (DML) statements such as Insert, Update, Delete, and DDL statements, the statement is complete as soon as it has finished executing.
- For Select statements, the statement is complete when the associated result set is closed.
- For CallableStatement objects or for statements that return multiple results, the statement is complete when all of the associated result sets have been closed, and all update counts and output parameters have been retrieved.

9.1.1 Disabling Auto-commit Mode

CODE EXAMPLE 9-1 shows how to disable auto-commit mode.

```
// Assume con is a Connection object
con.setAutoCommit(false);
```

CODE EXAMPLE 9-1   Setting auto-commit off

When auto-commit is disabled, each transaction must be explicitly committed by calling the Connection method commit or explicitly rolled back by calling the Connection method rollback, respectively. This is appropriate for cases where transaction management is being done in a layer above the driver, such as:

- when the application needs to group multiple SQL statements into a single transaction
- when the transaction is being managed by the application server

The default is for auto-commit mode to be enabled when the Connection object is created. If the value of auto-commit is changed in the middle of a transaction, the current transaction is committed. If setAutoCommit is called and the value for auto-commit is not changed from its current value, it is treated as a no-op.

It is an error to enable auto-commit for a connection participating in a distributed transaction, as described in Chapter 12 "Distributed Transactions".

9.2 Transaction Isolation Levels

Transaction isolation levels specify what data is "visible" to the statements within a transaction. They directly impact the level of concurrent access by defining what interaction, if any, is possible between transactions against the same target data source. Possible interaction between concurrent transactions is categorized as follows:

- *dirty reads* occur when transactions are allowed to see uncommitted changes to the data. In other words, changes made inside a transaction are visible outside the transaction before it is committed. If the changes are rolled back instead of being committed, it is possible for other transactions to have done work based on incorrect, transient data.

- *nonrepeatable reads* occur when:

  a. Transaction A reads a row

  b. Transaction B changes the row

  c. Transaction A reads the same row a second time and gets different results

- *phantom reads* occur when:

  a. Transaction A reads all rows that satisfy a WHERE condition

  b. Transaction B inserts an additional row that satisfies the same condition

  c. Transaction A reevaluates the WHERE condition and picks up the additional "phantom" row

JDBC augments the four levels of transaction isolation defined by SQL:2003, by adding TRANSACTION_NONE. From least restrictive to most restrictive, the transaction isolation levels are:

1. TRANSACTION_NONE — indicates that the driver does not support transactions, which means that it is not a JDBC compliant driver.

2. TRANSACTION_READ_UNCOMMITTED — allows transactions to see uncommitted changes to the data. This means that dirty reads, nonrepeatable reads, and phantom reads are possible.

3. TRANSACTION_READ_COMMITTED — means that any changes made inside a transaction are not visible outside the transaction until the transaction is committed. This prevents dirty reads, but nonrepeatable reads and phantom reads are still possible.

4. TRANSACTION_REPEATABLE_READ — disallows dirty reads and nonrepeatable reads. Phantom read are still possible.

5. TRANSACTION_SERIALIZABLE — specifies that dirty reads, nonrepeatable reads, and phantom reads are prevented.

9.2.1 Using the setTransactionIsolation Method

The default transaction level for a Connection object is determined by the driver supplying the connection. Typically, it is the default transaction level supported by the underlying data source.

The `Connection` method `setTransactionIsolation` is provided to allow JDBC clients to change the transaction isolation level for a given `Connection` object. The new isolation level remains in effect for the remainder of the session or until the next invocation of the `setTransactionIsolation` method.

The result of invoking the method `setTransactionIsolation` in the middle of a transaction is implementation-defined.

The return value of the method `getTransactionIsolation` should reflect the change in isolation level when it actually occurs. It is recommended that drivers implement the `setTransactionIsolation` method to change the isolation level starting with the next transaction. Committing the current transaction to make the effect immediate is also a valid implementation.

It is possible for a given JDBC driver to not support all four transaction isolation levels (not counting `TRANSACTION_NONE`). If a driver does not support the isolation level specified in an invocation of `setTransactionIsolation`, it is allowed to substitute a higher, more restrictive transaction isolation level. If a driver is unable to substitute a higher transaction level, it throws an `SQLException`. The `DatabaseMetaData` method `supportsTransactionIsolationLevel` may be used to determine whether or not the driver supports a given level.

9.2.2 Performance Considerations

As the transaction isolation level increases, more locking and other DBMS overhead is required to ensure the correct semantics. This in turn lowers the degree of concurrent access that can be supported. As a result, applications may see decreased performance when they use a higher transaction isolation level. For this reason, the transaction manager, whether it is the application itself or part of the application server, should weigh the need for data consistency against the requirements for performance when determining which transaction isolation level is appropriate.

9.3 Savepoints

Savepoints provide finer-grained control of transactions by marking intermediate points within a transaction. Once a savepoint has been set, the transaction can be rolled back to that savepoint without affecting preceding work.

The `DatabaseMetaData.supportsSavepoints` method can be used to determine whether a JDBC driver and DBMS support savepoints.

9.3.1 Setting and Rolling Back to a Savepoint

The method `Connection.setSavepoint` can be used to set a savepoint within the current transaction. A transaction will be started if `setSavePoint` is invoked and there is not an active transaction. The `Connection.rollback` method has been overloaded to take a savepoint argument.

CODE EXAMPLE 9-2 inserts a row into a table, sets the savepoint *svpt1*, and then inserts a second row. When the transaction is later rolled back to *svpt1*, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing 'FIRST' will be added to TAB1.

```
conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) VALUES " +
                              "('FIRST')");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
rows = stmt.executeUpdate("INSERT INTO TAB1 (COL1) " +
                              "VALUES ('SECOND')");
...
conn.rollback(svpt1);
...
conn.commit();
```

CODE EXAMPLE 9-2   Rolling back a transaction to a savepoint

9.3.2 Releasing a Savepoint

The method `Connection.releaseSavepoint` takes a Savepoint object as a parameter and removes it and any subsequent savepoints from the current transaction.

Once a savepoint has been released, attempting to reference it in a rollback operation will cause an SQLException to be thrown.

Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed or when the entire transaction is rolled back.

Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints that were created after the savepoint in question.

Connections

A `Connection` object represents a connection to a data source via a JDBC technology-enabled driver. The data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. A single application using the JDBC API may maintain multiple connections. These connections may access multiple data sources, or they may all access a single data source.

From the JDBC driver perspective, a `Connection` object represents a client session. It has associated state information such as user ID, a set of SQL statements and result sets being used in that session, and what transaction semantics are in effect.

To obtain a connection, the application may interact with either:

- the `DriverManager` class working with one or more `Driver` implementations

OR

- a `DataSource` implementation

Using a `DataSource` object is the preferred method because it enhances application portability, it makes code maintenance easier, and it makes it possible for an application to transparently make use of connection pooling and distributed transactions. All Java EE components that establish a connection to a data source use a `DataSource` object to get a connection.

This chapter describes the various types of JDBC drivers and the use of the `Driver` interface, the `DriverManager` class, and the basic `DataSource` interface. `DataSource` implementations that support connection pooling and distributed transactions are discussed in Chapter 11 "Connection Pooling" and Chapter 12 "Distributed Transactions".

10.1   Types of Drivers

There are many possible implementations of JDBC drivers. These implementations are categorized as follows:

- Type 1 — drivers that implement the JDBC API as a mapping to another data access API, such as ODBC. Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge driver is an example of a Type 1 driver.

- Type 2 — drivers that are written partly in the Java programming language and partly in native code. These drivers use a native client library specific to the data source to which they connect. Again, because of the native code, their portability is limited.

- Type 3 — drivers that use a pure Java client and communicate with a middleware server using a database-independent protocol. The middleware server then communicates the client's requests to the data source.

- Type 4 — drivers that are pure Java and implement the network protocol for a specific data source. The client connects directly to the data source.

10.2   The `Driver` Interface

JDBC drivers must implement the `Driver` interface, and the implementation must contain a static initializer that will be called when the driver is loaded. This initializer registers a new instance of itself with the `DriverManager`, as shown in CODE EXAMPLE 10-1.

```
public class AcmeJdbcDriver implements java.sql.Driver {

                static {

                java.sql.DriverManager.registerDriver(new
AcmeJdbcDriver());

                }

...

}
```

CODE EXAMPLE 10-1   Example static initializer for a driver implementing
                `java.sql.Driver`

When an application loads a `Driver` implementation, the static initializer will automatically register an instance of the driver.

To insure that drivers can be loaded using this mechanism, drivers are required to provide a niladic constructor.

The `DriverManager` class invokes `Driver` methods when it wishes to interact with a registered driver. The `Driver` interface also includes the method `acceptsURL`. The `DriverManager` can use this method to determine which of its registered drivers it should use for a given URL.

When the `DriverManager` is trying to establish a connection, it calls that driver's `connect` method and passes the driver the URL. If the `Driver` implementation understands the URL, it will return a `Connection` object; otherwise it returns `null`.

10.2.1　Loading a driver that implements java.sql.Driver

As part of its initialization, the `DriverManager` class will attempt to load any JDBC drivers classes referenced in the "*jdbc.drivers*" system property.

```
java -Djdbc.drivers=com.acme.jdbc.AcmeJdbcDriver Test
```

CODE EXAMPLE 10-2　Loading a driver using the `jdbc.drivers` system property

The `DriverManager.getConnection` method has been enhanced to support the Java Standard Edition Service Provider mechanism. JDBC 4.0 Drivers must include the file `META-INF/services/java.sql.Driver`. This file contains the name of the JDBC driver's implementation of `java.sql.Driver`. CODE EXAMPLE 10-3 shows the contents of the META-INF/services/java.sql.Driver file in order to load the my.sql.driver class.

```
my.sql.Driver
```

CODE EXAMPLE 10-3　`META-INF/services/java.sql.Driver` file contents

Note – Existing applications that currently load JDBC drivers using `Class.forName()` will continue to work without modification.

10.3　The `DriverManager` Class

The `DriverManager` class works with the `Driver` interface to manage the set of drivers available to a JDBC client. When the client requests a connection and provides a URL, the `DriverManager` is responsible for finding a driver that recognizes the URL and using it to connect to the corresponding data source.

Key `DriverManager` methods include:

- `registerDriver` — this method adds a driver to the set of available drivers and is invoked implicitly when the driver is loaded. The `registerDriver` method is typically called by the static initializer provided by each driver.

- `getConnection` — the method the JDBC client invokes to establish a connection. The invocation includes a JDBC URL, which the `DriverManager` passes to each driver in its list until it finds one whose `Driver.connect` method recognizes the URL. That driver returns a `Connection` object to the `DriverManager`, which in turn passes it to the application.

The format of a JDBC URL is :

- jdbc:<subprotocol>:<subname>

where *subprotocol* defines the kind of database connectivity mechanism that may be supported by one or more drivers. The contents and syntax of the *subname* will depend on the *subprotocol*.

Note – A JDBC URL is not required to fully adhere to the URI syntax as defined in RFC 3986, Uniform Resource Identifier (URI): Generic Syntax.

CODE EXAMPLE 10-4 illustrates how a JDBC client obtains a connection from the `DriverManager`.

```
// Set up arguments for the call to the getConnection method.
// The sub-protocol "odbc" in the driver URL indicates the
// use of the JDBC-ODBC bridge.
String url = "jdbc:odbc:DSN";
String user = "SomeUser";
String passwd = "SomePwd";


// Get a connection from the first driver in the DriverManager
// list that recognizes the URL "jdbc:odbc:DSN".

// The call to geConnection will also load the driver if needed.
// When the driver is loaded, an instance of the driver is created
// and the registerDriver method is also called to make acme.db.Driver
// available to clients.


Connection con = DriverManager.getConnection(url, user, passwd);
```

CODE EXAMPLE 10-4   Loading a driver and getting a connection using the `DriverManager`

The `DriverManager` class also provides two other `getConnection` methods:

- `getConnection(String url)` for connecting to data sources that do not use username and passwords.
- `getConnection(String url, java.util.Properties prop)`, which allows the client to connect using a set of properties describing the user name and password along with any addition information that may be required.

The `DriverPropertyInfo` class provides information on the properties that the JDBC driver can understand.

See the JDBC 4.0 API Specification for more details.

10.3.1   The `SQLPermission` Class

The `SQLPermission` class represents a set of permissions that a codebase may be granted.

Currently the only permission defined is `setLog`. The `SecurityManager` will check for the `setLog` permission when an Applet calls one of the `DriverManager` methods `setLogWriter` and `setLogStream`. If the codebase does not have the `setLog` permission a `java.lang.SecurityException` exception will be thrown.

See the JDBC 4.0 API Specification for more details.

10.4   The `DataSource` Interface

The `DataSource` interface, introduced in JDBC 2.0 Optional Package, is the preferred approach to obtaining data source connections. A JDBC driver that implements the `Datasource` interface returns connections that implement the same interface, `Connection`, as those returned by a `DriverManager` using the `Driver` interface. Using a `Datasource` object increases application portability by making it possible for an application to use a logical name for a data source instead of having to supply information specific to a particular driver. A logical name is mapped to a `DataSource` object via a naming service that uses the Java Naming and Directory Interface™ (JNDI). The `DataSource` object, represents a physical data source and provides connections to that data source. If the data source or information about it changes, the properties of the `DataSource` object can simply be modified to reflect the changes; no change in application code is necessary.

The DataSource interface can be implemented so that it transparently provides the following:

- Increased performance and scalability through connection pooling
- Support for distributed transactions through the XADataSource interface

The next three sections discuss (1) basic DataSource properties, (2) how logical naming using the JNDI API improves an applications portability and makes it easier to maintain, and (3) how to obtain a connection.

Connection pooling and distributed transactions will be discussed in Chapter 11 "Connection Pooling" and Chapter 12 "Distributed Transactions".

10.4.1   DataSource Properties

The JDBC API defines a set of properties to identify and describe a DataSource implementation. The actual set required for a specific implementation depends on the type of DataSource object, that is, whether it is a basic DataSource object, a ConnectionPoolDataSource object, or an XADataSource object. The only property required for all DataSource implementations is description.

The following table describes the standard DataSource properties:

TABLE 10-1   Standard Data Source Properties

Property Name	Type	Description
databaseName	String	name of a particular database on a server
dataSourceName	String	a data source name; used to name an underlying XADataSource object or ConnectionPoolDataSource object when pooling of connections is done
description	String	description of this data source
networkProtocol	String	network protocol used to communicate with the server
password	String	a database password
portNumber	int	port number where a server is listening for requests
roleName	String	the initial SQL rolename
serverName	String	database server name
user	String	user's account name

DataSource properties follow the convention specified for properties of JavaBeans™ components in the JavaBeans 1.01 Specification. DataSource implementations may augment this set with implementation-specific properties. If new properties are added, they must be given names that do not conflict with the standard property names.

DataSource implementations must provide "getter" and "setter" methods for each property they support. These properties typically are initialized when the DataSource object is deployed, as in CODE EXAMPLE 10-5, in which a VendorDataSource object implements the DataSource interface.

```
VendorDataSource vds = new VendorDataSource();

vds.setServerName("my_database_server");

String name = vds.getServerName();
```

CODE EXAMPLE 10-5   Setting and getting a DataSource property

DataSource properties are not intended to be directly accessible by JDBC clients. This design is reinforced by defining the access methods on the implementation class rather than on the public DataSource interface used by applications. Furthermore, the object that the client manipulates can be a wrapper that only implements the DataSource interface. The setter and getter methods for the properties need not be exposed to the client.

Management tools that need to manipulate the properties of a DataSource implementation can access those properties using introspection.

10.4.2   The JNDI API and Application Portability

The Java Naming and Directory Interface (JNDI) API provides a uniform way for applications to access remote services over the network. This section describes how it is used to register and access a JDBC DataSource object. See the JNDI specification for a complete description of this interface.

Using the JNDI API, applications can access a DataSource object by specifying its logical name. A naming service using the JNDI API maps this logical name to a corresponding data source. This scheme greatly enhances portability because any of the DataSource properties, such as portNumber or serverName, can be changed without impacting the JDBC client code. In fact, the application can be re-directed to a different underlying data source in a completely transparent fashion. This is particularly useful in the three-tier environment, where an application server hides the details of accessing different data sources.

CODE EXAMPLE 10-6 illustrates the use of a JNDI-based naming service to deploy a new VendorDataSource object.

```
// Create a VendorDataSource object and set some properties
VendorDataSource vds = new VendorDataSource();
vds.setServerName("my_database_server");
vds.setDatabaseName("my_database");
vds.setDescription("data source for inventory and personnel");

// Use the JNDI API to register the new VendorDataSource object.
// Reference the root JNDI naming context and then bind the
// logical name "jdbc/AcmeDB" to the new VendorDataSource object.
Context ctx = new InitialContext();
ctx.bind("jdbc/AcmeDB", vds);
```

CODE EXAMPLE 10-6   Registering a `DataSource` object with a JNDI-based naming service

---

Note – Java EE components use a special convention for naming their data sources — see Chapter 5 "Naming" in the Java EE platform specification for more details.

10.4.3   Getting a Connection with a `DataSource` Object

Once a `DataSource` object has been registered with a JNDI-based naming service, an application can use it to obtain a connection to the physical data source that it represents, as is done in CODE EXAMPLE 10-7.

```
// Get the initial JNDI naming context
Context ctx = new InitialContext();

// Get the DataSource object associated with the logical name
// "jdbc/AcmeDB" and use it to obtain a database connection
DataSource ds = (DataSource)ctx.lookup("jdbc/AcmeDB");
Connection con = ds.getConnection("user", "pwd");
```

CODE EXAMPLE 10-7   Getting a `Connection` object using a `DataSource` object

The `DataSource` implementation bound to the name "jdbc/AcmeDB" can be modified or replaced without affecting the application code.

Connection Pooling

In a basic `DataSource` implementation, there is a 1:1 correspondence between the client's `Connection` object and the physical database connection. When the `Connection` object is closed, the physical connection is dropped. Thus, the overhead of opening, initializing, and closing the physical connection is incurred for each client session.

A *connection pool* solves this problem by maintaining a cache of physical database connections that can be reused across client sessions. Connection pooling greatly improves performance and scalability, particularly in a three-tier environment where multiple clients can share a smaller number of physical database connections. In FIGURE 11-1, the JDBC driver provides an implementation of `ConnectionPoolDataSource` that the application server uses to build and manage the connection pool.

The algorithm used to manage the connection pool is implementation-specific and varies with application servers. The application server provides its clients with an implementation of the `DataSource` interface that makes connection pooling transparent to the client. As a result, the client gets better performance and scalability while using the same JNDI and `DataSource` APIs as before.
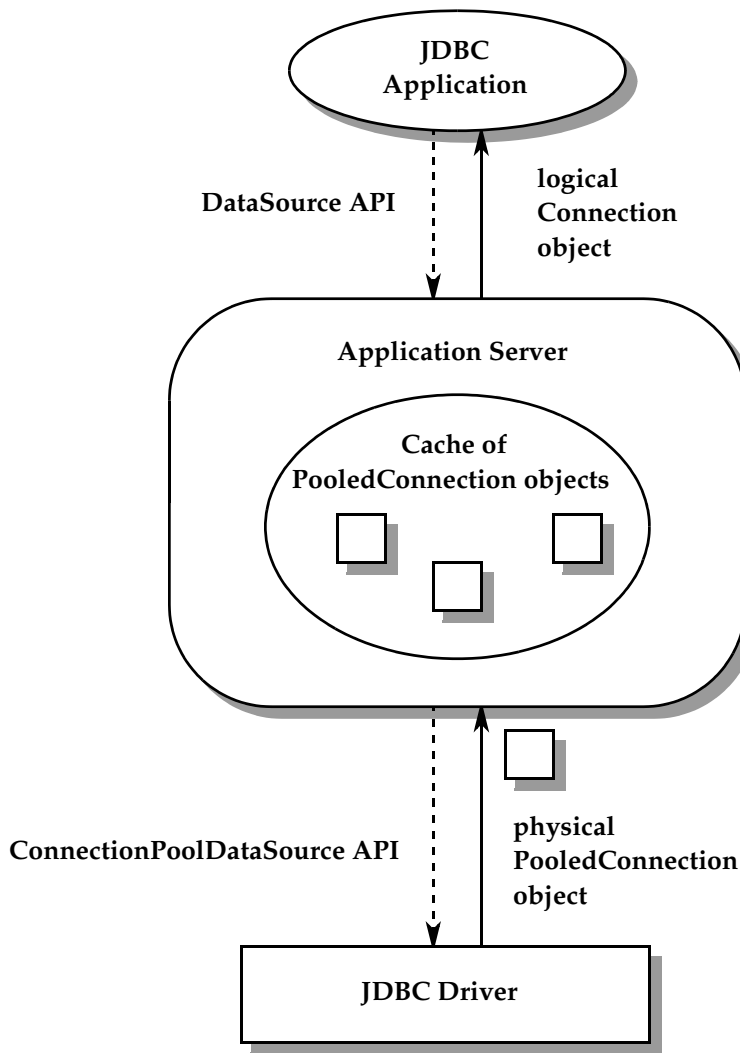
FIGURE 11-1 Connection pooling

The following sections introduce the ConnectionPoolDataSource interface, the PooledConnection interface, and the ConnectionEvent class. These pieces, which operate beneath the DataSource and Connection interfaces used by the client, are incorporated into a step-by-step description of a typical connection pooling implementation. This chapter also describes some important differences

between a basic `DataSource` object and one that implements connection pooling. In addition, it discusses how a pooled connection can maintain a pool of reusable `PreparedStatement` objects.

Although much of the discussion in this chapter assumes a three-tier environment, connection pooling is also relevant in a two-tier environment. In a two-tier environment, the JDBC driver implements both the `DataSource` and `ConnectionPoolDataSource` interfaces. This implementation allows an application that opens and closes multiple connections to benefit from connection pooling.

11.1 `ConnectionPoolDataSource` and `PooledConnection`

Typically, a JDBC driver implements the `ConnectionPoolDataSource` interface, and the application server uses it to obtain `PooledConnection` objects. CODE EXAMPLE 11-1 shows the signatures for the two versions of the `getPooledConnection` method.

```
public interface ConnectionPoolDataSource {

                PooledConnection getPooledConnection() throws
SQLException;

                PooledConnection getPooledConnection(String user,

                    String password) throws SQLException;

        ...

}
```

CODE EXAMPLE 11-1   The `ConnectionPoolDataSource` interface

A `PooledConnection` object represents a physical connection to a data source. The JDBC driver's implementation of `PooledConnection` encapsulates all of the details of maintaining that connection.

An application server caches and reuses `PooledConnection` objects within its implementation of the `DataSource` interface. When a client calls the method `DataSource.getConnection`, the application server uses the physical `PooledConnection` object to obtain a logical `Connection` object. CODE EXAMPLE 11-2 shows the `PooledConnection` interface definition.

```
public interface PooledConnection {

                Connection getConnection() throws SQLException;
                void close() throws SQLException;
                void addConnectionEventListener(
                    ConnectionEventListener listener);
                void addStatementEventListener(
                    StatementEventListener listener);
                void removeConnectionEventListener(
                    ConnectionEventListener listener);
                void removeStatementEventListener(
                    StatementEventListener listener);


}
```

CODE EXAMPLE 11-2   The `PooledConnection` interface

When an application is finished using a connection, it closes the logical connection using the method `Connection.close`. This closes the logical connection but does not close the physical connection. Instead, the physical connection is returned to the pool so that it can be reused.

Connection pooling is completely transparent to the client: A client obtains a pooled connection and uses it just the same way it obtains and uses a non pooled connection.

11.2   Connection Events

Recall that when an application calls the method `Connection.close`, the underlying physical connection—the `PooledConnection` object—becomes available for reuse. JavaBeans-style events are used to notify the connection pool manager (the application server) that a `PooledConnection` object can be recycled.

In order to be notified of an event on a `PooledConnection` object, the connection pool manager must implement the `ConnectionEventListener` interface and then be registered as a listener by that `PooledConnection` object. The `ConnectionEventListener` interface defines the following two methods, which correspond to the two kinds of events that can occur on a `PooledConnection` object:

■ `connectionClosed` — triggered when the logical `Connection` object associated with this `PooledConnection` object is closed, that is, the application called the method `Connection.close`

- `connectionErrorOccurred` — triggered when a fatal error, such as the server crashing, causes the connection to be lost

A connection pool manager registers itself as a listener for a `PooledConnection` object using the `PooledConnection.addConnectionEventListener` method. Typically, a connection pool manager registers itself as a `ConnectionEventListener` before returning a `Connection` object to an application.

The driver invokes the `ConnectionEventListener` methods `connectionClosed` and `connectionErrorOccurred` when the corresponding events occur. Both methods take a `ConnectionEvent` object as a parameter, which can be used to determine which `PooledConnection` object was closed or had an error. When the JDBC application closes its logical connection, the JDBC driver notifies the connection pool manager (the listener) by calling the listener's implementation of the method `connectionClosed`. At this point, the connection pool manager can return the `PooledConnection` object to the pool for reuse.

When an error occurs, the JDBC driver notifies the listener by calling its `connectionErrorOccurred` method and then throws an `SQLException` object to the application to notify it of the same error. In the event of a fatal error, the bad `PooledConnection` object is not returned to the pool. Instead, the connection pool manager calls the `PooledConnection.close` method on the `PooledConnection` object to close the physical connection.

11.3 Connection Pooling in a Three-tier Environment

The following sequence of steps outlines what happens when a JDBC client requests a connection from a `DataSource` object that implements connection pooling:

- The client calls `DataSource.getConnection`.
- The application server providing the `DataSource` implementation looks in its connection pool to see if there is a suitable `PooledConnection` object— a physical database connection—available. Determining the suitability of a given `PooledConnection` object may include matching the client's user authentication information or application type as well as using other implementation-specific criteria. The lookup method and other methods associated with managing the connection pool are specific to the application server.

- If there are no suitable `PooledConnection` objects available, the application server calls the `ConnectionPoolDataSource.getPooledConnection` method to get a new physical connection. The JDBC driver implementing `ConnectionPoolDataSource` creates a new `PooledConnection` object and returns it to the application server.

- Regardless of whether the `PooledConnection` was retrieved from the pool or was newly created, the application server does some internal bookkeeping to indicate that the physical connection is now in use.

- The application server calls the method `PooledConnection.getConnection` to get a logical `Connection` object. This logical `Connection` object is actually a "handle" to a physical `PooledConnection` object, and it is this handle that is returned by the `DataSource.getConnection` method when connection pooling is in effect.

- The application server registers itself as a `ConnectionEventListener` by calling the method `PooledConnection.addConnectionEventListener`. This is done so that the application server will be notified when the `PooledConnection` object is available for reuse.

- The logical `Connection` object is returned to the JDBC client, which uses the same `Connection` API as in the basic `DataSource` case. Note that the underlying `PooledConnection` object cannot be reused until the client calls the method `Connection.close`.

Connection pooling can also be implemented in a two-tier environment where there is no application server. In this case, the JDBC driver provides both the implementation of `DataSource` which is visible to the client and the underlying `ConnectionPoolDataSource` implementation.

11.4 `DataSource` Implementations and Connection Pooling

Aside from improved performance and scalability, a JDBC application should not see any difference between accessing a `DataSource` object that implements connection pooling and one that does not. However, there are some important differences in the application server and driver level implementations.

A basic `DataSource` implementation, that is, one that does not implement connection pooling, is typically provided by a JDBC driver vendor. In a basic `DataSource` implementation, the following are true:

- The `DataSource.getConnection` method creates a new `Connection` object that represents a physical connection and encapsulates all of the work to set up and manage that connection.

- The `Connection.close` method shuts down the physical connection and frees the associated resources.

In a `DataSource` implementation that includes connection pooling, a great deal happens behind the scenes. In such an implementation, the following are true:

- The `DataSource` implementation includes an implementation-specific connection pooling module that manages a cache of `PooledConnection` objects. The `DataSource` object is typically implemented by the application server as a layer on top of the driver's implementations of the `ConnectionPoolDataSource` and `PooledConnection` interfaces.

- The `DataSource.getConnection` method calls `PooledConnection.getConnection` to get a logical handle to an underlying physical connection. The overhead of setting up a new physical connection is incurred only if there are no existing connections available in the connection pool. When a new physical connection is needed, the connection pool manager will call the `ConnectionPoolDataSource` method `getPooledConnection` to create one. The work to manage the physical connection is delegated to the `PooledConnection` object.

- The `Connection.close` method closes the logical handle, but the physical connection is maintained. The connection pool manager is notified that the underlying `PooledConnection` object is now available for reuse. If the application attempts to reuse the logical handle, the `Connection` implementation throws an `SQLException`.

- A single physical `PooledConnection` object may generate many logical `Connection` objects during its lifetime. For a given `PooledConnection` object, only the most recently produced logical `Connection` object will be valid. Any previously existing `Connection` object is automatically closed when the associated `PooledConnection.getConnection` method is called. Listeners (connection pool managers) are not notified in this case.

  This gives the application server a way to take a connection away from a client. This is an unlikely scenario but may be useful if the application server is trying to force an orderly shutdown.

- A connection pool manager shuts down a physical connection by calling the method `PooledConnection.close`. This method is typically called only in certain circumstances: when the application server is undergoing an orderly shutdown, when the connection cache is being re initialized, or when the application server receives an event indicating that an unrecoverable error has occurred on the connection.

11.5 Deployment

Deploying a `DataSource` object that implements connection pooling requires that both a client-visible `DataSource` object and an underlying `ConnectionPoolDataSource` object be registered with a JNDI-based naming service.

The first step is to deploy the `ConnectionPoolDataSource` implementation, as is done in CODE EXAMPLE 11-3.

```
// ConnectionPoolDS implements the ConnectionPoolDataSource

// interface. Create an instance and set properties.

com.acme.jdbc.ConnectionPoolDS cpds =

                new com.acme.jdbc.ConnectionPoolDS();

cpds.setServerName("bookserver");

cpds.setDatabaseName("booklist");

cpds.setPortNumber(9040);

cpds.setDescription("Connection pooling for bookserver");


// Register the ConnectionPoolDS with JNDI, using the logical name

// "jdbc/pool/bookserver_pool"

Context ctx = new InitialContext();

ctx.bind("jdbc/pool/bookserver_pool", cpds);
```

CODE EXAMPLE 11-3   Deploying a `ConnectionPoolDataSource` object

Once this step is complete, the `ConnectionPoolDataSource` implementation is available as a foundation for the client-visible `DataSource` implementation. The `DataSource` implementation is deployed such that it references the `ConnectionPoolDataSource` implementation, as shown in CODE EXAMPLE 11-4.

```
// PooledDataSource implements the DataSource interface.

// Create an instance and set properties.

com.acme.appserver.PooledDataSource ds =

                new com.acme.appserver.PooledDataSource();

ds.setDescription("Datasource with connection pooling");


// Reference the previously registered ConnectionPoolDataSource

ds.setDataSourceName("jdbc/pool/bookserver_pool");
```

```
// Register the DataSource implementation with JNDI, using the logical
// name "jdbc/bookserver".

Context ctx = new InitialContext();

ctx.bind("jdbc/bookserver", ds);
```

CODE EXAMPLE 11-4   Deploying a `DataSource` object backed by a
                       `ConnectionPoolDataSource` object

The `DataSource` object is now available for use in an application.

11.6   Reuse of Statements by Pooled Connections

The JDBC specification provides support for statement pooling. This feature, which
allows an application to reuse a `PreparedStatement` object in much the same way
it can reuse a connection, is made available through a pooled connection.

FIGURE 11-2 provides a logical view of how a pool of `PreparedStatement` objects
can be associated with a `PooledConnection` object. As with the
`PooledConnection` object itself, the `PreparedStatement` objects can be reused
by multiple logical connections in a transparent manner.
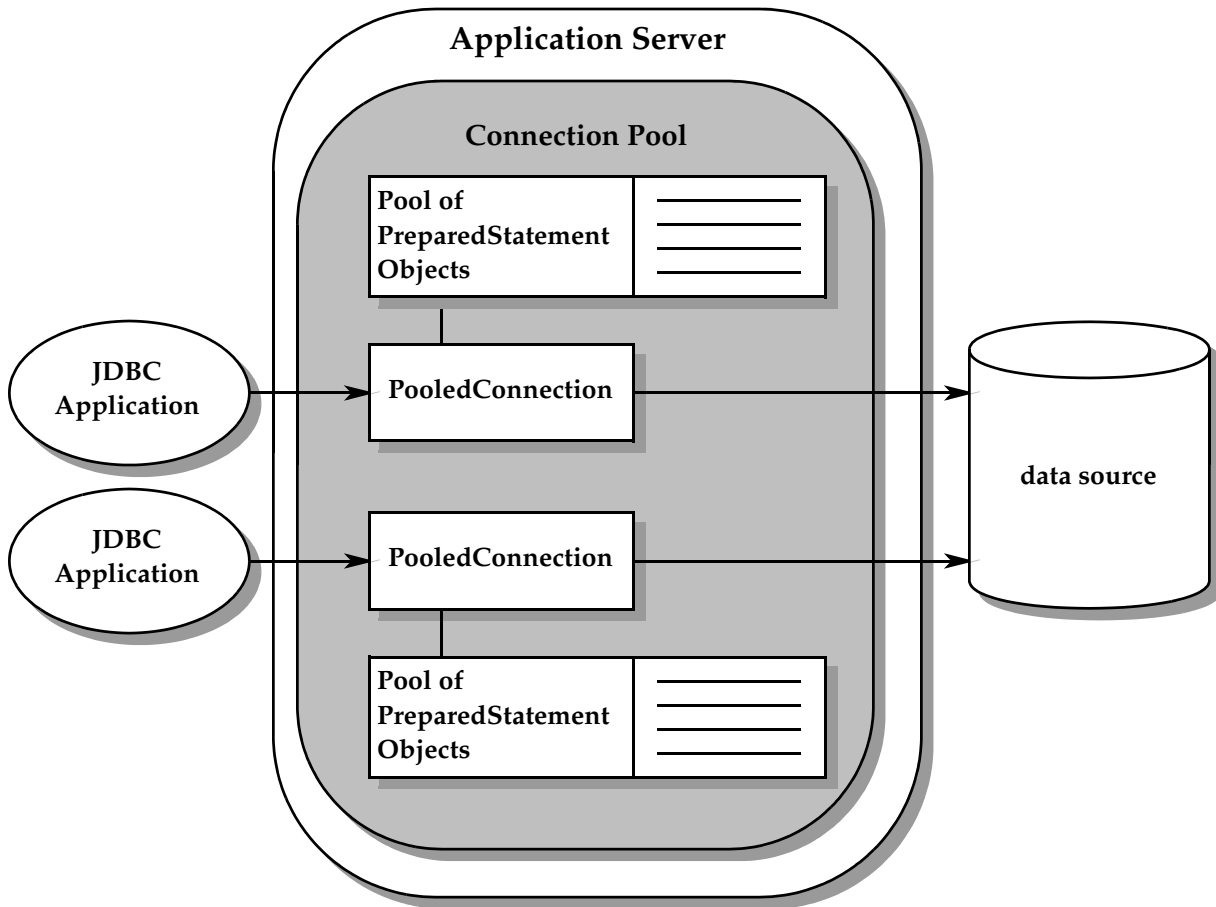
FIGURE 11-2 Logical view of prepared statements reused by pooled connections

In FIGURE 11-2, the connection pool and statement pool are implemented by the application server. However, this functionality could also be implemented by the driver or underlying data source. This discussion of statement pooling is meant to allow for any of these implementations.

11.6.1 Using a Pooled Statement

If a pooled connection reuses statements, the reuse must be completely transparent to an application. In other words, from the application's point of view, using a PreparedStatement object that participates in statement pooling is exactly the

same as using one that does not. Statements are kept open for reuse entirely under the covers, so there is no change in application code. If an application closes a `PreparedStatement` object, it must still call `Connection.prepareStatement` in order to use it again. The only visible effect of statement pooling is a possible improvement in performance.

An application may find out whether a data source supports statement pooling by calling the `DatabaseMetaData` method `supportsStatementPooling`. If the return value is `true`, the application can then choose to use `PreparedStatement` objects knowing that they are being pooled.

In many cases, reusing statements is a significant optimization. This is especially true for complex prepared statements. However, it should also be noted that leaving large numbers of statements open may have an adverse impact on the use of resources.

11.6.2   Closing a Pooled Statement

An application closes a pooled statement exactly the same way it closes a non pooled statement. Whether it is pooled or not, a statement that has been closed is no longer available for use by the application, and an attempt to reuse it will cause an exception to be thrown.

The following methods can close a pooled statement:

- `Statement.close` — called by an application; if the statement is being pooled, closes the logical statement used by the application but does not close the physical statement being pooled
- `Connection.close` — called by an application
  - Non pooled connection — closes the physical connection and all statements created by that connection. This is necessary because the garbage collection mechanism is unable to detect when externally managed resources can be released.
  - Pooled connection — closes the logical connection and the logical statements it returned but leaves open the underlying `PooledConnection` object and any associated pooled statements
- `PooledConnection.close` — called by the connection pool manager to close the physical connection and the associated physical statements being pooled by the `PooledConnection` object

An application cannot directly close a physical statement that is being pooled; instead, this is done by the connection pool manager. The method `PooledConnection.close` closes the connection and all of the statements open on a given connection, which releases the resources associated with those statements.

An application also has no direct control over how statements are pooled. A pool of statements is associated with a `PooledConnection` object, whose behavior is determined by the properties of the `ConnectionPoolDataSource` object that produced it. Section 11.8 "`ConnectionPoolDataSource` Properties" discusses these properties.

11.7 Statement Events

If the connection pool manager supports Statement pooling for `PreparedStatement` objects, it must implement the `StatementEventListener` interface and then be registered as a listener by that `PooledConnection` object. The `StatementEventListener` interface defines the following two methods, which correspond to the two kinds of events that can occur on a `PreparedStatement` object:

- `statementClosed` — triggered when the logical `PreparedStatement` object associated with this `PooledConnection` object is closed, that is, the application called the method `PreparedStatement.close`

- `statementErrorOccurred` — triggered when a JDBC driver determines that a `PreparedStatement` object is no longer valid

A connection pool manager registers itself as a listener for a `PreparedStatement` object using the `PooledConnection.addStatementEventListener` method. Typically, a connection pool manager registers itself as a `StatementEventListener` before returning a `PreparedStatement` object to an application.

The driver invokes the `StatementEventListener` methods `statementClosed` and `statementErrorOccurred` when the corresponding events occur. Both methods take a `statementEvent` object as a parameter, which can be used to determine which `PreparedStatement` object was closed or had an error. When the JDBC application closes its logical prepared statement, the JDBC driver notifies the connection pool manager (the listener) by calling the listener's implementation of the method `statementClosed`. At this point, the connection pool manager can return the `PreparedStatement` object to the pool for reuse.

When an error occurs that makes a `PreparedStatement` object invalid, the JDBC driver notifies the listener by calling its `statementErrorOccurred` method and then throws an `SQLException` object to the application to notify it of the same error.

11.8 ConnectionPoolDataSource Properties

As with the `DataSource` interface, the JDBC API defines a set of properties that can be used to configure the behavior of connection pools. These are shown in TABLE 11-1:

TABLE 11-1   Standard Connection Pool Properties

Property Name	Type	Description
maxStatements	int	The total number of statements that the pool should keep open. 0 (zero) indicates that caching of statements is disabled.
initialPoolSize	int	The number of physical connections the pool should contain when it is created
minPoolSize	int	The number of physical connections the pool should keep available at all times. 0 (zero) indicates that connections should be created as needed.
maxPoolSize	int	The maximum number of physical connections that the pool should contain. 0 (zero) indicates no maximum size.
maxIdleTime	int	The number of seconds that a physical connection should remain unused in the pool before the connection is closed. 0 (zero) indicates no limit.
propertyCycle	int	The interval, in seconds, that the pool should wait before enforcing the current policy defined by the values of the above connection pool properties

Connection pool configuration properties follow the convention specified for JavaBeans components in the JavaBeans specification. Connection pool vendors may choose to augment this set with implementation-specific properties. If so, the additional properties must be given names that do not conflict with the standard property names.

Like `DataSource` implementations, `ConnectionPoolDataSource` implementations must provide "getter" and "setter" methods for each property they support. These properties are typically initialized when the `ConnectionPoolDataSource` object is deployed. CODE EXAMPLE 11-5 illustrates setting properties in a vendor's implementation of the `ConnectionPoolDataSource` interface.

```
VendorConnectionPoolDS vcp = new VendorConnectionPoolDS();

vcp.setMaxStatements(25);

vcp.setInitialPoolSize(10);

vcp.setMinPoolSize(1);

vcp.setMaxPoolSize(0);

vcp.setMaxIdleTime(0);

vcp.setPropertyCycle(300);
```

CODE EXAMPLE 11-5   Setting connection pool configuration properties

The properties set on a ConnectionPoolDataSource object apply to the
PooledConnection objects that it creates. An application server managing a pool
of PooledConnection objects uses these properties to determine how to manage its
pool.

ConnectionPoolDataSource configuration properties are not intended to be
directly accessible by JDBC clients. Management tools that need to manipulate the
properties of a ConnectionPoolDataSource implementation can access those
properties using introspection.

Distributed Transactions

Up to this point, the discussion of transactions has focused on the local case—transactions involving a single data source. This chapter introduces the distributed case where a single transaction involves multiple connections to one or more underlying data sources.

The following discussion includes these topics:

- distributed transaction infrastructure
- transaction managers and resource managers
- the `XADataSource`, `XAConnection`, and `XAResource` interfaces
- two-phase commit

Transaction management in the JDBC API is designed to fit with the Java Transaction API^{TM} (JTA^{TM}) . The examples presented here are high-level; the JTA specification should be consulted for a more substantial discussion.

12.1 Infrastructure

Distributed transactions require an infrastructure that provides these roles:

- Transaction manager — controls transaction boundaries and manages the two-phase commit protocol. This typically will be an implementation of JTA.
- JDBC drivers that implement the `XADataSource`, `XAConnection`, and `XAResource` interfaces. These are described in the next section.
- An application-visible implementation of `DataSource` to "sit on top of" each `XADataSource` object and interact with the transaction manager. The `DataSource` implementation is typically provided by an application server. Because the `XADataSource` interface extends the `ConnectionPoolDataSource` interface, the `DataSource` implementation will usually include a connection pooling module as well.

■ Resource manager(s) to manage the underlying data. In the context of the JDBC API, a resource manager is a DBMS server. The term "resource manager" is borrowed from JTA to emphasize the point that distributed transactions using the JDBC API follow the architecture specified in that document.

This infrastructure is most often implemented in a three-tier architecture that includes the following:

1. A client tier

2. A middle tier that includes applications, an EJB server working with an external transaction manager, and a set of JDBC drivers

3. Multiple resource managers

Distributed transactions can also be implemented in two tiers. In a two-tier architecture, the application itself acts as the transaction manager and interacts directly with the JDBC drivers' XADataSource implementations.

The following diagram illustrates the distributed transaction infrastructure:
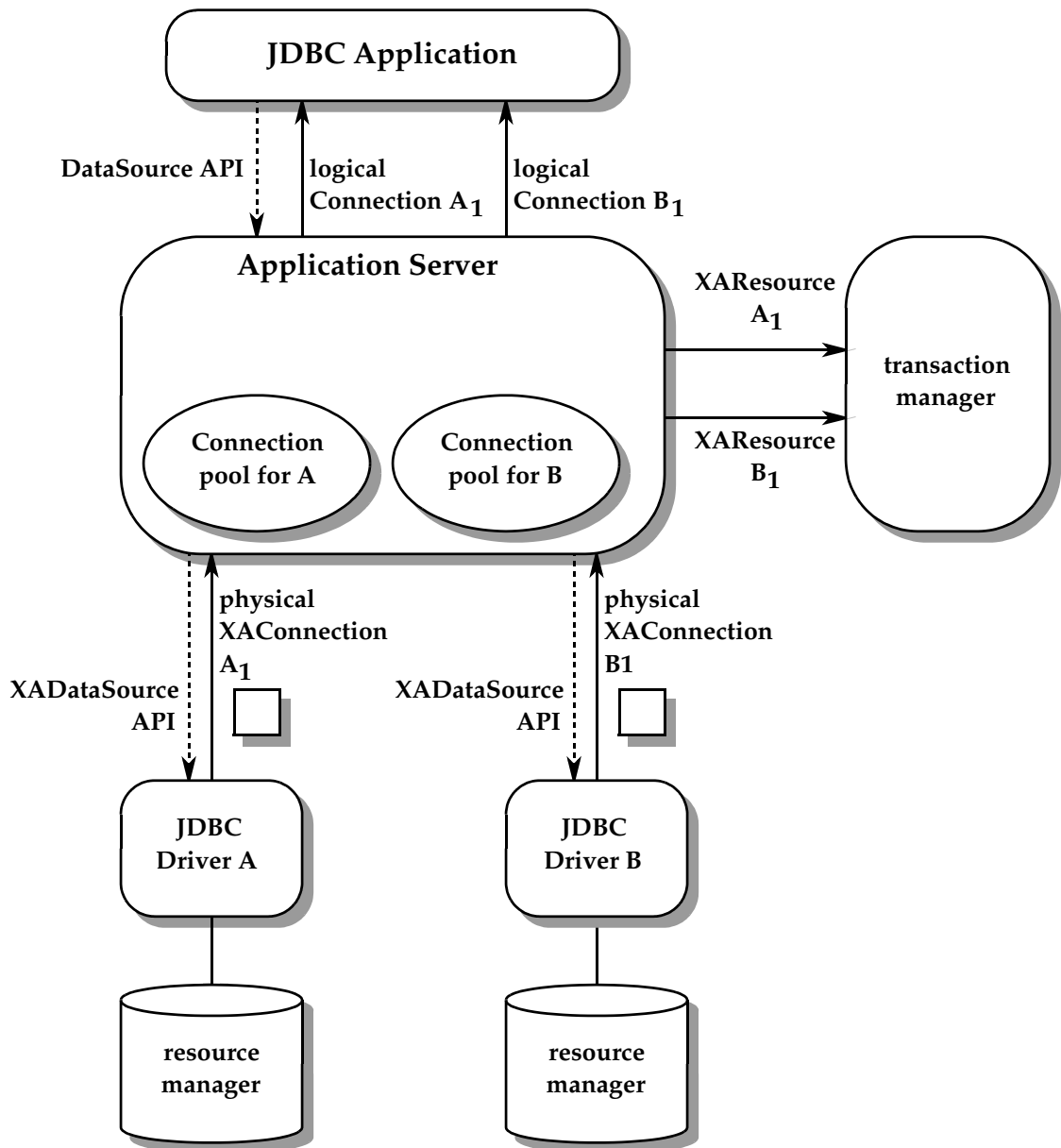
FIGURE 12-1 Infrastructure for distributed transactions

The following sections provide more detail on the components of this architecture.

12.2 XADataSource and XAConnection

The XADataSource and XAConnection interfaces, which are defined in the package javax.sql, are implemented by JDBC drivers that support distributed transactions. An XAConnection object is a pooled connection that can participate in a distributed transaction. More precisely, XAConnection extends the PooledConnection interface by adding the method getXAResource. This method produces an XAResource object that can be used by a transaction manager to coordinate the work done on this connection with the other participants in the distributed transaction. CODE EXAMPLE 12-1 gives the definition of the XAConnection interface.

```
public interface XAConnection extends PooledConnection {

                javax.transaction.xa.XAResource getXAResource()

                    throws SQLException;

}
```

CODE EXAMPLE 12-1   The XAConnection interface

Because they extend the PooledConnection interface, XAConnection objects support all the methods of PooledConnection objects. They are reusable physical connections to an underlying data source and produce logical connection handles that can be passed back to a JDBC application.

XAConnection objects are produced by an XADataSource object. There is some similarity between ConnectionPoolDataSource objects and XADataSource objects in that they are both implemented below a DataSource layer that is visible to the JDBC application. This architecture allows JDBC drivers to support distributed transactions in a way that is transparent to the application.
CODE EXAMPLE 12-2 shows the signatures for the two getXAConnection methods defined in XADataSource.

```
public interface XADataSource {

                XAConnection getXAConnection() throws SQLException;

                XAConnection getXAConnection(String user,

                    String password) throws SQLException;

                ...

}
```

CODE EXAMPLE 12-2   The XADataSource interface

Typically, `DataSource` implementations built on top of an `XADataSource` implementation will also include a connection pooling module.

12.2.1   Deploying an `XADataSource` Object

Deploying an `XADataSource` object is done in exactly the same manner as previously described for `ConnectionPoolDataSource  objects`. The two-step process includes deploying the `XADataSource` object and the application-visible `DataSource` object, as is done in CODE EXAMPLE 12-3.

```
// com.acme.jdbc.XADataSource implements the
// XADataSource interface.

// Create an instance and set properties.
com.acme.jdbc.XADataSource xads = new com.acme.jdbc.XADataSource();
xads.setServerName("bookstore");
xads.setDatabaseName("bookinventory");
xads.setPortNumber(9040);
xads.setDescription("XADataSource for inventory");

// First register xads with a JNDI naming service, using the
// logical name "jdbc/xa/inventory_xa"
Context ctx = new InitialContext();
ctx.bind("jdbc/xa/inventory_xa", xads);

// Next register the overlying DataSource object for application
// access. com.acme.appserver.DataSource is an implementation of
// the DataSource interface.

// Create an instance and set properties.
com.acme.appserver.DataSource ds =
                new com.acme.appserver.DataSource();
ds.setDescription("Datasource supporting distributed transactions");

// Reference the previously registered XADataSource
ds.setDataSourceName("jdbc/xa/inventory_xa");
```

```
// Register the DataSource implementation with a JNDI naming service,
// using the logical name "jdbc/inventory".
ctx.bind("jdbc/inventory", ds);
```

CODE EXAMPLE 12-3   Deploying a `DataSource` object backed by an `XADataSource` object

12.2.2   Getting a Connection

As in the connection pooling case, the application call to the method
`DataSource.getConnection` returns a logical handle produced by the physical
`XAConnection` object. The application code to get a logical connection is shown in
CODE EXAMPLE 12-4.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/inventory");
Connection con = ds.getConnection("myID","mypasswd");
```

CODE EXAMPLE 12-4   Application code to get a logical connection

CODE EXAMPLE 12-5 is an example of code from the middle-tier server's
implementation of the method `DataSource.getConnection`.

```
// Assume xads is a driver's implementation of XADataSource
XADataSource xads = (XADataSource)ctx.lookup("jdbc/xa/" +
                                             "inventory_xa");
// xacon implements XAConnection
XAConnection xacon = xads.getXAConnection("myID", "mypasswd");
// Get a logical connection to pass back up to the application
Connection con = xacon.getConnection();
```

CODE EXAMPLE 12-5   Getting a logical connection from an `XAConnection` object

12.3   XAResource

The `XAResource` interface is defined in the JTA specification and is the mapping in
the Java programming language of the X/Open Group `XA` interface. An
`XAResource` object is produced by calling the `XAConnection.getXAResource`
method and is used to associate an `XAConnection` object with a distributed
transaction. A given `XAConnection` object may be associated with at most one

transaction at a time. The JDBC driver maintains a one-to-one correspondence between an XAResource object and its associated XAConnection object; that is, multiple calls to the getXAResource method must all return the same object.

In a typical scenario, the middle-tier application server calls the method XAConnection.getXAResource and passes the returned object to an external transaction manager. The transaction manager uses the XAResource object exclusively—it does not access an XAConnection object directly.

The transaction manager coordinates the work of multiple XAResource objects, each of which represents a resource manager participating in the distributed transaction. Note that two XAResource objects may "point" to the same resource manager, that is, they may be associated with XAConnection objects that were produced by the same XADataSource.

The following XAResource methods are used by the transaction manager to implement a two-phase commit protocol. Each method takes an xid parameter that identifies the distributed transaction:

- start — tells the resource manager that the subsequent operations are part of the distributed transaction.
- end — marks the end of this resource manager's part of the distributed transaction.
- prepare — gets the resource manager's vote on whether to commit or roll back the distributed transaction.
- commit — tells the resource manager to commit its part of the distributed transaction. This method is invoked only if all the participating resource managers voted to commit the transaction.
- rollback — tells the resource manager to roll back its part of the distributed transaction. This method is invoked if one or more of the participating resource managers voted to roll back the transaction.

See the JTA specification for a complete description of the XAResouce interface.

12.4 Transaction Management

Participation in a distributed transaction is defined as the work done between invocations of the methods XAResource.start and XAResource.end. Outside these boundaries, the transaction mode is local, and a connection behaves exactly like a local connection.

With one exception, there is no difference in how an application participating in a distributed transaction is coded. In contrast to the local case, the boundaries of a distributed transaction must be controlled by an external transaction manager that is

coordinating the work of multiple connections. For this reason, it is an error for applications to call any of the following Connection methods while they are participating in a distributed transaction:

- setAutoCommit(true)
- commit
- rollback
- setSavepoint

The JDBC driver throws an SQLException if one of these operations is attempted on a connection that is participating in a distributed transaction. If the connection is later used for a local transaction, these operations are legal at that point.

Applications should also refrain from calling Connection.setTransactionIsolation within the bounds of a distributed transaction. The resulting behavior is implementation-defined.

If a connection has auto-commit mode already enabled at the time it joins a global transaction, the attribute will be ignored. The auto-commit behavior will resume when the connection returns to local transaction mode.

12.4.1   Two-phase Commit

The following steps outline how a transaction manager uses XAResource objects to implement the two-phase commit protocol. These steps assume a three-tier architecture where an application server is working with an external transaction manager:

1. The application server gets XAResource objects from two different connections:

```
// XAConA connects to resource manager A

javax.transaction.xa.XAResource resourceA = XAConA.getXAResource();

// XAConB connects to resource manager B
javax.transaction.xa.XAResource resourceB = XAConB.getXAResource();
```

CODE EXAMPLE 12-6   Getting the XAResource object from an XAConnection object

2. The application server passes the XAResource objects to the transaction manager. The transaction manager does not access the associated XAConnection objects directly.

3. The transaction manager uses the XAResource objects to assign a piece of the transaction to each of the associated resource managers. The transaction is identified by *xid*, which represents the identifier generated by the transaction manager when the transaction is created.

```
// Send work to resource manager A. The TMNOFLAGS argument indicates
// we are starting a new branch of the transaction, not joining or
// resuming an existing branch.
resourceA.start(xid, javax.transaction.xa.TMNOFLAGS);
// do work with resource manager A
...
// tell resource manager A that it's done, and no errors have occurred
resourceA.end(xid, javax.transaction.xa.TMSUCCESS);

// do work with resource manager B.
resourceB.start(xid, javax.transaction.xa.TMNOFLAGS);
// B's part of the distributed transaction
...
resourceB.end(xid, javax.transaction.xa.TMSUCCESS);
```

CODE EXAMPLE 12-7   Starting and ending transaction branches using the XAResource interface

4. The transaction manager initiates the two-phase commit protocol by asking each participant to vote:

```
resourceA.prepare(xid);
resourceB.prepare(xid);
```

CODE EXAMPLE 12-8   Initiating two-phase commit

A participating resource manager can vote to roll back the transaction by throwing a `javax.transaction.xa.XAException`.

5. If both participants vote to commit, the transaction manager tells each one to commit its piece of the distributed transaction (the second parameter tells the resource manager not to use a one phase commit protocol on behalf of the xid):

```
resourceA.commit(xid, false);
resourceB.commit(xid, false);
```

CODE EXAMPLE 12-9   Committing the distributed transaction

6. If either resource manager votes to roll back, the transaction manager tells each one to roll back its piece of the transaction:

```
resourceA.rollback(xid);
resourceB.rollback(xid);
```

CODE EXAMPLE 12-10   Rolling back the distributed transaction

The transaction manager is not required to use the same `XAResource` object to commit/rollback a transaction branch as was used to execute the branch. If different `XAResource` objects are used, however, they must be associated with `XAConnection` objects that connect to the same resource manager.

Note – Steps 1-6 also apply to the case where *XAConA* and *XAConB* are two physical connections to the same resource manager.

12.5 Closing the Connection

In a typical distributed transaction environment, the middle-tier server needs to be notified when an application has finished using a connection. As in the earlier discussion of `PooledConnection` objects, the middle-tier server will add itself as a `ConnectionEventListener` so that it will be notified when the application calls the method `Connection.close`. At this point, the server will notify the transaction manager so that it can end the transaction branch for the corresponding `XAResource` object. If the server's `DataSource` implementation includes connection pooling, the connection pooling module will be notified that it can return the physical `XAConnection` object to the pool.

Note – A distributed transaction may still be active after a participating `Connection` object is closed. This is not true for local transactions.

12.6 Limitations of the `XAResource` Interface

The `javax.transaction.xa.XAResource` interface is limited to defining only the set of methods needed to join and participate in global transactions, as defined by the X/Open XA standard. This allows any resource manager that implements the interface to participate with any other resource manager or transaction manager that has the same level of support.

Functionality that is not defined in the X/Open standard is correspondingly not defined in the `XAResource` interface. Resource managers that provide for support of features not defined in the X/Open XA standard, such as setting isolation levels in global transactions, will have to do so in an implementation-defined way.

Users who use implementation-defined features should be aware that they will limit the portability of their applications.

Statements

This section describes the Statement interface and its subclasses, PreparedStatement and CallableStatement. It also describes related topics, including escape syntax, performance hints, and auto-generated keys.

13.1 The Statement Interface

The Statement interface defines methods for executing SQL statements that do not contain parameter markers. The PreparedStatement interface adds methods for setting input parameters, and the CallableStatement interface adds methods for retrieving output parameter values returned from stored procedures.

Note – Any reference to the Statement interface within this specification includes its subclasses PreparedStatement and CallableStatement unless stated otherwise.

13.1.1 Creating Statements

Statement objects are created by Connection objects, as is done in CODE EXAMPLE 13-1.

Connection conn = dataSource.getConnection(user, passwd);

```
Statement stmt = conn.createStatement()
```

CODE EXAMPLE 13-1   Creating a Statement object

Each Connection object can create multiple Statement objects that may be used concurrently by the program. This is demonstrated in CODE EXAMPLE 13-2.

```
// get a connection from the DataSource object ds
Connection conn = ds.getConnection(user, passwd);
// create two instances of Statement
Statement stmt1 = conn.createStatement();
Statement stmt2 = conn.createStatement();
```

CODE EXAMPLE 13-2 Creating multiple Statement objects from a single connection

13.1.1.1   Setting `ResultSet` Characteristics

Additional constructors may be used to set the type and concurrency or the type, concurrency, and holdability of any result sets produced by a statement. See Chapter 16 "Result Sets" for more on the ResultSet interface.

CODE EXAMPLE 13-3 creates a `Statement` object that returns result sets that are scrollable, that are insensitive to changes made while the `ResultSet` object is open, that can be updated, and that do not close the ResultSet objects when `commit` is called.

```
Connection conn = ds.getConnection(user, passwd);
Statement stmt = conn.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE,
                    ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

CODE EXAMPLE 13-3 Creating a scrollable, insensitive, updatable result set that stays open after the method `commit` is called

See Chapter 16 "Result Sets" for more information on `ResultSet` types.

13.1.2   Executing `Statement` Objects

The method used to execute a `Statement` object depends on the type of SQL statement being executed. If the `Statement` object represents an SQL query returning a ResultSet object, the method executeQuery should be used. If the SQL is known to be a DDL statement or a DML statement returning an update count, the method executeUpdate should be used. If the type of the SQL statement is not known, the method execute should be used.

13.1.2.1   Returning a ResultSet object

CODE EXAMPLE 13-4 shows the execution of an SQL string returning a `ResultSet` object.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select TITLE, AUTHOR, ISBN " +
                                 "from BOOKLIST");
while (rs.next()){
            ...
}
```

CODE EXAMPLE 13-4   Executing a Statement object that returns a ResultSet object

If the SQL string being executed does not return a `ResultSet` object, the method executeQuery throws an `SQLException`.

13.1.2.2   Returning an Update Count

In CODE EXAMPLE 13-5, the SQL statement being executed returns the number of rows affected by the update for SQL Data Manipulation Language (DML) statements or 0 for SQL statements that return nothing.

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("update STOCK set ORDER = 'Y' " +
                              "where SUPPLY = 0");
if (rows > 0) {
            ...
}
```

CODE EXAMPLE 13-5   Executing a `Statement` object that returns an update count

The method executeUpdate throws an SQLException if the SQL string being executed does not return an update count.

13.1.2.3   Using the Method `execute`

The method execute should be used only when the SQL string being executed could return either an update count or a `ResultSet` object. The execute method returns `true` if the first result is a `ResultSet` object and `false` if it is an update count. Additional methods must be called to retrieve the `ResultSet` object or update count or to retrieve additional results, if any.

```
String sql;

...

Statement stmt = conn.createStatement();
boolean b = stmt.execute(sql);
if (b == true) {
                // b is true if a ResultSet is returned
                ResultSet rs;
                rs = stmt.getResultSet();
                while (rs.next()) {
                ...
                }
} else {
                // b is false if an update count is returned
                int rows = stmt.getUpdateCount();
                if (rows > 0) {
                ...
                }
}
```

CODE EXAMPLE 13-6 Executing a Statement object that may return an update count or a
                      `ResultSet` object

When the SQL string being executed returns a `ResultSet` object, the method
getUpdateCount returns `-1`. If the SQL string being executed returns an update
count, the method `getResultSet` returns null.

13.1.3 Closing `Statement` Objects

An application calls the method `Statement.close` to indicate that it has finished
processing a statement. All `Statement` objects will be closed when the connection
that created them is closed. However, it is good coding practice for applications to
close statements as soon as they have finished processing them. This allows any
external resources that the statement is using to be released immediately.

Closing a `Statement` object will close and invalidate any instances of `ResultSet`
produced by that `Statement` object. The resources held by the `ResultSet` object
may not be released until garbage collection runs again, so it is a good practice to
explicitly close `ResultSet` objects when they are no longer needed.

These comments about closing `Statement` objects apply to `PreparedStatement`
and `CallableStatement` objects as well.

13.2 The `PreparedStatement` Interface

The `PreparedStatement` interface extends `Statement`, adding the ability to set values for parameter markers contained within the statement.

PreparedStatement objects represent SQL statements that can be prepared, or precompiled, for execution once and then executed multiple times. Parameter markers, represented by "?" in the SQL string, are used to specify input values to the statement that may vary at runtime.

13.2.1 Creating a `PreparedStatement` Object

An instance of `PreparedStatement` is created in the same manner as a `Statement` object, except that the SQL command is supplied when the statement is created:

```
Connection conn = ds.getConnection(user, passwd);
PreparedStatement ps = conn.prepareStatement("INSERT INTO BOOKLIST" +
                         "(AUTHOR, TITLE, ISBN) VALUES (?, ?, ?)");
```

CODE EXAMPLE 13-7 Creating a PreparedStatement object with three placeholder markers

13.2.1.1 Setting `ResultSet` Characteristics

As with createStatement, the method prepareStatement defines a constructor that can be used to specify the characteristics of result sets produced by that prepared statement.

```
Connection conn = ds.getConnection(user, passwd);
PreparedStatement ps = conn.prepareStatement(
            "SELECT AUTHOR, TITLE FROM BOOKLIST WHERE ISBN = ?",
            ResultSet.TYPE_FORWARD_ONLY,
            ResultSet.CONCUR_UPDATABLE);
```

CODE EXAMPLE 13-8 Creating a PreparedStatement object that returns forward only, updatable result sets

13.2.2 Setting Parameters

The `PreparedStatement` interface defines setter methods that are used to substitute values for each of the parameter markers in the precompiled SQL string. The names of the methods follow the pattern "set<Type>".

For example, the method setString is used to specify a value for a parameter marker that expects a string. Each of these setter methods takes at least two parameters. The first is always an `int` equal to the ordinal position of the parameter to be set, starting at 1. The second and any remaining parameters specify the value to be assigned to the parameter.

```
PreparedStatement ps = conn.prepareStatement("INSERT INTO BOOKLIST" +
                        "(AUTHOR, TITLE, ISBN) VALUES (?, ?, ?)");
ps.setString(1, "Zamiatin, Evgenii");
ps.setString(2, "We");
ps.setLong(3, 0140185852);
```

CODE EXAMPLE 13-9   Setting parameters in a `PreparedStatement` object

A value must be provided for each parameter marker in the PreparedStatement object before it can be executed. The methods used to execute a PreparedStatement object (executeQuery, executeUpdate and execute) will throw an SQLException if a value is not supplied for a parameter marker.

The values set for the parameter markers of a `PreparedStatement` object are not reset when it is executed. The method `clearParameters` can be called to explicitly clear the values that have been set. Setting a parameter with a different value will replace the previous value with the new one.

Note – If, in the execution of a `PreparedStatement` object, the JDBC driver reads values set for the parameter markers by the methods `setAsciiStream`, `setBinaryStream`, `setCharacterStream`, `setNCharacterStream` or `setUnicodeStream`, those parameters must be reset prior to the next execution of the `PreparedStatement` object otherwise a `SQLException` will be thrown.   A `SQLException` will also be thrown if the same stream is used by multiple parameter markers within the same `PreparedStatement` object execution.

13.2.2.1　Type Conversions

The data type specified in a `PreparedStatement` setter method is a data type in the Java programming language. The JDBC driver is responsible for mapping this to the corresponding JDBC type (one of the SQL types defined in `java.sql.Types`) so that it is the appropriate type to be sent to the data source. The default mapping is specified in Appendix B TABLE B-2.

13.2.2.2   National Character Set Conversions

SQL:2003 provides support for national character set types, which are described in the SQL:2003 specification to be an implementation defined character set. The following JDBC Types may be used to access national character set types: `NCHAR`, `NVARCHAR`, `LONGNVARCHAR`, and `NCLOB`. These types are analogous to the types `CHAR`, `VARCHAR`, `LONGVARCHAR` and `CLOB`, except that the values are encoded using an alternate character set, the national character set. Since Java types encode character data using UTF-16, there is no reason to use an alternate Java type to hold these values. However, there may be an advantage to distinguishing `CLOB` from `NCLOB`. The JDBC specification uses Strings to represent `NCHAR`, `NVARCHAR`, and `LONGNVARCHAR` data, automatically converting between the Java character set and the national character set. JDBC uses `NClob` to represent `NCLOB` values. There is no automatic conversion between `Clob` and `NClob` values. Please refer to the Java API docs for `java.lang.Character` for additional information on how the Java Language uses Unicode.

For maximum portability, an application must indicate to the JDBC driver when a particular value corresponds to a national character type. When specifying a value for a parameter marker which is a national character type, the application would call the `setNString`, `setNCharacterStream`, `setNClob`, or `setObject` method. If the `setObject` method is used , the target data type must be specified as `Types.NCHAR`, `Types.NCLOB`, `Types.NVARCHAR`, or `Types.LONGNVARCHAR`. In the event that an application does not indicate that a parameter marker value corresponds to a national character type, the JDBC driver may interpret the value incorrectly, resulting in the possibility of a data conversion error. In situations where the JDBC driver can detect that a data conversion error might occur, the call to the `setXXX` method will result in a `SQLException` being thrown. It might not always possible for the driver to detect that a data conversion error may occur.

If the driver does not support national character types, an attempt to invoke the methods `setNString`, `setNCharacterStream`, `setNClob` or `setObject`, specifying a target data type as a national character set, may result in a `SQLException` being thrown.

To retrieve a national character value, an application would call the method `getNString`, `getNClob`, `getNCharacterStream`, or `getObject`.

13.2.2.3   Type Conversions Using the Method `setObject`

The method `setObject` can be used to convert an object in the Java programming language to a JDBC type.

The conversion is explicit when setObject is passed a Java Object and a JDBC data type. The driver will attempt to convert the Object to the specified JDBC type before passing it to the data source. If the object cannot be converted to the target type, an SQLException object is thrown. In CODE EXAMPLE 13-10, a Java Object of type Integer is being converted to the JDBC type SHORT.

```
Integer value = new Integer(15);
ps.setObject(1, value, java.sql.Types.SHORT);
```

CODE EXAMPLE 13-10 Converting an Integer object to an SQL SHORT

If setObject is called without a type parameter, the Java Object is implicitly mapped using the default mapping for that object type.

```
Integer value = new Integer(15);
// value is mapped to java.sql.Types.INTEGER
ps.setObject(1, value);
```

CODE EXAMPLE 13-11 The method setObject using the default mapping

The default mapping is described in Appendix B TABLE B-4

Note – The method setObject will do custom mapping for SQL UDTs that have a custom mapping. See Chapter 17 "Customized Type Mapping" for more information.

13.2.2.4   Setting NULL Parameters

The method setNull can be used to set any parameter to JDBC NULL. It takes two parameters, the ordinal position of the parameter marker, and the JDBC type of the parameter.

```
ps.setNull(2, java.sql.Types.VARCHAR);
```

CODE EXAMPLE 13-12 Setting a String parameter to JDBC NULL

If a Java null is passed to any of the setter methods that take a Java object, the parameter will be set to JDBC NULL.

13.2.2.5   Clearing Parameters

The values set for IN parameter markers of a `PreparedStatement` object can be explicitly cleared by invoking the `clearParameters` method. Any resources that are used by `PreparedStatement` objects to represent the set values are also released.

13.2.3   Describing Outputs and Inputs of a `PreparedStatement` Object

The method PreparedStatement.getMetaData retrieves a ResultSetMetaData object containing a description of the columns that will be returned by a prepared statement when is it executed. The ResultSetMetaData object contains a record for each column being returned. Methods in the ResultSetMetaData interface provide information about the number of columns being returned and the characteristics of each column.

```
PreparedStatement pstmt = conn.prepareStatement(
                           "SELECT * FROM CATALOG");
ResultSetMetaData rsmd = pstmt.getMetaData();
int colCount = rsmd.getColumnCount();
int colType;
String colLabel;
for (int i = 1; i <= colCount; i++) {
                colType = rsmd.getColumnType(i);
                colLabel = rsmd.getColumnLabel(i);
                ...
}
```

CODE EXAMPLE 13-13 Creating a ResultSetMetaData object and retrieving column information from it

The method `PreparedStatement.getParameterMetaData` returns a `ParameterMetaData` object describing the parameter markers that appear in the `PreparedStatement` object. Methods in the `ParameterMetaData` interface provide information about the number of parameters and their characteristics.

```
PreparedStatement pstmt = conn.prepareStatement(
                           "SELECT * FROM BOOKLIST WHERE ISBN = ?");
...
```

```
ParameterMetaData pmd = pstmt.getParameterMetaData();

int colType = pmd.getParameterType(1);

...
```

CODE EXAMPLE 13-14 Creating a ParameterMetaData object and retrieving parameter
information from it

See the API specification for more details.

13.2.4   Executing a `PreparedStatement` Object

As with Statement objects, the method used to execute a PreparedStatement object
depends on the type of SQL statement being executed. If the PreparedStatement
object is a query returning a ResultSet object, it should be executed with the method
executeQuery. If it is a DML statement returning a row count, it should be executed
with the method executeUpdate. The method execute should be used only if the
return type of the statement is unknown.

If any of the PreparedStatement execute methods is called with an SQL string as a
parameter, an SQLException is thrown.

13.2.4.1   Returning a `ResultSet` Object

CODE EXAMPLE 13-15 shows a query being prepared and then executed multiple
times.

```
PreparedStatement pstmt = conn.prepareStatement("SELECT AUTHOR, " +
                             "TITLE FROM BOOKLIST WHERE SECTION = ?");

for (int i = 1; i <= maxSectionNumber; i++) {
            pstmt.setInt(1, i);
            ResultSet rs = pstmt.executeQuery();
            while (rs.next()) {
                // process the record
            }
            rs.close();
}
pstmt.close();
```

CODE EXAMPLE 13-15 Preparing and executing a statement returning a result set

If the statement being executed does not return a ResultSet object an SQLException
is thrown by executeQuery.

13.2.4.2 Returning a Row Count

In CODE EXAMPLE 13-16, the SQL statement being prepared and executed returns the number of rows affected by the update for SQL Data Manipulation Language (DML) statements or 0 for SQL statements that return nothing.

```
PreparedStatement pstmt = conn.prepare(
                    "update stock set reorder = 'Y' where stock < ?");
pstmt.setInt(1, 5);
int num = pstmt.executeUpdate();
```

CODE EXAMPLE 13-16 Preparing and executing a statement returning an update count

If the statement being executed returns a ResultSet object, an SQLException is thrown.

13.2.4.3 Using the Method `execute`

If the return type of a PreparedStatement object is not known, it should be executed with the execute method. As is true with Statement objects, the methods getResultSet and getUpdateCount can be used to retrieve a result.

```
PreparedStatement pstmt = conn.prepareStatement(sqlStatement);
// set any parameters the user passes
...

boolean b = pstmt.execute();
if (b == true) {
                ResultSet rs = pstmt.getResultSet();
                // process a ResultSet
                ...
                }
} else {
                int rowCount = pstmt.getUpdateCount();
                // process row count
                ...
                }
}
```

13.3 The `CallableStatement` Interface

The `CallableStatement` interface extends `PreparedStatement` with methods for executing and retrieving results from stored procedures.

13.3.1 Creating a CallableStatement Object

As with Statement and PreparedStatement objects, CallableStatement objects are created by Connection objects. CODE EXAMPLE 13-18 shows the creation of a CallableStatement object for calling the stored procedure 'validate', which has a return parameter and two other parameters.

```
CallableStatement cstmt = conn.prepareCall(
                          "{? = call validate(?, ?)}");
```

CODE EXAMPLE 13-18 Creating a CallableStatement object

All the examples in this chapter use the escape syntax for calling stored procedures. See "Stored Procedures and Functions" on page 113.

13.3.2 Setting Parameters

CallableStatement objects may take three types of parameters: IN, OUT, and INOUT. The parameter can be specified as either an ordinal parameter or a named parameter. A value must be set for each parameter marker in the statement that represents an IN or INOUT parameter. The `registerOutParameter` method must be called for each parameter marker which represents an OUT or INOUT parameter.

The number, type, and attributes of parameters to a stored procedure can be determined using the DatabaseMetaData method getProcedureColumns.

Parameter ordinals, which are integers passed to the appropriate setter method, refer to the parameter markers ("?") in the statement, starting at one. Literal parameter values in the statement do not increment the ordinal value of the parameter markers. In CODE EXAMPLE 13-19, the two parameter markers have the ordinal values 1 and 2.

```
CallableStatement cstmt = con.prepareCall(
"{CALL PROC(?, "Literal_Value", ?)}");
```

```
cstmt.setString(1, "First");
cstmt.setString(2, "Third");
```

CODE EXAMPLE 13-19 Specifying ordinal parameters

Named parameters can also be used to specify specific parameters. This is especially useful when a procedure has many parameters with default values. Named parameters can be used to specify only the values that have no default value. The name of a parameter corresponds to the COLUMN_NAME field returned by DatabaseMetaData.getProcedureColumns.

In CODE EXAMPLE 13-20, the procedure COMPLEX_PROC takes ten parameters, but only the first and fifth parameters, PARAM_1 and PARAM_5, are required.

```
CallableStatement cstmt = con.prepareCall(
                "{CALL COMPLEX_PROC(?, ?)}";
cstmt.setString("PARAM_1", "Price");
cstmt.setFloat("PARAM_5", 150.25);
```

CODE EXAMPLE 13-20 Specifying two input parameters to a stored procedure

Additional methods in the CallableStatement interface allow parameters to be registered and retrieved by name.

The DatabaseMetaData.supportsNamedParameters method can be called to determine if a JDBC driver and underlying data source support specifying named parameters.

It is not possible to combine setting parameters with ordinals and with names in the same statement. If ordinals and names are used for parameters in the same statement, an SQLException is thrown.

Note – In some cases it may not be possible to provide only some of the parameters for a procedure. For example, if the procedure name is overloaded, the data source determines which procedure to call based on the number of parameters. Enough parameters must be provided to allow the data source to resolve any ambiguity.

13.3.2.1   IN Parameters

IN parameters are assigned values using the setter methods as described in "Setting Parameters" on page 98. In CODE EXAMPLE 13-21, a string parameter and a date parameter are set.

```
cstmt.setString(1, "October");
cstmt.setDate(2, date);
```

13.3.2.2   OUT Parameters

The method `registerOutParameter` must be called to set the type for each OUT parameter before a `CallableStatement` object is executed. When the stored procedure returns from execution, it will use these types to set the values for any OUT parameters.

The values of OUT parameters can be retrieved using the appropriate getter methods defined in the `CallableStatement` interface. CODE EXAMPLE 13-22 shows the execution of a stored procedure with two OUT parameters, a string and float, and the retrieval of the OUT parameter values.

```
CallableStatement cstmt = conn.prepareCall(
                "{CALL GET_NAME_AND_NUMBER(?, ?)}");
cstmt.registerOutParameter(1, java.sql.Types.STRING);
cstmt.registerOutParameter(2, java.sql.Types.FLOAT);
cstmt.execute();
// Retrieve OUT parameters
String name = cstmt.getString(1);
float number = cstmt.getFloat(2);
```

CODE EXAMPLE 13-22 Registering and retrieving OUT parameters

13.3.2.3   INOUT Parameters

Parameters that are both input and output parameters must be both set by using the appropriate setter method and also registered by calling the `registerOutParameter` method. The type implied by the setter method (see TABLE B-1 in Appendix B "Data Type Conversion Tables") and the type supplied to the method registerOutParameter must be the same.

CODE EXAMPLE 13-23 shows the stored procedure calc, which takes one INOUT float parameter.

```
CallableStatement cstmt = conn.prepareCall("{CALL CALC(?)}");
cstmt.setFloat(1, 1237.98f);
ctsmt.registerOutParameter(1, java.sql.Types.FLOAT);
cstmt.execute();
float f = cstmt.getFloat(1);
```

CODE EXAMPLE 13-23 Executing a CallableStatement object with an INOUT parameter

13.3.2.4 Clearing Parameters

The values set for IN parameter markers or registered for OUT parameter markers of a `CallableStatement` object can be explicitly cleared by invoking the `clearParameters` method. Any resources that are used by `CallableStatement` objects to represent the set or registered values are also released.

13.3.3 Executing a `CallableStatement` Object

As with Statement and PreparedStatement objects, the method used to execute a CallableStatement object depends on whether it returns a single `ResultSet` object, an update count, or multiple mixed results.

13.3.3.1 Returning a Single `ResultSet` Object

CODE EXAMPLE 13-24 shows the execution of a `CallableStatement` object that takes one input parameter and returns a single `ResultSet` object.

```
CallableStatement cstmt = conn.prepareCall("{CALL GETINFO(?)}");
cstmt.setLong(1, 1309944422);
ResultSet rs = cstmt.executeQuery();
// process the results
while (rs.next()) {
                ...
}
rs.close();
cstmt.close();
```

CODE EXAMPLE 13-24 Executing a CallableStatement object that returns a single result set

The method executeQuery throws an SQLException if the stored procedure does not return a ResultSet object.

13.3.3.2 Returning a Row Count

CODE EXAMPLE 13-25 shows the execution of a `CallableStatement` object that returns a row count.

```
CallableStatement cstmt = conn.prepareCall("{call GETCOUNT(?)}");
cstmt.setString(1, "Smith");
int count = cstmt.executeUpdate();
cstmt.close();
```

CODE EXAMPLE 13-25 Executing a CallableStatement object returning an update count

If the stored procedure does not return a update count, the method `executeUpdate` throws an `SQLException`.

13.3.3.3   Returning Unknown or Multiple Results

If the type or number of results returned by a `CallableStatement` object are not known until run time, the `CallableStatement` object should be executed with the method execute. The methods getMoreResults, getUpdateCount, and getResultSet can be used to retrieve all the results.

The method execute returns `true` if the first result is a `ResultSet` object and `false` if it is an update count.

When the method execute returns true, the method getResultSet is called to retrieve the ResultSet object. When execute returns false, the method getUpdateCount returns an `int`. If this number is greater than or equal to zero, it indicates the update count returned by the statement. If it is -1, it indicates that there are no more results.

If multiple results are being returned, the method getMoreResults can be called to get the next result. As with the method `execute`, getMoreResults will return true if the next result is a ResultSet object and false if the next result is a row count or no more result are available.

CODE EXAMPLE 13-26 shows how to retrieve all the results from a `CallableStatment` object.

```
CallableStatement cstmt = conn.prepareCall(procCall);
boolean retval = cstmt.execute();
ResultSet rs;
int count;
do {
                if (retval == false) {
                    count = cstmt.getUpdateCount();
                    if (count == -1) {
                        // no more results
                        break;
                    } else {
```

```
                          // process row count
                      }
                  } else { // ResultSet
                      rs = cstmt.getResultSet();
                      // process ResultSet
                  }
                  retval = cstmt.getMoreResults();
while (true);
```

CODE EXAMPLE 13-26 Executing a callable statement that returns multiple results

By default, each call to the method getMoreResults closes any previous
ResultSet object returned by the method getResultSet. However, the method
getMoreResults may take a parameter that specifies whether a ResultSet object
returned by getResultSet should be closed. The Statement interface defines
three constants that can be supplied to the method getMoreResults:

- CLOSE_CURRENT_RESULT — indicates that the current ResultSet object should
  be closed when the next ResultSet object is returned

- KEEP_CURRENT_RESULT — indicates that the current ResultSet object should
  not be closed when the next ResultSet object is returned

- CLOSE_ALL_RESULTS — indicates that any ResultSet objects that have been
  kept open should be closed when the next result is returned

If the current result is an update count and not a ResultSet object, any parameter
passed to getMoreResults is ignored.

To determine whether a driver implements this feature, an application can call the
DatabaseMetaData method supportsMultipleOpenResults.

```
ResultSet rs1 = cstmt.getResultSet();
rs1.next();
...
retval = cstmt.getMoreResults(Statement.KEEP_CURRENT_RESULT);
if (retval == true) {
    ResultSet rs2 = cstmt.getResultSet();
    rs2.next();
    ...
    rs1.next();
}
retval = cstmt.getMoreResults(Statement.CLOSE_ALL_RESULTS);
...
```

13.4   Escape Syntax

The SQL string used in a `Statement` object may include JDBC *escape syntax*. Escape syntax allows the driver to more easily scan for syntax that requires special processing. Implementing this special processing in the driver layer improves application portability.

Special escape processing might be needed for the following:

- commonly used features that do not have standard syntax defined by SQL, or where the native syntax supported by the underlying data source varies widely among vendors. The driver may translate the escape syntax to a specific native syntax in this case.

- features that are not supported by the underlying data source but are implemented by the driver.

Escape processing for a Statement object is turned on or off using the method setEscapeProcessing, with the default being on. The RowSet interface also includes a `setEscapeProcessing` method. The RowSet method applies to the SQL string used to populate a RowSet object. The `setEscapeProcessing` method does not work for a `PreparedStatement` object because its SQL string may have been precompiled when the `PreparedStatement` object was created.

JDBC defines escape syntax for the following:

- scalar functions
- date and time literals
- outer joins
- calling stored procedures and functions
- escape characters for LIKE clauses

13.4.1   Scalar Functions

Almost all underlying data sources support numeric, string, time, date, system, and conversion functions on scalar values. The escape syntax to access a scalar function is:

{fn <function-name> (argument list)}

For example, the following code calls the function concat with two arguments to be concatenated:

{fn concat("Hot", "Java")}

The following syntax gets the name of the current database user:

{fn user()}

Scalar functions may be supported by different data sources with slightly different native syntax, and they may not be supported by all drivers. The driver will either map the escaped function call into the native syntax or implement the function directly.

Various DatabaseMetaData methods list the functions that are supported. For example, the method getNumericFunctions returns a comma-separated list of the Open Group CLI names of numeric functions, the method getStringFunctions returns string functions, and so on.

Appendix 2 "Scalar Functions" provides a list of the scalar functions a driver is expected to support. A driver is required to implement these functions only if the data source supports them, however.

The escape syntax for scalar functions must only be used to invoke the scalar functions defined in Appendix 2 "Scalar Functions". The escape syntax is not intended to be used to invoke user-defined or vendor specific scalar functions.

13.4.2   Date and Time Literals

Data sources differ widely in the syntax they use for date, time, and timestamp literals. The JDBC API supports ISO standard format for the syntax of these literals, using an escape clause that the driver translates to native syntax.

The escape syntax for date literals is:

{d 'yyyy-mm-dd'}

The driver will replace the escape clause with the equivalent native representation. For example, the driver might replace {d '1999-02-28'} with '28-FEB-99' if that is the appropriate format for the underlying data source.

The escape syntax for TIME and TIMESTAMP literals are:

{t 'hh:mm:ss'}

{ts 'yyyy-mm-dd hh:mm:ss.f . . .'}

The fractional seconds (.f . . .) portion of the timestamp can be omitted.

13.4.3   Outer Joins

Outer joins are an advanced feature and are not supported by all data sources. Consult relevant SQL documentation for an explanation of outer joins.

The escape syntax for an outer join is:

{oj <outer-join>}

where <outer-join> has the form:

table {LEFT | RIGHT | FULL} OUTER JOIN {table | <outer-join>} ON search-condition

(Note that curly braces ({}) in the preceding line indicate that one of the items between them must be used; they are not part of the syntax.) The following SELECT statement uses the escape syntax for an outer join.

```
Statement stmt = con.createStatement();
stmt.executeQuery("SELECT * FROM {oj TABLE1 " +
                "LEFT OUTER JOIN TABLE2 ON DEPT_NO = 003420930}");
```

The JDBC API provides three DatabaseMetaData methods for determining the kinds of outer joins a driver supports: supportsOuterJoins, supportsFullOuterJoins, and supportsLimitedOuterJoins.

13.4.4   Stored Procedures and Functions

If a database supports stored procedures, they can be invoked using JDBC escape syntax as follows:

{call <procedure_name> [(<argument-list>)]}

or, where a procedure returns a result parameter:

{? = call <procedure_name> [(<argument-list>)]}

The square brackets indicate that the (argument-list) portion is optional. Input arguments may be either literals or parameter markers. See "Setting Parameters" on page 105 for information on parameters.

The method `DatabaseMetaData.supportsStoredProcedures` returns `true` if the database supports stored procedures.

JDBC drivers may optionally provide support for invoking user-defined or vendor defined functions using the escape syntax for stored procedures.

The method `DatabaseMetaData.supportsStoredFunctionsUsingCallSyntax` returns `true` if the database supports invoking user-defined or vendor defined functions using the escape syntax for stored procedures. Please consult the documentation for your JDBC driver for additional information

13.4.5   LIKE Escape Characters

The percent sign (%) and underscore (_) characters are wild card characters in SQL LIKE clauses (% matches zero or more characters, and _ matches exactly one character). In order to interpret them literally, they can be preceded by a backslash (\), which is a special escape character in strings. One can specify which character to use as the escape character by including the following syntax at the end of a LIKE predicate:

{escape '<escape-character>'}

For example, the following query uses the backslash as an escape character, and finds identifier names that begin with an underscore. Note that the Java compiler will not recognize the backslash as a character unless it is preceded by a backslash.

```
stmt.executeQuery("SELECT name FROM Identifiers " +
                "WHERE Id LIKE '\\_%' {escape '\\'}");
```

13.5   Performance Hints

The `Statement` interface has two methods that can be used to provide hints to a JDBC driver: setFetchDirection and setFetchSize. The values supplied to these methods are applied to each result set produced by the statement. The methods of the same name in the `ResultSet` interface can be used to supply hints for just that result set.

Hints provided to the driver via this interface may be ignored by the driver if they are not appropriate.

The methods getFetchDirection and getFetchSize return the current value of the hints. If either of these methods is called before the corresponding setter method has been called, the value returned is implementation-defined.

13.6   Retrieving Auto Generated Keys

Many database systems have a mechanism that automatically generates a unique
key field when a row is inserted. The method `Statement.getGeneratedKeys`,
which can be called to retrieve the value of such a key, returns a `ResultSet` object
with a column for each automatically generated key. A flag indicating that any auto
generated columns should be returned is passed to the methods execute,
executeUpdate or prepareStatement when the statement is executed or prepared.

```
Statement stmt = conn.createStatement();

// indicate that the key generated is going to be returned

int rows = stmt.executeUpdate("INSERT INTO ORDERS " +
                              "(ISBN, CUSTOMERID) " +
                              "VALUES (195123018, 'BILLG')",
                              Statement.RETURN_GENERATED_KEYS);

ResultSet rs = stmt.getGeneratedKeys();

boolean b = rs.next();

if (b == true) {

            // retrieve the new key value

            ...

}
```

CODE EXAMPLE 13-28 Retrieving auto generated keys

Additional methods allow the ordinals or names of the columns that should be
returned to be specified as an array. If the columns are not specified, the JDBC driver
implementation will determine the columns which best represent the auto-generated
keys.

In CODE EXAMPLE 13-29 the Statement method executeUpdate is called with two
parameters, the first is the SQL statement to be executed, the second is an array of
String containing the column name that should be returned when
`getGeneratedKeys` is called:

```
String keyColumn[] = {"ORDER_ID"};

...

Statement stmt = conn.createStatement();

int rows = stmt.executeUpdate("INSERT INTO ORDERS " +
                              "(ISBN, CUSTOMERID) " +
```

```
                              "VALUES (966431502, 'BILLG')",
                              keyColumn);
ResultSet rs = stmt.getGeneratedKeys();

....
```

CODE EXAMPLE 13-29 Retrieving a named column using executeUpdate and
                       getGeneratedKeys

The result of invoking the method `getGeneratedKeys` when auto-commit is `true`
is implementation-defined. In order to increase application portability when
retrieving auto generated keys, the `Connection` auto-commit attribute should be
set to `false`.

See the API Specification for more details.

Calling `ResultSet.getMetaData` on the `ResultSet` object returned by
`getGeneratedKeys` will produce a ResultSetMetaData object that can be used to
determine the number, type and properties of the generated keys.

In some cases, such as in an insert select statement, more than one key may be
returned. The `ResultSet` object returned by `getGeneratedKeys` will contain a
row for each key that a statement generated. If no keys are generated, an empty
result set will be returned.

The concurrency of the `ResultSet` object returned by `getGeneratedKeys` must be
`CONCUR_READ_ONLY`. The type of the `ResultSet` object must be either
`TYPE_FORWARD_ONLY` or `TYPE_SCROLL_INSENSITIVE`.

The method `DatabaseMetaData.supportsGetGeneratedKeys` returns true if a
JDBC driver and underlying data source support the retrieval of automatically
generated keys. If a value of true is returned by the call to
`supportsGetGeneratedKeys`, a JDBC driver must support the return of auto-
generated keys for SQL INSERT statements. Some JDBC driver implementations
might also support auto-generated keys with SQL statements other than INSERT.
Please consult your JDBC driver documentation for details.

Batch Updates

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

14.1 Description of Batch Updates

14.1.1 Statements

The batch update facility allows a `Statement` object to submit a set of heterogeneous SQL statements together as a single unit, or batch, to the underlying data source.

Since the JDBC 2.0 API, a `Statement` object has had 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. An application adds commands to a statement's batch one at a time by calling the method `Statement.addBatch` and providing it with the SQL statement to be added. All of the commands added to a batch must be statements that return an update count.

If an application decides not to submit a batch of commands that has been constructed for a statement, it can call the method `Statement.clearBatch` to clear the batch of all commands.

In CODE EXAMPLE 14-1, all of the SQL statements 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();
```

CODE EXAMPLE 14-1   Creating and executing a batch of insert statements

In the example, auto-commit mode is disabled to prevent the driver from committing the transaction when `Statement.executeBatch` is called. Disabling auto-commit 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, auto-commit should always be turned off when batch updates are done. The commit behavior of `executeBatch` is always implementation-defined when an error occurs and auto-commit is `true`.

It is not possible to set a savepoint "within" a batch of statements to enable partial recovery. If a savepoint is set any time before the method `executeBatch` is called, it is set before any of the statements that have been added to the batch are executed.

Although the focus in this section is on using `Statement` objects to do batch updates, the discussion that follows applies to `PreparedStatment` and `CallableStatement` objects as well.

14.1.2   Successful Execution

The `Statement.executeBatch` method submits a statement's batch to the underlying data source for execution. Batch commands are executed serially (at least logically) in the order in which they were added to the batch. When all of the commands in a batch execute successfully, the method `executeBatch` returns an integer array containing one entry for each command in the batch.

The entries in the array are ordered according to the order in which the commands were processed (which, again, is the same as the order in which the commands were originally added to the batch). When all of the commands in a batch have been executed successfully, an entry in the array of update counts may have the following values :

■ 0 or greater — the command was processed successfully and the value is an update count indicating the number of rows in the database that were affected by the command's execution

- `Statement.SUCCESS_NO_INFO` — the command was processed successfully, but the number of rows affected is unknown

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

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

14.1.3 Handling Failures during Execution

A JDBC driver may or may not continue processing the remaining commands in a batch once execution of a command fails. However, a JDBC driver must always provide the same behavior with a particular data source. 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 the method `BatchUpdateException.getUpdateCounts` will always contain fewer entries than there were statements in the batch. Since statements 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 in the array returned by the method `BatchUpdateException.getUpdateCounts` always equals the number of commands in the batch. When a `BatchUpdateException` object is thrown and the driver continues processing after a failure, the array of update counts will contain the following `BatchUpdateException` constant:

- `Statement.EXECUTE_FAILED` — the command failed to execute successfully. This value is also returned for commands that could not be processed for some reason—such commands fail implicitly.

JDBC drivers that do not continue processing after a failure never return `Statement.EXECUTE_FAILED` 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 commands in the batch.

14.1.4 `PreparedStatement` Objects

When a `PreparedStatement` object is used, a command in a batch consists of a parameterized SQL statement and an associated set of parameters . The batch update facility is used with a `PreparedStatement` object to associate multiple sets of input parameter values with a single `PreparedStatement` object. The sets of parameter values together with their associated parameterized update commands can then be sent to the underlying data source engine for execution as a single unit.

CODE EXAMPLE 14-2 inserts two new employee records into a database as a single batch. The `PreparedStatement` interface setter methods are used to create each parameter set, one for each employee. The `PreparedStatement.addBatch` method adds a set of parameters to the current command.

```
// 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();
```

CODE EXAMPLE 14-2   Creating and executing a batch of prepared statements

Finally, the method `PreparedStatement.executeBatch` is called to submit the updates to the underlying data source. Calling this method clears the statement's associated list of commands. The array returned by `PreparedStatement.executeBatch` contains an element for each set of parameters in the batch, similar to the case for `Statement` objects. Each element contains either an update count or the generic 'success' indicator `SUCCESS_NO_INFO`.

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 with `Statement` objects, 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: 0 or higher, `Statement.SUCCESS_NO_INFO`, or `Statement.EXECUTE_FAILED`. The order of the entries in the array is the same order as the order in which commands were added to the batch.

14.1.5   `CallableStatement` Objects

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 `CallableStatement` object and sent to the underlying data source together.

Stored procedures invoked using the batch update facility with a callable statement must return a maximum of one update counts, if no update count is returned the array element value will be `Statement.SUCCESS_NO_INFO`. Additionally, a batchable stored procedure may not have OUT or INOUT parameters. The `CallableStatement.executeBatch` method throws an `SQLException`, not a sub-class of `SQLException`, if this restriction is violated. Error handling is analogous to that for `PreparedStatement` objects.

Advanced Data Types

Chapter 15 "Advanced Data Types" and Chapter 17 "Customized Type Mapping" discuss additions to the JDBC API that allow an application written in the Java programming language to access SQL:2003 data types, such as binary large objects and structured types. This chapter also describes the use of the `RowId` data type which is supported by many data sources even though it is not currently defined as an SQL:2003 data type. If a data source does not support an advanced data type described in these two chapters, a driver for that data source is not required to implement the methods and interfaces associated with that data type.

15.1   Taxonomy of SQL Types

The latest version of the ANSI/ISO SQL standard is commonly referred to as *SQL:2003*. The JDBC API incorporates a model of the new SQL:2003 data types that includes only those properties that are essential to exchanging data between a database and an application written in the Java programming language.

SQL:2003 specifies these data types:

- SQL92 built-in types—the familiar SQL 'column types'
  - CHAR
  - FLOAT
  - DATE
  - and so on
- SQL99 built-in types — new types added by SQL99
  - `BOOLEAN` — a truth value
  - `BLOB` — a Binary Large OBject
  - `CLOB` — a Character Large OBject

- New built-in types — new types added by SQL:2003
    - `XML` — a XML Object
- User Defined Types
    - Structured type — a user-defined type; for example:

        `CREATE TYPE PLANE_POINT AS (X FLOAT, Y FLOAT) NOT FINAL`
    - `DISTINCT` type — a user-defined type based on a built-in type; for example:

        `CREATE TYPE MONEY AS NUMERIC(10,2) FINAL`
- Constructed types — new types based on a given base type
    - `REF(structured-type)` — a pointer that persistently denotes an instance of a structured type that resides in the database
    - `base-type ARRAY[n]` — an array of n base-type elements
- Locators — new entities that are logical pointers to data that resides on the database server. A `LOCATOR` exists in the client environment and is a transient, logical pointer to data on the server. A locator typically refers to data that is too large to materialize on the client, such as images or audio. There are operators defined at the SQL level to retrieve random-access pieces of the data denoted by the locator.
    - `LOCATOR(structured-type)` — locator to a structured instance in server
    - `LOCATOR(array)` — locator to an array in server
    - `LOCATOR(blob)` — locator to a Binary Large Object in server
    - `LOCATOR(clob)` — locator to a Character Large Object in server
- Type for managing data external to the data source
    - `Datalink` — a reference to data external to the data source that is managed by the data source. `Datalink` values are part of SQL MED (Management of External Data), a part of the SQL ANSI/ISO standard specification. Having the data source manage the reference to external data has several advantages:

        i. Referential integrity — the referenced data can no longer be deleted or renamed directly through file system APIs

        ii. Access control — access to the data may be configured such that it is controlled by the data source instead of the file system

        iii. Coordinated backup and recovery — fields referenced by `Datalink` values may be included in the data source's backup process

        iv. Transaction consistency — changes that affect both relational and external data are executed in a transactional context to preserve the integrity and consistency of the data

The remainder of this chapter discusses the default mechanism provided by the JDBC API for accessing each of the SQL data types mentioned above. The JDBC API also provides a means of customizing the mapping of SQL DISTINCT and structured types into Java classes. This mechanism is discussed in Chapter 17 "Customized Type Mapping".

15.2 Mapping of Advanced Data Types

The JDBC API provides default mappings for advanced data types. Except for the DISTINCT and DATALINK types, these default mappings take the form of interfaces. The following list gives the data types and the interfaces to which they are mapped.

- BLOB — the Blob interface
- CLOB — the Clob interface
- NCLOB — the NClob interface
- ARRAY — the Array interface
- XML — the SQLXML interface
- Structured types — the Struct interface
- REF(structured type) — the Ref interface
- ROWID — the RowId interface

The other advanced data types with default mappings to the Java programming language are:

- DISTINCT — the type to which the base type is mapped. For example, a DISTINCT value based on an SQL NUMERIC type maps to a java.math.BigDecimal type because NUMERIC maps to BigDecimal in the Java programming language.
- DATALINK — a java.net.URL object.

15.3   `Blob` and `Clob` Objects

15.3.1   Creating `Blob`, `Clob` and `NClob` Objects

The `Connection` interface provides support for the creation of `Blob`, `Clob` and `NClob` objects using the methods `createBlob`, `createClob` and `createNClob`. The object that is created does not contain any data. Data may be added to the object by calling one of the `setXXX` methods on the `Blob`, `Clob` or `NClob` interface.

In CODE EXAMPLE 15-1, the method `Connection.createBlob` is used to create an empty `Blob` object. `Blob.setBytes` is used to write data to a `Blob` object.

```
Connection con = DriverManager.getConnection(url, props);

Blob aBlob = con.createBlob();

int numWritten = aBlob.setBytes(1, val);
```

CODE EXAMPLE 15-1   Creating and writing bytes to a `Blob` object

15.3.2   Retrieving `BLOB`, `Clob` and `NClob` Values in a ResultSet

The binary large object (`BLOB`) and character large object (`CLOB` and `NCLOB`) data types are treated similarly to the more primitive built-in types. Values of these types can be retrieved by calling the `getBlob`, `getClob` and `getNClob` methods in the `ResultSet` and `CallableStatement` interfaces.

For example, CODE EXAMPLE 15-2 retrieves a `BLOB` value from the first column of the `ResultSet` *rs* and a `CLOB` value from the second column.

```
Blob blob = rs.getBlob(1);

Clob clob = rs.getClob(2);
```

CODE EXAMPLE 15-2   Retrieving `BLOB` and `CLOB` values

The `Blob` interface contains operations for returning the length of the `BLOB` value, a specific range of bytes contained in the `BLOB` value, and so on. The `Clob` interface contains corresponding operations that are character based. The API documentation gives more details.

An application does not deal directly with the `LOCATOR(blob)`, `LOCATOR(clob)` and `LOCATOR(nclob)` types that are defined in SQL. By default, a JDBC driver should implement the `Blob`, `Clob` and `NClob` interfaces using the appropriate locator type.

`Blob`, `Clob` and `NClob` objects remain valid for at least the duration of the transaction in which they are created, unless their `free` method is invoked.

15.3.3 Accessing `Blob`, `Clob` and `NClob` Object Data

The `Blob`, `Clob` and `NClob` interfaces provide methods to access their internal content. It is also possible to access a subset of the data contained within a `Blob`, `Clob` and `NClob` object.

The example, CODE EXAMPLE 15-3, retrieves a 100 bytes of data from a `BLOB` and `CLOB` object starting at offset 250.

```
InputStream is = aBlob.getBinaryStream(250, 100);

BufferedReader br = aClob.getCharacterStream(250, 100);
```

CODE EXAMPLE 15-3 Retrieving a data subset from a `BLOB` and `CLOB` object

15.3.4 Storing `Blob`, `Clob` and `NClob` Objects

A `Blob`, `Clob` or `NClob` object can be passed as an input parameter to a `PreparedStatement` object just like other data types. The method `setBlob` sets a `PreparedStatement` parameter with a `Blob` object, the method `setClob` sets a `Clob` object and the method `setNClob` sets a `NClob` object as a parameter. In CODE EXAMPLE 15-4, *authorImage* is an instance of `java.sql.Blob` retrieved from another SQL statement, and *authorBio* is a an instance of `java.sql.Clob` retrieved from another SQL statement.

```
PreparedStatement pstmt = conn.prepareStatement(
                   "INSERT INTO bio (image, text) VALUES (?, ?)");
```

```
pstmt.setBlob(1, authorImage);

pstmt.setClob(2, authorBio);
```

CODE EXAMPLE 15-4 Setting `Blob` and `Clob` objects as parameters to a
PreparedStatement object

The `setBinaryStream` and `setObject` methods may also be used to set a `Blob`
object as a parameter in a `PreparedStatement` object. The `setAsciiStream`,
`setCharacterStream`, and `setObject` methods are alternate means of setting a
`Clob` object as a parameter.The methods `setNCharacterStream`, and `setObject`
provide an alternate means of setting a `NClob` object as a parameter.

The `updateBlob`, `updateClob` and `updateNClob` methods can be used to update
a column value in an updatable result set.

15.3.5 Altering `Blob`, `Clob` and `NClob` Objects

The `Blob`, `Clob` and `NClob` interfaces provide methods to alter their internal
content. In CODE EXAMPLE 15-5, the method `setBytes` is used to write the first five
bytes of the `Blob` object retrieved from the column DATA.

```
byte[] val = {0,1,2,3,4};

...

Blob data = rs.getBlob("DATA");

int numWritten = data.setBytes(1, val);
```

CODE EXAMPLE 15-5 Writing bytes to a `Blob` object

Similarly, the `Clob` or `NClob` methods `setString` and `truncate` can be used to
change the value of a column containing a `Clob` or a `NClob` object.

15.3.6 Releasing `Blob`, `Clob` and `NClob` Resources

`Blob`, `Clob` and `NClob` objects remain valid for at least the duration of the
transaction in which they are created. This could potentially result in an application
running out of resources during a long running transaction. Applications may
release `Blob`, `Clob` and `NClob` resources by invoking their `free` method.

In CODE EXAMPLE 15-6, the method `Clob.free` is used to release the resources held
for a previously created `Clob` object.

```
Clob aClob = con.createClob();

int numWritten = aClob.setString(1, val);
```

```
aClob.free();
```

CODE EXAMPLE 15-6   releasing resources for a `Clob` object

15.4   SQLXML Objects

15.4.1   Creating SQLXML Objects

The `Connection` interface provides support for the creation of SQLXML objects using the method `createSQLXML`. The object that is created does not contain any data. Data may be added to the object by calling the `setString` or `createXMLStreamWriter` method on the SQLXML interface.

In CODE EXAMPLE 15-7, the method `Connection.createSQLXML` is used to create an empty SQLXML object. `SQLXML.setString` is used to write data to the SQLXML object that was created.

```
Connection con = DriverManager.getConnection(url, props);
SQLXML xmlVar = con.createSQLXML();
int numWritten = xmlVar.setString(val);
```

CODE EXAMPLE 15-7   Creating and writing bytes to a SQLXML object

15.4.2   Retrieving SQLXML values in a ResultSet

The SQLXML data type is treated similarly to the more primitive built-in types. A SQLXML value can be retrieved by calling the `getSQLXML` method in the `ResultSet` or `CallableStatement` interface.

For example, CODE EXAMPLE 15-8 retrieves a SQLXML value from the first column of the `ResultSet` *rs*.

```
SQLXML xmlVar = rs.getSQLXML(1);
```

CODE EXAMPLE 15-8   Retrieving a SQLXML value

SQLXML objects remain valid for at least the duration of the transaction in which
they are created, unless their free method is invoked.

15.4.3   Accessing SQLXML Object Data

The SQLXML interface provide the getString and createXMLStreamReader
methods to access its internal content. The example, CODE EXAMPLE 15-9, retrieves the
contents of an SQLXML object using the getString method.

```
SQLXML xmlVal= rs.getSQLXML(1);

int xmlLen = xmlVal.getString();
```

CODE EXAMPLE 15-9   Retrieving a data from a SQLXML object using getString

The createXMLStreamReader method returns a
javax.xml.stream.XMLStreamReader object allowing for the XML to be
iterated and parsed using the next and hasNext methods of XMLStreamReader.

The example, CODE EXAMPLE 15-10, retrieves data from an SQLXML object using the
XMLStreamReader object returned by the invocation of the
createXMLStreamReader method:

```
SQLXML xmlVal= rs.getSQLXML(1);

XMLStreamReader aReader = xmlVal.createXMLStreamReader();

while(aReader.hasNext()) {

  // parse the next event

  for (int event = reader.next();

                event != XMLStreamConstants.END_DOCUMENT;

                event = reader.next()) {

            switch (event) {
            case XMLStreamConstants.START_ELEMENT:
                // process the element

                …
            break;
```

```
        }

    }
```
CODE EXAMPLE 15-10 Using `createStreamReader` to access SQLXML data

15.4.4 Storing SQLXML Objects

A SQLXML object can be passed as an input parameter to a PreparedStatement
object just like other data types. The method setSQLXML sets the designated
PreparedStatement parameter with a SQLXML object.

In CODE EXAMPLE 15-11, *authorData* is an instance of java.sql.SQLXML whose
data was initialized using java.xml.stream.StreamWriter.

```
PreparedStatement pstmt = conn.prepareStatement(
                "INSERT INTO bio (xmlData, authId) VALUES (?, ?)");
pstmt.setSQLXML(1, authorData);
pstmt.setInt(2, authorId);
```
CODE EXAMPLE 15-11 Setting an SQLXML object as a parameter to a PreparedStatement
object

The updateSQLXML method can be used to update a column value in an updatable
result set.

If the java.xml.stream.XMLStreamWriter for the SQLXML object has not been
closed prior to calling setSQLXML or updateSQLXML, a SQLException will be
thrown.

15.4.5 Initializing SQLXML Objects

The SQLXML interface provides the methods setString and
createXMLStreamWriter to initialize the content for an SQLXML object that has
been created by calling the Connection.createSQLXML method. The
createXMLStreamWriter method returns an instance of
java.xml.stream.XMLStreamWriter which is used to populate the contents of
the SQLXML object. In CODE EXAMPLE 15-12, the method createXMLStreamWriter
is used to initialize the newly created SQLXML object.

```
Connection con = DriverManager.getConnection(url, props);
SQLXML xmlVar = con.createSQLXML();
```

```
XMLStreamWriter aWriter = xmlVar.createXMLStreamWriter();
// write to the stream using XMLStreamWriter methods
aWriter.close();
```

CODE EXAMPLE 15-12 Populating a SQLXML object

Similarly, the SQLXML setString method can be used to initialize an SQLXML
object.

If an attempt is made to call the setString and createXMLStreamWriter
methods on an SQLXML object that has previously been initialized, a SQLException
will be thrown. If more than one call to createXMLStreamWriter occurs for the
same SQLXML object, a SQLException is thrown and the previously returned
javax.xml.stream.XMLStreamWriter object is not effected.

15.4.6 Releasing SQLXML Resources

SQLXML objects remain valid for at least the duration of the transaction in which
they are created. This could potentially result in an application running out of
resources during a long running transaction. Applications may release SQLXML
resources by invoking their free method.

In CODE EXAMPLE 15-13, the method SQLXML.free is used to release the resources
held for a previously created SQLXML object.

```
SQLXML xmlVar = con.createSQLXML();
int numWritten = xmlVar.setString( val);
xmlVar.free();
```

CODE EXAMPLE 15-13 releasing resources for a SQLXML object

15.5   Array Objects

15.5.1   Retrieving `Array` Objects

Data of type SQL `ARRAY` can be retrieved by calling the `getArray` method of the
`ResultSet` and `CallableStatement` interfaces. For example, the following line
of code retrieves an `Array` value from the first column of the `ResultSet` *rs*.

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

By default, a JDBC driver should implement the `Array` interface using an SQL
`LOCATOR(array)` internally. Also by default, `Array` objects remain valid only
during the transaction in which they are created.

The `Array` object returned to an application by the `ResultSet.getArray` and
`CallableStatement.getArray` methods is a logical pointer to the SQL `ARRAY`
value in the database; it does not contain the contents of the SQL `ARRAY` value. The
`Array` interface provides several versions of the methods `getArray` and
`getResultSet` that return the contents of an SQL `ARRAY` value to the client as a
materialized Java programming language array (Java array) or as a `ResultSet`
object. The API documentation gives complete details.

15.5.2   Storing `Array` Objects

The `PreparedStatement` methods `setArray` and `setObject` may be called to
pass an `Array` value as an input parameter to a `PreparedStatement` object.
CODE EXAMPLE 15-14 sets the `Array` object *member_array*, which was retrieved
from another table in the database, as the second parameter to the
`PreparedStatement` *pstmt*.

```
PreparedStatement pstmt = conn.prepareStatement(
                "INSERT INTO dept (name, members) VALUES (?, ?)");
pstmt.setString(1, "biology");
pstmt.setArray(2, member_array);
pstmt.executeUpdate();
```

CODE EXAMPLE 15-14 Storing an Array object

A Java array may be passed as an input parameter by calling the method
`PreparedSatement.setObject`.

15.5.3 Updating `Array` Objects

The `ResultSet` methods `updateArray` and `updateObject` can be used to update a column value.

CODE EXAMPLE 15-15 uses the method `ResultSet.updateArray` to update the value of the column LATEST_NUMBERS in one `ResultSet` object with an `Array` object retrieved from the column NUMBERS in another `ResultSet` object.

```
// retrieve a column containing an SQL ARRAY value from ResultSet rs

java.sql.Array num = rs.getArray("NUMBERS");

...
// update the column "LATEST_NUMBERS" in a second ResultSet

// with the value retrieved...

rs2.updateArray("LATEST_NUMBERS", num);

rs2.updateRow();
```

CODE EXAMPLE 15-15 Updating a column with an Array object

15.6 `Ref` Objects

15.6.1 Retrieving `REF` Values

An SQL `REF(structured type)` value can be retrieved as a `Ref` object by calling the `getRef` method of the `ResultSet` and `CallableStatement` interfaces. For example, in CODE EXAMPLE 15-16, the `ResultSet` *rs* contains a reference to an instance of the SQL structured type `dog` that is stored in the table `dogs`. The code retrieves this `REF(dog)` from the first column of *rs*.

```
ResultSet rs = stmt.executeQuery("SELECT OID FROM DOGS " +

                                    "WHERE NAME = 'ROVER'");

rs.next();

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

CODE EXAMPLE 15-16 Retrieving a `REF` value

An SQL REF value is a pointer; therefore, a Ref object, which is the mapping of a REF value, is likewise a pointer and does not contain the data of the structured type instance to which it refers. A Ref object remains valid while the session or connection on which it is created is open.

15.6.2 Retrieving the Referenced Value

The Ref object returned from the method getRef is a reference to an instance of a structured type in the underlying data source. The methods getObject() and getObject(Map map) can be used to retrieve the structured type instance that is referenced. CODE EXAMPLE 15-17 shows how a reference to an instance of the structured type Address can be dereferenced to retrieve the instance of Address. This example would require that a map, mapping Address to its SQL type, had been supplied to the Connection using the method setMap.

```
Ref ref = rs.getRef(1);
Address addr = (Address)ref.getObject();
```

CODE EXAMPLE 15-17 Retrieving the structured type instance referenced by a Ref object

15.6.3 Storing Ref Objects

The PreparedStatement.setRef method may be called to pass a Ref object as an input parameter to a PreparedStatement object.

15.6.4 Storing the Referenced Value

An instance of a structured type retrieved with the method ResultSet.getRef or CallableStatement.getRef is stored using the Ref.setObject method. In CODE EXAMPLE 15-18, the table DOGS stores instances of the structured type DOG. The SELECT statement selects the REF(DOG) that refers to the instance in which the name is Rover. The referenced instance of the type DOG is retrieved using getValue. The parameter map describes a mapping from the SQL type DOG to the Java class Dog, which implements the SQLData interface.

```
ResultSet rs = stmt.executeQuery("SELECT OID FROM DOGS " +
                                       "WHERE NAME = 'ROVER'");
rs.next();
Ref rover = rs.getRef("OID");
Dog dog = (Dog)rover.getObject(map);
```

```
// manipulate instance of Dog
dog.setAge(14);
...
// store updated Dog
rover.setObject((Object)dog);
```

CODE EXAMPLE 15-18 Retrieving and storing the structured type instance referenced by a
`Ref` object

15.6.5 Metadata

The type `REF` is defined in the class `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.

15.7 Distinct Types

An SQL `DISTINCT` type is a new user defined data type that is based on one of the
primitive types. C and C++ programmers can think of it as being similar to a
`typedef`.

15.7.1 Retrieving Distinct Types

By default, a column of SQL type `DISTINCT` is retrieved by calling any getter
method that is appropriate to the type on which it is based. For example, the
following type declaration creates the type `MONEY`, which is based on the SQL type
`NUMERIC`.

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

CODE EXAMPLE 15-19 Creating a distinct type

CODE EXAMPLE 15-20 uses the method `getBigDecimal` to retrieve a `MONEY` value
because the underlying SQL `NUMERIC` type is mapped to the
`java.math.BigDecimal` type.

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

CODE EXAMPLE 15-20 Retrieving a distinct type

15.7.2   Storing Distinct Types

Any setter method in the `PreparedStatement` interface that is appropriate for the base 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` in CODE EXAMPLE 15-19, the method `PreparedStatement.setBigDecimal` would be used.

15.7.3   Metadata

The type code `DISTINCT` is defined in the class `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 can be 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, CODE EXAMPLE 15-21 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, the `getUDTs` method returns an empty result set.

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

CODE EXAMPLE 15-21 Querying a `DatabaseMetaData` object for distinct types

Each row in the `ResultSet` object returned by the method `getUDTs` describes a UDT. Each row contains 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 SQL type name
CLASS_NAME	String => a Java class name
DATA_TYPE	short => value defined in `java.sql.Types`, such as DISTINCT
REMARKS	String => explanatory comment on the type
BASE_TYPE	short => value defined in `java.sql.Types`, for DISTINCT or reference types (may be null)

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 CLASS_NAME 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, CLASS_NAME 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 CLASS_NAME. The CLASS_NAME value reflects a custom type mapping when one is used. See Chapter 17 "Customized Type Mapping" for details.

15.8 Structured Types

15.8.1 Retrieving Structured Types

An SQL structured type value is always retrieved by calling the method getObject. By default, getObject returns a value of type Struct for a structured type. For example, the following line of code retrieves a Struct value from the first column of the current row of the ResultSet object rs.

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

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 materializes 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.

15.8.2 Storing Structured Types

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

15.8.3 Metadata

The type code STRUCT is defined in the class `java.sql.Types`. This value is returned by methods such as `DatabaseMetaData.getTypeInfo` and `DatabaseMetaData.getColumns` when a JDBC driver supports structured data types.

An SQL structured 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, CODE EXAMPLE 15-22 returns descriptions of all the SQL structured types defined in the `catalog-name.schema-name` schema.

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

CODE EXAMPLE 15-22 Querying a `DatabaseMetaData` object for structured types

If the driver does not support UDTs or no matching UDTs are found, an empty result set is returned. See section 15.7.3 for a description of the result set returned by the method `getUDTs`.

When the DATA_TYPE returned by `getUDTs` is `Types.STRUCT`, the CLASS_NAME 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, CLASS_NAME defaults to `java.sql.Struct` for structured types. If there is a custom mapping for the STRUCT type, CLASS_NAME will be the implementation of the interface SQLData that specifies the mapping. The JDBC API does not prohibit a driver from returning a subtype of the class named by CLASS_NAME. Chapter 17 "Customized Type Mapping" provides more information about implementations of the SQLData interface.

15.9 Datalinks

A DATALINK value references a file outside of the underlying data source that the data source manages.

15.9.1 Retrieving References to External Data

A reference to external data being managed by the data source can be retrieved using the method `ResultSet.getURL`. The `java.net.URL` object that is returned can be used to manipulate the data.

```
java.net.URL url = rs.getURL(1);
```

CODE EXAMPLE 15-23 Retrieving a reference to an external data object

In cases where the type of URL returned by the methods `getObject` or `getURL` is not supported by the Java platform, the URL can be retrieved as a `String` by calling the method `getString`.

15.9.2 Storing References to External Data

The method `PreparedStatement.setURL` can be used to pass a `java.net.URL` object to a prepared statement. In cases where the type of URL being set is not supported by the Java platform, the URL can be stored using the `setString` method.

15.9.3 Metadata

The type code `DATALINK` is defined in the class `java.sql.Types`. This value is returned by methods such as `DatabaseMetaData.getTypeInfo` and `DatabaseMetaData.getColumns` when a JDBC driver supports the `Datalink` data type for references to external files.

15.10 RowId Objects

15.10.1 Lifetime of RowId Validity

A `RowId` may be thought of as the address for a given row. The address may be logical or physical, however this is determined by the originating data source. A `RowId` object may be valid so long as the identified row is not deleted and the lifetime of the `RowId` is within the bounds of the lifetime specified by that `RowId`'s data source.

RowId object lifetime validity may be determined by calling the
`DatabaseMetaData.getRowIdLifetime()` method, which returns a value of the
`RowIdLifetime` enumerated data-type.

TABLE 15-1 Standard RowId life-time validities

RowIdLifeTime Enumerated Datatype	Specification
ROWID_UNSUPPORTED	Indicates that this data source does not support the ROWID type.
ROWID_VALID_OTHER	Indicates that the lifetime of a RowId from this data source is implementation dependant.
ROWID_VALID_TRANSACTION	Indicates that the lifetime of a RowId from this data source is at least the containing transaction as long as the identified row is not deleted.
ROWID_VALID_SESSION	Indicates that the lifetime of a RowId from this data source is at least the containing session as long as the identified row is not deleted.
ROWID_VALID_FOREVER	Indicates that the lifetime of a RowId from this data source is, effectively, unlimited as long as the identified row is not deleted.

15.10.2 Retrieving RowId Values

A `RowId` may be retrieved using the getter methods defined in `ResultSet` and
`CallableStatement`. The returned `java.sql.RowId` object that is returned is an
immutable object that can be used for subsequent referrals as a unique identifier to a
row.

```
java.sql.RowId rowId_1 = rs.getRowId(1);
```

CODE EXAMPLE 15-24 Retrieving a RowId for a ResultSet object, rs

15.10.3 Using RowId Values

A `RowId` value may be used in a parameterized `PreparedStatement` to set a
parameter with a `RowId` object.

```
Connection conn = ds.getConnection(user, passwd);
PreparedStatement ps = conn.prepareStatement("INSERT INTO BOOKLIST" +
```

```
                              "(ID, AUTHOR, TITLE, ISBN) VALUES (?, ?, ?,
?)");
```

```
ps.setRowId(1, rowId_1);
```

CODE EXAMPLE 15-25 Setting a PreparedStatement RowId parameter

A `RowId` value may also be used in an updatable ResultSet to update a column with a specific `RowId` value.

```
ResultSet rs = ...
```

```
rs.next();
```

```
rs.updateRowId(1, rowId_1);
```

CODE EXAMPLE 15-26 Setting a RowId value in the current row of a ResultSet

A `RowId` object value is typically not portable between data sources and should be considered as specific to the data source when using the set or update method in `PreparedStatement` and `ResultSet` objects respectively. It is therefore inadvisable to get a `RowId` from a `ResultSet` object with a connection 'Foo' and then attempt to use the `RowId` in a unrelated `ResultSet` object with a connection 'Bar'.

Result Sets

The `ResultSet` interface provides methods for retrieving and manipulating the results of executed queries.

16.1 Kinds of `ResultSet` Objects

`ResultSet` objects can have different functionality and characteristics. These characteristics are result set type, result set concurrency, and cursor holdability.

16.1.1 `ResultSet` Types

The type of a `ResultSet` object determines the level of its functionality in two main areas: (1) the ways in which the cursor can be manipulated and (2) how concurrent changes made to the underlying data source are reflected by the `ResultSet` object. The latter is called the *sensitivity* of the `ResultSet` object.

The three different `ResultSet` types are described below.

1. `TYPE_FORWARD_ONLY`
   - The result set is not scrollable; its cursor moves forward only, from before the first row to after the last row.
   - The rows contained in the result set depend on how the underlying database materializes the results. That is, it contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

2. `TYPE_SCROLL_INSENSITIVE`
   - The result set is scrollable; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position.

- The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.

3. TYPE_SCROLL_SENSITIVE

- The result set is scrollable; its cursor can move both forward and backward relative to the current position, and it can move to an absolute position.

- The result set reflects changes made to the underlying data source while the result set remains open.

The default ResultSet type is TYPE_FORWARD_ONLY.

The method DatabaseMetaData.supportsResultSetType returns true if the specified type is supported by the driver and false otherwise.

If the driver does not support the type supplied to the methods createStatement, prepareStatement, or prepareCall, it generates an SQLWarning on the Connection object that is creating the statement. When the statement is executed, the driver returns a ResultSet object of a type that most closely matches the requested type. An application can find out the type of a ResultSet object by calling the method ResultSet.getType.

16.1.2   ResultSet Concurrency

The concurrency of a ResultSet object determines what level of update functionality is supported.

The two concurrency levels are:

- CONCUR_READ_ONLY

    The ResultSet object cannot be updated using the ResultSet interface.

- CONCUR_UPDATABLE

    The ResultSet object can be updated using the ResultSet interface.

The default ResultSet concurrency is CONCUR_READ_ONLY.

The method DatabaseMetaData.supportsResultSetConcurrency returns true if the specified concurrency level is supported by the driver and false otherwise.

If the driver does not support the concurrency level supplied to the methods createStatement, prepareStatement, or prepareCall, it generates an SQLWarning on the Connection object that is creating the statement. An application can find out the concurrency of a ResultSet object by calling the method ResultSet.getConcurrency.

If the driver cannot return a `ResultSet` object at the requested type and concurrency, it determines the appropriate type before determining the concurrency.

16.1.3 `ResultSet` Holdability

Calling the method `Connection.commit` can close the `ResultSet` objects that have been created during the current transaction. In some cases, however, this may not be the desired behaviour. The `ResultSet` property `holdability` gives the application control over whether `ResultSet` objects (cursors) are closed when `commit` is called.

The following `ResultSet` constants may be supplied to the `Connection` methods `createStatement`, `prepareStatement`, and `prepareCall`:

1. HOLD_CURSORS_OVER_COMMIT

- `ResultSet` objects (cursors) are not closed; they are held open when the method `commit` is called.

2. CLOSE_CURSORS_AT_COMMIT

- `ResultSet` objects (cursors) are closed when `commit` is called. Closing cursors at commit can result in better performance for some applications.

16.1.3.1 Determining `ResultSet` Holdability

The default holdability of `ResultSet` objects is implementation defined. The `DatabaseMetaData` method `getResultSetHoldability` can be called to determine the default holdability of result sets returned by the underlying data source.

If the driver does not support the holdability level supplied to the methods `createStatement`, `prepareStatement`, or `prepareCall`, it generates an `SQLWarning` on the `Connection` object that is creating the statement. In the case of a SQL CALL statement, the holdability of any returned `ResultSets` are determined by the stored procedure, independent of the holdability level specified to the SQL CALL statement.

It is the application's responsibility, not the JDBC driver implementation, to validate the holdability of the returned `ResultSet`. An application can find out the holdability of a `ResultSet` object by calling the method `ResultSet.getHoldability`.

16.1.4 Specifying `ResultSet` Type, Concurrency and Holdability

The parameters supplied to the methods `Connection.createStatement`, `Connection.prepareStatement`, and `Connection.prepareCall` determine the type, concurrency, and holdability of `ResultSet` objects that the statement produces. CODE EXAMPLE 16-1 creates a `Statement` object that will return scrollable, read-only `ResultSet` objects that are insensitive to updates made to the data source and that will be closed when the transaction in which they were created is committed.

```
Connection conn = ds.getConnection(user, passwd);

Statement stmt = conn.createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY,
                    ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

CODE EXAMPLE 16-1 Creating a scrollable, insensitive, read-only result set with a cursor that is not holdable

The `Statement`, `PreparedStatement` and `CallableStatement` interfaces also provide setter and getter methods for each of these properties.

16.2 Creating and Manipulating `ResultSet` Objects

16.2.1 Creating `ResultSet` Objects

A `ResultSet` object is most often created as the result of executing a `Statement` object. The `Statement` methods `executeQuery` and `getResultSet` both return a `ResultSet` object, as do various `DatabaseMetaData` methods. CODE EXAMPLE 16-2 executes an SQL statement returning a `ResultSet` object.

```
Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("select author, title, isbn " +
                                "from booklist");
```

CODE EXAMPLE 16-2 Executing a query returning a `ResultSet` object

For each book in the table `booklist`, the `ResultSet` object will contain a row consisting of three columns, `author`, `title`, and `isbn`. The following sections detail how these rows and columns can be retrieved.

16.2.2 Cursor Movement

A `ResultSet` object maintains a cursor, which points to its current row of data. When a `ResultSet` object is first created, the cursor is positioned before the first row. The following methods can be used to move the cursor:

■ `next()` — moves the cursor forward one row. Returns `true` if the cursor is now positioned on a row and `false` if the cursor is positioned after the last row.

■ `previous()` — moves the cursor backwards one row. Returns `true` if the cursor is now positioned on a row and `false` if the cursor is positioned before the first row.

■ `first()` — moves the cursor to the first row in the `ResultSet` object. Returns `true` if the cursor is now positioned on the first row and `false` if the `ResultSet` object does not contain any rows.

■ `last()` — moves the cursor to the last row in the `ResultSet` object. Returns `true` if the cursor is now positioned on the last row and `false` if the `ResultSet` object does not contain any rows.

■ `beforeFirst()` — positions the cursor at the start of the `ResultSet` object, before the first row. If the `ResultSet` object does not contain any rows, this method has no effect.

■ `afterLast()` — positions the cursor at the end of the `ResultSet` object, after the last row. If the `ResultSet` object does not contain any rows, this method has no effect.

■ `relative(int rows)` — moves the cursor relative to its current position.

If `rows` is 0 (zero), the cursor is unchanged. If `rows` is positive, the cursor is moved forward `rows` rows. If the cursor is less than the specified number of rows from the last row, the cursor is positioned after the last row. If `rows` is negative, the cursor is moved backward `rows` rows. If the cursor is less than `rows` rows from the first row, the cursor is positioned before the first row.

The method `relative` returns `true` if the cursor is positioned on a valid row and `false` otherwise.

If `rows` is 1, `relative` is identical to the method `next`. If `rows` is -1, `relative` is identical to the method `previous`.

■ `absolute(int row)` — positions the cursor on the `row`-th row of the `ResultSet` object.

If `row` is positive, the cursor is moved `row` rows from the beginning of the `ResultSet` object. The first row is 1, the second 2, and so on. If `row` is greater than the number of rows in the `ResultSet` object, the cursor is positioned after the last row.

If `row` is negative, the cursor is moved `row` rows from the end of the `ResultSet` object. The last row is -1, the penultimate -2, and so on. If `row` is greater than the number of rows in the `ResultSet` object, the cursor is positioned before the first row.

Calling `absolute(0)` moves the cursor before the first row.

For a `ResultSet` object that is of type `TYPE_FORWARD_ONLY`, the only valid cursor movement method is `next`. All other cursor movement methods throw an `SQLException`.

16.2.3   Retrieving Values

The `ResultSet` interface provides methods for retrieving the values of columns from the row where the cursor is currently positioned.

Two getter methods exist for each JDBC type: one that takes the column index as its first parameter and one that takes the column name or label.

The columns are numbered from left to right, as they appear in the select list of the query, starting at 1.

Column names supplied to getter methods are case insensitive. If a select list contains the same column more than once, the first instance of the column will be returned.

The index of the first instance of a column name can be retrieved using the method `findColumn`. If the specified column is not found, the method `findColumn` throws an `SQLException`.

```
ResultSet rs = stmt.executeQuery(sqlstring);
int colIdx = rs.findColumn("ISBN");
```

CODE EXAMPLE 16-3   Mapping a column name to a column index

16.2.3.1 Data Type Conversions

The recommended `ResultSet` getter method for each JDBC type is shown in TABLE B-6 on page B-216. This table also shows all of the possible conversions that a JDBC driver may support. The method `DataBaseMetaData.supportsConvert(int fromType, int toType)` returns `true` if the driver supports the given conversion.

16.2.3.2 `ResultSet` Metadata

When the `ResultSet` method `getMetaData` is called on a `ResultSet` object, it returns a `ResultSetMetaData` object describing the columns of that `ResultSet` object. In cases where the SQL statement being executed is unknown until runtime, the result set metadata can be used to determine which of the getter methods should be used to retrieve the data. In CODE EXAMPLE 16-4, result set metadata is used to determine the type of each column in the result set.

```
ResultSet rs = stmt.executeQuery(sqlString);
ResultSetMetaData rsmd = rs.getMetaData();
int colType [] = new int[rsmd.getColumnCount()];
for (int idx = 0, int col = 1; idx < colType.length; idx++, col++)
                colType[idx] = rsmd.getColumnType(col);
```
CODE EXAMPLE 16-4 Retrieving result set metadata

16.2.3.3 Retrieving `NULL` values

The method `wasNull` can be called to determine if the last value retrieved was a JDBC `NULL` in the database.

When the column value in the database is JDBC `NULL`, it may be returned to the Java application as `null`, `0`, or `false`, depending on the type of the column value. Column values that map to Java `Object` types are returned as a Java `null`; those that map to numeric types are returned as `0`; those that map to a Java `boolean` are returned as `false`. Therefore, it may be necessary to call the `wasNull` method to determine whether the last value retrieved was a JDBC `NULL`.

16.2.4 Modifying `ResultSet` Objects

`ResultSet` objects with concurrency `CONCUR_UPDATABLE` can be updated using `ResultSet` methods. Columns can be updated, new rows can be inserted, and rows can be deleted using methods defined in the `ResultSet` interface.

16.2.4.1 Updating a Row

Updating a row in a `ResultSet` object is a two-phase process. First, the new value for each column being updated is set, and then the change is applied to the row. The row in the underlying data source is not updated until the second phase is completed.

The `ResultSet` interface contains two update methods for each JDBC type, one specifying the column to be updated as an index and one specifying the column name as it appears in the select list.

Column names supplied to updater methods are case insensitive. If a select list contains the same column more than once, the first instance of the column will be updated.

If an update method is called on a `ResultSet` whose concurrency level is `ResultSet.CONCUR_READ_ONLY`, then a `SQLException` must be thrown.

The method `updateRow` is used to apply all column changes to the current row, and to clear the record of changes made by prior update methods. The changes are not made to the row until `updateRow` has been called. If updateRow is called on a `ResultSet` whose concurrency level is `ResultSet.CONCUR_READ_ONLY`, then a `SQLException` must be thrown. If the concurrency level is `ResultSet.CONCUR_UPDATABLE` and updateRow is called without changes being made to the row, the call will be considered a no-op.

The method `cancelUpdates` can be used to back out changes made to the row before the method `updateRow` is called. CODE EXAMPLE 16-5 shows the current row being updated to change the value of the column "author" to "Zamyatin, Evgenii Ivanovich":

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                    ResultSet.CONCUR_UPDATABLE);

ResultSet rs = stmt.executeQuery("select author from booklist " +
                                "where isbn = 140185852");

rs.next();

rs.updateString("author", "Zamyatin, Evgenii Ivanovich");

rs.updateRow();
```

CODE EXAMPLE 16-5   Updating a row in a `ResultSet` object

The method `DatabaseMetaData.ownUpdatesAreVisible(int type)` returns `true` if a `ResultSet` object of the specified type is able to see its own updates and `false` otherwise.

A `ResultSet` object may be able to use the method `rowUpdated` to detect rows that have had the method `updateRow` called on them. The method `DatabaseMetaData.updatesAreDetected(int type)` returns `true` if a `ResultSet` object of the specified type can determine if a row is updated using the method `rowUpdated` and `false` otherwise.

16.2.4.2   Deleting a Row

A row in a `ResultSet` object can be deleted using the method `deleteRow`. CODE EXAMPLE 16-6 shows the fourth row of the `ResultSet` *rs* being deleted.

```
rs.absolute(4);

rs.deleteRow();
```

CODE EXAMPLE 16-6   Deleting a row in a `ResultSet` object

After the method `deleteRow` has been called, the current row is deleted in the underlying data source. This deletion is visible as a change in the open `ResultSet` object if the row is either removed or replaced by an empty or invalid row.

If the deleted row is removed or replaced by an empty row, the method `DatabaseMetaData.ownDeletesAreVisible(int type)` will return `true`. It returns `false` if the `ResultSet` object still contains the deleted row, which means that the deletion is not visible as a change to `ResultSet` objects of the given type.

The method `DatabaseMetaData.othersDeletesAreVisible(int type)` checks whether deletions made by others (another transaction or another `ResultSet` object in the same transaction) are visible to `ResultSet` objects of the specified type. This method returns `true` if a row deleted by others is visible and `false` if it is not.

If a `ResultSet` object can detect deletions, the `ResultSet` method `rowDeleted` returns `true` when the current row has been deleted and `false` when it has not. However, `rowDeleted` also returns `false` if the `ResultSet` object cannot detect deletions. The method `DatabaseMetaData.deletesAreDetected(int type)` can be called to see whether a `ResultSet` object of the specified type can call the method `rowDeleted` to detect a deletion that is visible. The method `deletesAreDetected` returns `false` if a row deleted from the `ResultSet` object is removed from it and `true` if the deleted row is replaced by an empty or invalid row.

In CODE EXAMPLE 16-7, application code uses metadata to process a `ResultSet` object that may contain deleted rows.

```
if (dbmd.ownDeletesAreVisible(ResultSet.TYPE_SCROLL_INSENSITIVE) &&
    dbmd.deletesAreDetected(ResultSet.TYPE_SCROLL_INSENSITIVE)) {
    while (rs.next) {
```

```
                    if (rs.rowDeleted()) {
                        continue;
                    } else {
                        // process row
                        ...
                    }
        }
} else {
        // if up-to-date data is needed, it is better to close this
        // ResultSet object and reexecute the query to get an updated
        // ResultSet object
        ...
        rs.close();
        break;
}
```

CODE EXAMPLE 16-7   Processing a `ResultSet` object containing deleted rows

---

Note – CODE EXAMPLE 16-7 does not cover the case where `ownDeletesAreVisible` returns `true` and `deletesAreDetected` returns `false`. This will cause an `SQLException` to be thrown when the cursor is positioned on a deleted row, so an implementation with these characteristics requires that an application handle the exception. Such an implementation does not appear to be a very likely.

After the method `deleteRow` has been called, the cursor will be positioned before the next valid row. If the deleted row is the last row, the cursor will be positioned after the last row.

16.2.4.3   Inserting a Row

New rows may be inserted using the `ResultSet` interface. New rows are constructed in a special *insert row*. The steps to insert a new row are:

1. Move the cursor to the insert row

2. Set the values for the columns of the row using the `ResultSet` interface update methods

3. Insert the new row into the `ResultSet` object

CODE EXAMPLE 16-8 shows the steps necessary to insert a new row into the table `booklist`.

```
// select all the columns from the table booklist
ResultSet rs = stmt.executeQuery("select author, title, isbn " +
                                 "from booklist");
rs.moveToInsertRow();
// set values for each column
rs.updateString(1, "Huxley, Aldous");
rs.updateString(2, "Doors of Perception and Heaven and Hell");
rs.updateLong(3, 60900075);
// insert the row
rs.insertRow();
// move the cursor back to its position in the result set
rs.moveToCurrentRow();
```

CODE EXAMPLE 16-8 Inserting a new row into a `ResultSet` object

Each column in the insert row that does not allow `null` as a value and does not have a default value must be given a value using the appropriate update method. If this is not the case, the method `insertRow` will throw an `SQLException`.

The method `DatabaseMetaData.ownInsertsAreVisible(int type)` will return `true` if newly inserted rows can be seen in result sets of the specified type.

If the `ResultSet` objects of the specified type can identify newly inserted rows, the method `DatabaseMetaData.insertsAreDetected(int type)` will return `true`. This indicates that the inserted rows are visible to the `ResultSet` object.

16.2.4.4 Positioned Updates and Deletes

JDBC drivers or DBMSs that do not support performing updates via the `ResultSet` interface may support positioned updates and deletes via SQL commands. This method of updating a row relies on using named cursors to allow multiple statements to act on a single result set. CODE EXAMPLE 16-9 shows the use of the method `setCursorName` to associate a cursor with a `Statement` object and then the use of the method `getCursorName` to retrieve the name for use by a second `Statement` object.

```
Statement stmt1 = conn.createStatement();
stmt1.setCursorName("CURSOR1");
ResultSet rs = stmt1.executeQuery("select author, title, isbn " +
                "from booklist for update of author");
// move to the row we want to update
while ( ... ) {
```

```
                    rs.next()
}
String cursorName = rs.getCursorName();
Statement stmt2 = conn.createStatement();
// now update the row
int updateCount = stmt2.executeUpdate("update booklist " +
                   "set author = 'Zamyatin, Evgenii Ivanovich' " +
                   "where current of " + cursorName);
```

CODE EXAMPLE 16-9   Updating a row using positioned updates

The method DatabaseMetaData.supportsPositionedUpdates returns true if
the JDBC driver and DBMS support this facility.

16.2.5   Closing a ResultSet Object

A ResultSet object is explicitly closed when

- The close method on the ResultSet is executed, thereby releasing any
  external resources
- The Statement or Connection object that produced the ResultSet is
  explictly closed

A ResultSet object is implicitly closed when

- The associated Statement object is re-executed
- The ResultSet is created with a Holdability of
  CLOSE_CURSORS_AT_COMMIT and an implicit or explicit commit occurs

---

Note – Some JDBC driver implementations may also implicitly close the
ResultSet when the ResultSet type is TYPE_FORWARD_ONLY and the next
method of ResultSet returns false.

Once a ResultSet has been closed, any attempt to access any of its methods with
the exception of the isClosed method will result in a SQLException being
thrown. ResultSetMetaData instances that were created by a ResultSet that
has been closed are still accessible.

Note – The closing of a `ResultSet` object does **not** close the `Blob`, `Clob`, `NClob` or `SQLXML` objects created by the `ResultSet`. `Blob`, `Clob`, `NClob` and `SQLXML` objects remain valid for at least the duration of the transation in which they are created, unless their `free` method is invoked.

Customized Type Mapping

This chapter describes the support that the JDBC API provides for mapping SQL structured and distinct types to classes in the Java programming language. Because the mechanism for this custom mapping is an extension of the existing `getObject` and `setObject` mechanism, it involves minimal extensions to the JDBC API from the user's point of view.

17.1   The Type Mapping

The SQL user-defined types (UDTs), structured types and `DISTINCT` types, can be given a custom mapping to a class in the Java programming language. The default is for a driver to use the default mappings between SQL data types and types in the Java programming language. The default mapping for an SQL structured type is to the interface `Struct`; the default mapping for an SQL `DISTINCT` type is to the type to which the underlying type is mapped. If a custom mapping has been set up for a UDT, the driver will use the custom mapping instead of the default mapping when an application calls the `getObject` or `setObject` methods on that UDT.

Setting up a custom mapping requires two things:

1. Writing an implementation of the `SQLData` interface for the UDT. This class typically maps the attribute(s) of an SQL structured type (or the single attribute of a `DISTINCT` type) to fields. There is, however, great latitude allowed in how a UDT is custom mapped. It is expected that most `SQLData` implementations will be created using a tool.

2. Putting an entry in a `java.util.Map` object. The entry must contain the following two items:

   a. The fully qualified name of the SQL UDT that is to be mapped.

b. The `Class` object for the `SQLData` implementation. It is an error if the class listed in a type map entry does not implement the `SQLData` interface.

For example, if the UDT is named `mySchemaName.AUTHORS` and the `SQLData` implementation is the class `Authors`, the entry for the type map associated with the `Connection` object *conn* would look like CODE EXAMPLE 17-1.

```
java.util.Map map = conn.getTypeMap();

map.put("mySchemaName.AUTHORS", Class.forName("Authors"));

conn.setTypeMap(map);
```

CODE EXAMPLE 17-1   Putting an entry in a connection's type map

The method `Connection.getTypeMap` returns the type map associated with the `Connection` object *conn*; the method `Connection.setTypeMap` sets the given `java.util.Map` object as the type map for *conn*.

When an SQL value with a custom mapping is being retrieved (by the method `ResultSet.getObject`, `CallableStatement.getObject`, or any of the other methods that materialize an SQL value's data on the client), the driver will check to see if there is an entry in the connection's type map for the SQL value that is to be retrieved. If there is, the driver will map the SQL UDT to the class specified in the type map. If there is no entry for the UDT in the connection's type map, the UDT is mapped to the default mapping.

Certain methods may take a type map as a parameter. A type map supplied as a parameter supersedes the type map associated with the connection. A UDT that does not have an entry in the type map supplied as a parameter will be mapped to the default mapping. When a type map is explicitly supplied to a method, the connection's type map is never used.

17.2   Class Conventions

A class that appears in a type map entry must do the following:

1. Implement the interface `java.sql.SQLData`

2. Provide a niladic constructor, that is, a constructor that takes no parameters

The `SQLData` interface contains methods that convert instances of SQL UDTs to Java class instances and that convert Java class instances back to SQL UDTs. For example, the method `SQLData.readSQL` reads a stream of data values and builds a Java object, while the method `SQLData.writeSQL` writes a sequence of values from a Java object to a stream. These methods will typically be generated by a tool that 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, each attribute value, which may itself be a structured type, appears fully (its structure recursively elaborated) in the stream before the next attribute. In addition, the attributes of an SQL structured type must appear in the stream in the order in which they are declared in the type definition. 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 supertype must appear before attributes of a subtype.

If multiple inheritance is used, then the attributes of supertypes should appear in the stream in the order in which the supertypes are listed in the type declaration. This protocol does not require the database server to have any knowledge of the Java programming language. However, as there is no support for multiple inheritance in SQL:2003, this issue should not arise.

17.3 Streams of SQL Data

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

17.3.1 Retrieving Data

In a custom mapping, 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. CODE EXAMPLE 17-2 illustrates how a driver can use an SQLInput stream to provide values for the fields of an SQLData object. The SQLData object—the this object in the example—contains three persistent fields: the String *str*, the Blob object *blob*, and the Employee object *emp*.

```
SQLInput sqlin;

...

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

```
this.emp = (Employee)sqlin.readObject();
```

CODE EXAMPLE 17-2 Retrieving data using the `SQLInput` interface

The `SQLInput.readString` method reads a `String` value from the stream; the `SQLInput.readBlob` method reads a `Blob` value from the stream. By default, the `Blob` interface is implemented using an SQL locator, so calling the method `readBlob` doesn't materialize the SQL `BLOB` contents on the client. The `SQLInput.readObject` method retrieves an object reference from the stream. In the example, the `Object` returned is narrowed to an `Employee` object.

There are a number of additional methods defined on the `SQLInput` interface for reading each of the types (`readLong`, `readBytes`, and so on). The `SQLInput.wasNull` method can be called to check whether the last value read was SQL `NULL` in the database.

17.3.2   Storing Data

When an instance of a class that implements `SQLData` is passed to a driver as an input parameter via the `setObject` method, the JDBC driver calls the object's `SQLData.writeSQL` method. It also creates an `SQLOutput` stream to which the method `writeSQL` writes the attributes of the custom mapped UDT. The method `writeSQL` will typically have been generated by a tool from an SQL type definition. CODE EXAMPLE 17-3 illustrates the use of the `SQLOutput` object *SQLData*.

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

CODE EXAMPLE 17-3 Storing data using the `SQLOutput` interface

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: the `String` *str*, the `Blob` object *blob*, and the `Employee` object *emp*. Each field is written in turn to the `SQLOutput` stream, *SQLData*. The `SQLOutput` interface contains methods for writing each of the types defined in the JDBC API.

17.4 Examples

This section gives examples of SQL code as well as code in the Java programming language. SQL code is used for creating structured types, creating tables for instances of those types, populating the tables with instances of the structured types, and creating an SQL DISTINCT type. This code sets up the SQL values that will be mapped to classes in the Java programming language.

The examples of code in the Java programming language create implementations of the SQLData interface for the newly created SQL UDTs and also show how a class in the Java programming language can mirror SQL inheritance for structured types.

17.4.1 An SQL Structured Type

CODE EXAMPLE 17-4, which defines the structured types PERSON, FULLNAME, and RESIDENCE, shows that it is possible for an attribute to be a REF value or another structured type. PERSON and RESIDENCE each have an attribute that is a REF value, and the REF value in one structured type references the other structured type. Note also that FULLNAME is used as an attribute of PERSON.

```
CREATE TYPE RESIDENCE AS
(
                DOOR NUMERIC(6),
                STREET VARCHAR(100),
                CITY VARCHAR(50),
                OCCUPANT REF(PERSON)
) NOT FINAL


CREATE TYPE FULLNAME AS
(
                FIRST VARCHAR(50),
                LAST VARCHAR(50)
) NOT FINAL


CREATE TYPE PERSON AS
(
                NAME FULLNAME,
```

```
                    HEIGHT REAL,
                    WEIGHT REAL,
                    HOME REF(RESIDENCE)
) NOT FINAL
```

CODE EXAMPLE 17-4 Creating SQL structured types

The types created in CODE EXAMPLE 17-4 are presumed to be created in the current
schema for the following examples.

CODE EXAMPLE 17-5 creates two tables that are maintained by the DBMS
automatically. The CREATE statements do two things:

1. Create tables that store instances of the structured types named in the OF part of
   the statement (RESIDENCE in the first one, PERSON in the second). Each of the
   subsequent INSERT INTO statements adds a new row representing an instance of
   the UDT.

2. Create a REF value that is a pointer to each instance that is inserted into the table.
   As indicated in the CREATE statement, the REF value is generated by the system,
   which is done implicitly. Because REF values are stored in the table, they are
   persistent pointers. This contrasts with LOCATOR types, which are logical pointers
   but exist only as long as the transactions in which they are created.

```
CREATE TABLE HOMES OF RESIDENCE
                (REF IS OID SYSTEM GENERATED,
                OCCUPANT WITH OPTIONS SCOPE PEOPLE)
CREATE TABLE PEOPLE OF PERSON
                (REF IS OID SYSTEM GENERATED,
                HOME WITH OPTIONS SCOPE HOMES)
```

CODE EXAMPLE 17-5 Creating tables to store instances of a structured type

CODE EXAMPLE 17-6 uses INSERT INTO statements to populate the tables created in
CODE EXAMPLE 17-5. For example, the INSERT INTO PEOPLE statement inserts an
instance of the UDT PERSON into the table PEOPLE. When this command is executed,
the DBMS will also automatically generate a REF value that is a pointer to this
instance of PERSON and store it in the column OID (the column name specified in the
CREATE statement that created the table PEOPLE).

Each column value in these special tables is an attribute of the UDT, which may
itself be a UDT. For example, the first attribute of the UDT PERSON is the value in
the column NAME, which must be an instance of the UDT FULLNAME. The example
assumes that the UDT FULLNAME has an additional two parameter constructor.

A column value may also be a reference to an SQL structured type. For example, the attribute OCCUPANT of the UDT RESIDENCE is of type REF(PERSON). It takes an SQL SELECT statement to retrieve the REF value from the table HOMES and use it as the value for OCCUPANT, which is shown at the end of CODE EXAMPLE 17-6.

```
INSERT INTO PEOPLE (NAME, HEIGHT, WEIGHT) VALUES
(
                NEW 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'
```

CODE EXAMPLE 17-6   Populating and updating tables that store instances of structured types

17.4.2   SQLData Implementations

The Java classes defined in CODE EXAMPLE 17-7 are mappings of the SQL structured types used in the examples in Section 17.4.1. 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 in the example.

In each implementation of SQLData, the method SQLData.readSQL reads the attributes in the order in which they appear in the SQL definition of the structured type. Attributes are also read in "row order, depth-first" order, where the complete

structure of each attribute is read recursively before the next attribute is read. The method `SQLData.writeSQL` writes each attribute to the output stream in the same 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; }
```

```java
    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);
    }
}
```

CODE EXAMPLE 17-7   Classes implementing the SQLData interface

CODE EXAMPLE 17-8 puts entries for custom mappings in the connection's type map.
Then it retrieves the Ref instance stored in the OCCUPANT column of the table
HOMES. This Ref instance is set as a parameter in the where clause of the query to
get the name of the occupant. When the method getObject is called to retrieve an
instance of FULLNAME, the driver looks in the connections type map and uses the
SQLData implementation, Fullname, to custom map the FULLNAME value.

```
// set up 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);
```

CODE EXAMPLE 17-8   Retrieving a custom mapping

17.4.3 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, which inherits from PERSON:

```
CREATE TYPE PERSON AS
                (NAME VARCHAR(20),
                BIRTH DATE)
                NOT FINAL;
CREATE TYPE STUDENT UNDER PERSON AS
                (GPA NUMERIC(4,2))
                NOT FINAL;
```

CODE EXAMPLE 17-9 Creating a hierarchy of SQL types

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 cascade each call to the corresponding method in its superclass in order to read or write the superclass attributes before reading or writing the subclass attributes.

```java
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 BigDecimal 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.readBigDecimal();
      }

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

CODE EXAMPLE 17-10 Mirroring SQL type hierarchies in Java classes

The Java class hierarchy need not mirror the SQL inheritance hierarchy. For
example, the class `Student` above could have been declared without a superclass.
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.

17.4.4   Example Mapping of SQL DISTINCT Type

CODE EXAMPLE 17-11 illustrates creating an SQL DISTINCT type, MONEY, and
CODE EXAMPLE 17-12 illustrates a Java class, Money, that represents it.

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

CODE EXAMPLE 17-11 Creating an SQL DISTINCT type

```
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);
  }
}
```

CODE EXAMPLE 17-12 Java class that represents a DISTINCT type

17.5   Effect of Transform Groups

Transform groups (SQL:2003) can be used to convert a user-defined SQL type into
predefined SQL types. This transformation is performed by the underlying data
source before it is returned to the JDBC driver.

If transform groups are used for a user-defined type, and the application has not
defined a mapping for that type to a Java class, then the ResultSetMetaData
method getColumnClass should return the Java class corresponding to the data
type produced by the transformation function (that is, String for a VARCHAR).

Note – This is consistent with the behavior for DISTINCT types.

If transform groups are used for a UDT, and the application has defined a mapping for that type to a Java class, then the `SQLInput` stream delivered by the JDBC driver during an invocation of the method `readSQL` contains only a single value, that is, the result produced by the transformation function. The same model holds for the method `writeSQL`.

17.6 Generality of the Approach

Users have great flexibility in customizing 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 JavaBeans components 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 the method `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 a connection's type map incrementally. This flexibility will allow users to map SQL types appropriately for different kinds of applications.

17.7 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`, the driver executes the SQL statement as if the SQL literal `NULL` had appeared in its place.

```
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 not 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.

Relationship to Connectors

The Java EE Connector Architecture 1.0 Specification defines a set of contracts that allow a resource adapter to extend a container in a pluggable way. A resource adapter provides connectivity to an external system from the application server. The resource adapter's functionality is similar to that provided by the JDBC interfaces used in the Java EE platform to establish a connection with a data source. These interfaces, which the Connector specification refers to as the *service provider interface* (SPI), are the following:

- `DataSource`
- `ConnectionPoolDataSource`
- `XADataSource`

Additionally, the Connector Architecture defines a packaging format to allow a resource adapter to be deployed into a Java EE compliant application server.

18.1 System Contracts

The system contracts defined in the Connector specification describe the interface between an application server and one or more resource adapters. This interface allows a resource adapter to be bundled in such a way that it can be used by any application server that supports the system contracts.

The following standard contracts are defined between an application server and a back end system:

- A connection management contract that enables application components to connect to a back end system.

  The connection management contract is equivalent to the services described by the JDBC interfaces `DataSource` and `ConnectionPoolDataSource`.

- A transaction management contract between the transaction manager and a back end system supporting transactional access to its resources.

The transaction contract is equivalent to the services described by the JDBC interface `XADataSource`.

- A security contract that enables secure access to a back end system.

    The security contract does not have an equivalent in the JDBC API. Authentication in the JDBC API always consists of providing a user name and a password.

The JDBC specification does not make a distinction between its application programming interface (API) and the SPI. However, a driver can map the JDBC interfaces in the SPI to the Connector system contracts.

18.2 Mapping Connector System Contracts to JDBC Interfaces

Driver vendors who want to supply JDBC drivers that use the Connector system contracts have several options:

1. To write a set of classes that wrap a JDBC driver and implement the Connector system contracts. Constructing these wrappers is fairly straightforward and should allow JDBC driver vendors to provide resource adapters quickly enough so that they are available when application server vendors have implemented the Connector contracts.

2. To implement the Connector system contracts natively. This approach avoids the overhead of wrapper classes, but the implementation effort may be more involved and time-consuming. This alternative is a more long-term option.

Either approach will allow JDBC driver vendors to package their drivers as resource adapters and get all the benefits of pluggability, packaging, and deployment.

Note – There are no plans to deprecate or remove the current JDBC interfaces, `DataSource`, `ConnectionPoolDataSource` and `XADataSource`.

18.3 Packaging JDBC Drivers in Connector RAR File Format

Resource adapters can be packaged, along with a deployment descriptor, into a Resource adapter ARchive, or RAR file. The RAR file contains the Java classes/interfaces, native libraries, deployment descriptor, and other resources needed to deploy the adapter.

The deployment descriptor maps the classes in the resource adapter to the specific roles that they perform. The descriptor also details the capabilities of the resource adapter in terms of what level of transactional support it provides, the kind of security it supports, and so on.

CODE EXAMPLE 18-1 is an example of a deployment descriptor for a JDBC driver. The class `com.acme.JdbcManagedConnectionFactory` could be supported by an implementation of `javax.sql.XADataSource`. The resource adapter section contains information on the level of transaction support, the mechanism used for authentication, and configuration information for deploying the data source in the JNDI namespace.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE connector PUBLIC '-//Sun Microsystems, Inc.//DTD Connector
1.0//EN' 'http://java.sun.com/j2ee/dtds/connector_1_0.dtd'>

<connector>
                <display-name>Acme JDBC Adapter</display-name>
                <vendor-name>Acme Software Inc</vendor-name>
                <spec-version>1.0</spec-version>
                <version>1.0</version>
                <eis-type>JDBC Database</eis-type>
                <resourceadapter>
                    <managedconnectionfactory-
class>com.acme.JdbcManagedConnectionFactory</
managedconnectionfactory-class>
                    <connectionfactory-
interface>javax.sql.DataSource<connectionfactory-interface>
                    <connectionfactory-impl-
class>com.acme.JdbcDataSource<connectionfactory-impl-class>
                    <connection-interface>java.sql.Connection</
connection-interface>
```

```
                        <connection-impl-
class>com.acme.JdbcConnection</connection-impl-class>
                        <transaction-support>xa_transaction</
transaction-support>
                        <config-property>
                            <config-property-name>XADataSourceName</
config-property-name>
                            <config-property-type>java.lang.String</
config-property-type>
                            <config-property-value>jdbc/XAAcme</config-
property-value>
                        </config-property>
                        <auth-mechanism>
                            <auth-mech-type>basic-password</auth-mech-
type>
                            <credential-
interface>javax.resource.security.PasswordCredential</credential-
interface>
                        </auth-mechanism>
                        <reauthentication-support>false</
reauthentication-support>
                </resourceadapter>
</connector>
```

CODE EXAMPLE 18-1   Example of a resource adapter deployment descriptor.

See the Connector specification for more details.

CHAPTER **19**

Ease of Development

JDBC 4.0 introduces a set of classes and concepts that simplify the work that a developer must do when accessing data stores.

19.1 Overview

The JDBC Ease of Development features are intended to:

- Make it easier to execute SQL queries which return a single result set; execute SQL Data Manipulation Language (DML) statements that return a row count or that return nothing.
    - This reduces the amount of code required to execute queries that normally would be executed using the `Statement.executeQuery` and `Statement.executeUpdate` methods.
- Simplify the developer experience for users who want to access, navigate and modify their data as rows and columns.
- Provide a means for more strongly typed data.
- Process the returned data in a connected or disconnected environment.
- Leverage the new features added to J2SE 5.0 such as generics and annotations.

The JDBC Ease of Development features are **not** intended to:

- Be used as an Object-Relational Mapping (ORM) Technology. Developers who need the features found in an ORM Technology should consider using the Java Persistence API.
- Address more complex JDBC application needs such as processing multiple result sets.

19.1.1 JDBC Annotations

JDBC Annotations assist developer by reducing the amount of code that must be written when using the JDBC APIs. The annotations will be used in combination with Query interfaces and `DataSet` objects to simplify the access and processing of data that is returned as a single SQL result set.

TABLE 19-1 lists the annotations provided in JDBC 4.0.

TABLE 19-1   JDBC Annotations

Annotation	Description
AutoGeneratedKeys	Defines the fields within a `DataSet` representing the columns that are to be returned for auto-generated keys
ResultColumn	Maps a field within a `DataSet` to a specific column in a SQL ResultSet
Select	Associates a SQL Select statement with a method in a Query Interface
Update	Associates a SQL statement that may return an update count, with a method in a Query Interface

19.1.2 Query Interface

A Query interface defines a set of methods that are decorated with JDBC annotations. These annotations describe the SQL queries and SQL update statements to be invoked by a given method. A Query interface must be a subinterface of the `BaseQuery` interface. The interface must also specify how the data returned for a SQL query should be bound to a `DataSet`.

19.1.3 BaseQuery Interface

The BaseQuery interface must be the superinterface for all Query interfaces. This interface defines methods for closing a concrete implementation of a Query interface and for determining if an instance of a Query object has been closed.

19.1.4 DataSet interface

The DataSet interface is a subinterface of java.util.List and provides a type safe view of the data returned from the execution of a SQL query. A DataSet interface is a parameterized type. The parameter type is a *data class* describing the columns for the rows that are returned from a method on a Query interface decorated by the Select annotation.

A DataSet may operate in a connected or disconnected mode. When used in a connected mode, the DataSet is implemented as a ResultSet. A disconnected DataSet is implemented as a CachedRowSet.

A DataSet object allows for the iteration through the rows that were returned using the java.util.Iterator API. A DataSet object allows user to iterate through the rows that were returned using the java.util.Iterator API.

19.1.5 User-Defined Class

A user-defined class is used to represent the type parameter for a DataSet. The class defines the fields that represent the columns returned from an underlying data store. The user-defined class may be specified in two ways: as a structure or as a JavaBeans™ component. The user-defined class must be created with an access modifier of public.

Note – Throughout the rest of this chapter, this "user-defined class" will be referred to as *data class*.

When the *data class* is used as a simple structure, it consists of public fields whose names match the columns in the data returned from a data source.

```
public class Person {
    public String firstName;
    public String lastName;
}
```

CODE EXAMPLE 19-1   *data class* as a structure

The *data class* can also be represented as a JavaBeans^{TM} component, providing setters and getters, that match the column names in the returned data to access the fields.

```
public class Person {
     private String firstName;
     private String lastName;
     public String getFirstName() { return firstName;}
     public String getLastName() { return lastName;}
     public void setFirstName(String fName) { firstName= fName;}
     public void setLastName(String lName) { lastName=lName;}
}
```

CODE EXAMPLE 19-2 *data class* as a JavaBean

19.1.6   QueryObjectGenerator Interface

The `QueryObjectGenerator` interface may be implemented by JDBC drivers. A `QueryObjectGenerator` is used to process JDBC annotations and to provide the mapping between `DataSet` objects and the resulting data returned by a method decorated with a `Select` annotation. The interface also contains the method `createQueryObject` that returns an instance of a Query interface.

The method `DatabaseMetaData.providesQueryObjectGenerator` may be used to determine whether a JDBC driver provides a `QueryObjectGenerator` implementation.

An implementation of `QueryObjectGenerator` is provided by Java SE 6.

19.2   Creating an instance of a Query interface

This section describes how to create a concrete implementation of a Query interface.

The database table that will be used in the example was created by the following DDL:

```
create table mammals(firstName varchar(20), lastName varchar(20), int
age, int weight, description varchar(50), longDescription
varchar(254), int key1, int key2);
```

CODE EXAMPLE 19-3   DDL used to create the mammal table

19.2.1   Creating a Query Interface

The following section will provide an example of creating a Query interface.

Prior to creating a Query interface, developers should create any needed *data classe*s that will be supplied as a type parameter for a `DataSet`. Please refer to the `DataSet` section for additional details.

```
public class Mammal {

     public String firstName;

     public String lastName;

     public int      age;

     public int      weight;

     public String description;

}
```

CODE EXAMPLE 19-4   Mammal *data class*

The methods of the Query interface are decorated with `Select` and `Update` annotations. Methods decorated by `Select` annotations will return instances of `DataSet<T>`.

```
interface MyQueries extends BaseQuery {

     @Select(sql="SELECT lastName, description FROM mammals")

     DataSet<Mammal> getAllMammals();

     @Update(sql="delete from mammals")

     int deleteAllMammals();

}
```

CODE EXAMPLE 19-5   A sample Query interface

19.2.2   Concrete Query Interface implementation

A concrete implementation of a Query interface can be created by invoking either the `Connection.createQueryObject` or `DataSource.createQueryObject` method passing a Query interface as its parameter.

```
Connection con = DriverManager.getConnection(url, props);
MyQueries myQueries = con.createQueryObject(MyQueries.class);
```
CODE EXAMPLE 19-6   Creating an instance of a SQL Interface

19.2.2.1   Connection.createQueryObject and DataSource.createQueryObject methods

JDBC drivers must implement the createQueryObject method on the
Connection and DataSource interface. If the JDBC driver does not provide its
own implementation of a QueryObjectGenerator, it must invoke the default
QueryObjectGenerator implementation provided by Java SE 6.

When DataSource.createQueryObject is used for creating a Query object
instance, a Connection will be obtained from the DataSource and used when
executing a method on the Query interface. The QueryObjectGenerator
implementation will be responsible for closing the Connection.

If Connection.createQueryObject is used for creating a Query object instance,
the application is responsible for closing the connection.

19.2.2.2   Invoking Query Interface Methods

Once you have created an implementation of the Query interface, you may invoke
any of the methods that are defined by the interface.

```
DataSet<Mammal> mammalRows = myQueries.getAllMammals();
```
CODE EXAMPLE 19-7   invoking a method on the Query interface

19.2.2.3   Closing a Query Object

An application must explicitly close a Query object by calling the close method.
The close method will close any connected DataSets created from the Query
object, thereby releasing any external resources and making it available for garbage
collection.

19.3   JDBC Annotations

The following section defines the JDBC 4.0 annotation types. All of the JDBC annotations specify a retention policy of RUNTIME. Please refer to the `RetentionPolicy` API specification for additional information on the retention policy for annotations.

19.3.1   Select Annotation

The `Select` annotation is used to specify a SQL query statement that decorates a Query interface method. A `Select` annotation must include a `sql` annotation element. Additional annotation type elements may be specified to define the configuration of the returned `DataSet` object. The `Select` annotation must be used in conjunction with SQL statements that returns a single result set.

TABLE 19-2   Select annotation elements

Type	Name	Description	Default
String	sql or value	The SQL Select statement to be invoked	None
String	tableName	The name of the table which will be updated when a DataSet.sync method is called	None
boolean	readOnly	Indicates if the returned DataSet is read-only	true
boolean	connected	Specifies whether the DataSet is connected to the backend data source	true
boolean	allColumnsMapped	Indicates if there is a 1-to-1 mapping between the column names in the `sql` annotation element and the fields in the DataSet	false
boolean	scrollable	Indicates if the DataSet when used in a connected mode is scrollable	false

19.3.1.1 sql annotation element

The `sql` annotation element is used to specify a SQL query that returns tabular data. If the specified SQL query returns an update count, a `SQLRuntimeException` will be thrown.

You may also use the `value` annotation element instead of the `sql` annotation element to specify the SQL Query. The `value` annotation element would normally only be used when no other annotation elements are specified on the `Select` annotation.

```
interface MyQueries extends BaseQuery {
    @Select("SELECT lastName, description FROM mammals")
    DataSet<Mammal> getAllMammals();
}
```
CODE EXAMPLE 19-8   A Select annotation using the value annotation element

If the `sql` annotation element and the `value` annotation element are both specified in a `Select` annotation, then a `SQLRuntimeException` will be thrown.

Note – If the specified SQL Query produces a read-only result set, for example the query contains a GROUP BY clause or returns columns from multiple tables, the `DataSet` will be considered read-only. If a `DataSet` is read-only, any attempt to invoke the `DataSet.insert`, `DataSet.delete` or `DataSet.modify` methods will result in a `SQLRuntimeException` being thrown.

19.3.1.2 readOnly annotation element

The `readOnly` annotation element is used to specify whether a `DataSet` may be modified. If set to `true` or not specified, the fields within a `DataSet` may not be modified. If set to `false`, the fields within a `DataSet` may be modified. Any attempt to modify a field within a `DataSet` when `readOnly` is set to `true` results in a `SQLRuntimeException` being thrown when the `DataSet` methods `insert`, `modify` or `delete` are invoked.

19.3.1.3 connected annotation element

The `connected` annotation element is used to specify whether a `DataSet` is connected to the data source. If set to `true` or is not specified, the `DataSet` object is connected to the underlying data source. Any modifications to the `DataSet` would be made directly to the underlying data source.

If the value of the `connected` annotation element is set to `false`, the `DataSet` operates in a manner similar to a `CachedRowSet` and any modifications to the `DataSet` object are not propagated to the data source until the `DataSet.sync` method is called.

19.3.1.4 allColumnsMapped annotation element

The `allColumnsMapped` annotation element is used to indicate whether all columns returned by a method decorated by a `Select` annotation must be present in the *data class* for a `DataSet`. The returned columns are mapped by matching their column names to the field names in the *data class* for a `DataSet`.

If the `allColumnsMapped` annotation element is set to `false` or is not specified, then any columns returned by a method decorated by a `Select` annotation that are not defined in the *data class* will be ignored.

```
public class Mammal {
    public String firstName;
    public String lastName;
    public int    age;
    public int    weight;
    public String description;
 }
```

CODE EXAMPLE 19-9   DataSet parameterized type Mammal

```
interface MyQueries extends BaseQuery {
    @Select(sql="SELECT lastName, longDescription FROM mammals",
    allColumnsMapped=false)
    DataSet<Mammal> getAllMammals();
}
```

CODE EXAMPLE 19-10   allColumnsMapped annotation element with a value of false

In the above example, `Mammal.lastName` would contain the value of the column "lastName" and the value of the column "longDescription" would be ignored because the Mammal class does not define the "longDescription" field.

If the `allColumnsMapped` annotation element is set to `true`, then every column that is returned by a method decorated by a `Select` annotation must exist in the `DataSet` parameterized type. If a returned column does not exist in the `DataSet` parameterized type, a `SQLRuntimeException` must be thrown. The mapping of the returned columns to the fields in a `DataSet` instance is done in a case insensitive manner.

If the `allColumnsMapped` annotation element is set to `true` and a *data class* for a `DataSet` contains more fields than there are returned columns, a `SQLWarning` will be generated. The method, `BaseQuery.getWarnings` may be called to obtain any generated `SQLWarnings` after invoking a method on a Query interface. Any fields that do not map to a returned column value will be set to the default value for the datatype as defined in the Java Language Specification.

In the following example, `Mammal.firstName` would contain the value of the column "firstName", *Mammal.lastName* would contain the value for the column "lastName" and `Mammal.description` would contain the value for the column "description".

```
interface MyQueries extends BaseQuery {

    @Select(sql="SELECT lastName, firstName, description FROM
mammals",

    allColumnsMapped=true)

    DataSet<Mammal> getMammalNames();

}
```

CODE EXAMPLE 19-11 allColumnsMapped annotation element with a value of true

19.3.1.5   scrollable annotation element

The `scrollable` annotation element is used to specify whether a `DataSet` that is used in a connected mode is scrollable. If set to `false` or is not specified, the `DataSet` object is not scrollable and is treated as a `ResultSet` type of `TYPE_FORWARD_ONLY`. If set to `true`, the `DataSet` object is scrollable. It is recommended that a `ResultSet` type of `TYPE_SCROLL_SENSITIVE` be used when a `DataSet` is scrollable. If the driver cannot support the type used by the QueryObject generator, it generates an `SQLWarning` on the `createQueryObject` method invocation.

If a `DataSet` is used in a disconnected mode, the `scrollable` annotation element is ignored and the `DataSet` is scrollable.

19.3.1.6 tableName annotation element

The tableName annotation element is used to specify the table in the underlying data store which will be modified when the method DataSet.sync is called. A JDBC programmer would utilize this annotation element when using a DataSet in a disconnected manner.

The tableName annotation element is not required to be used when using a DataSet in a connected manner, and if specified, will be ignored.

```
interface MyQueries extends BaseQuery {

    @Select(sql="SELECT lastName, description FROM mammals",
tableName="mammals")

    DataSet<Mammal> getAllMammals();

}
```

CODE EXAMPLE 19-12 A Select annotation using the tableName annotation element

In the above example, the table "mammal" will be updated when the the sync method is invoked on a disconnected DataSet.

19.3.1.7 Parameterized sql Annotation element

The sql annotation element allows developers to specify parameter markers similar to PreparedStatements. Parameter markers are defined as:

- A String value designated by a question mark (?) prefix followed by an integer.
- The numbering for parameter markers starts at 1. For example: ?1 represents the first argument of the decorated method.
- The number of distinct parameter markers must not exceed the number of method parameters for the method that was decorated by the Select annotation. If this occurs, a SQLRuntimeException will be thrown.
- A parameter marker may be specified more than once in a sql annotation element.

```
interface MyQueries extends BaseQuery {

    @Select(sql="SELECT * from mammals where firstName= ?1 and
lastName= ?2")

    DataSet<Mammal> getBigMammals(String fName, String lName);

}
```

CODE EXAMPLE 19-13 Specifying parameter markers in a sql annotation element

19.3.2 Update Annotation

The `Update` annotation is used to decorate a Query interface method with a SQL statement that returns an update count. An `Update` annotation must include a `sql` annotation type element. Additional annotation type elements may be specified to define the configuration of the returned `DataSet` object. The `Update` annotation must not be used in-conjunction with SQL statements that return a `ResultSet`.

TABLE 19-3   Update annotation elements

Type	Name	Description	Default
String	sql or value	The SQL Select statement to be invoked	None
GeneratedKeys	keys	Specifies whether auto-generated keys are returned	GeneratedKeys.NO_KEYS_RETURNED

19.3.2.1 sql annotation element

The `sql` annotation element is used to specify a SQL command that returns an update count. If the specified SQL command returns tabular data instead of an update count, a `SQLRuntimeException` will be thrown.

You may also use the `value` annotation element instead of the `sql` annotation element to specify the SQL command. The `value` annotation element would normally only be used when no other annotation elements are specified on the `Update` annotation.

```
interface MyQueries extends BaseQuery {
    @Update("delete from mammals")
    int deleteAllMammals();
}
```

CODE EXAMPLE 19-14 A sample Update annotation using the value annotation element

If the `sql` annotation element and the `value` annotation element are both specified in a `Update` annotation, a `SQLRuntimeException` will be thrown.

19.3.2.2   keys annotation element

The `keys` annotation element may be used to determine whether any auto-generated keys will be returned. If the value for the `keys` annotation element is set to `Generated.NO_KEYS_RETURNED` or is not specified, then any auto-generated keys that might have occurred as a result of the invocation of the specified SQL query will not be returned. A `keys` annotation element value of `GeneratedKeys.RETURNED_KEYS_DRIVER_DEFINED` indicates that the JDBC driver implementation will determine the columns to return which best represent the auto-generated keys. A `keys` annotation element value of `GeneratedKeys.RETURNED_KEYS_COLUMN_SPECIFIED` indicates that the field names in the *data class,* defined as the return type for a method decorated by a `Update` annotation, indicate the columns to be returned as the auto-generated keys.

Please refer to the section "AutoGeneratedKeys annotation" on page 192 for additional details on how to access the returned auto-generated keys.

19.3.2.3   Returning an Update Count

It is possible to have a method decorated by a `Update` annotation return an update count. To have an update count returned, the decorated method must define a return parameter of type `int`. If an update count is not required by the application, the return type for the decorated method can be `void`.

```
interface MyQueries extends BaseQuery {

    @Update(sql="update mammals set age = ?1 where lastName = ?2 and
firstName= ?3")

    int setMammalAge(int age, String lName, String fName);

}
```
CODE EXAMPLE 19-15 Having a method decorated by the Update annotation return an update count

19.3.2.4   Parameterized sql Annotation element

The `sql` annotation element allows developers to specify parameter markers similar to `PreparedStatements`. Parameter markers are defined as:

- A String value designated by a question mark (?) prefix followed by an integer.
- The numbering for parameter markers starts at 1. For example: ?1 represents the first argument of the decorated method.

- The number of distinct parameter markers must not exceed the number of method parameters for the method that was decorated by the Update annotation. If this occurs, a SQLRuntimeException will be thrown.
- A parameter marker may be specified more than once in a sql annotation element.

```
interface MyQueries extends BaseQuery {

    @Update(sql="update mammals set age = ?1 where lastName = ?2 and
firstName= ?3")

    int setMammalAge(int age, String lName, String fName);

}
```
CODE EXAMPLE 19-16 Specifying parameter markers in a sql annotation element

19.3.3   ResultColumn Annotation

The ResultColumn annotation decorates a field within a *data class* for a DataSet. It is used to map a DataSet parameterized type field to a column in a returned SQL ResultSet, and specify that a DataSet field will be used to uniquely identify a row when the DataSet is not connected to the underlying data source.

TABLE 19-4   ResultColumn annotation elements

Type	Name	Description	Default
String	name or value	The name of the column in the SQL ResultSet	None
Boolean	uniqueIdentifier	True if the field represents a column to use to uniquely identify a row when modifying a row when working in a disconnected manner.	false

19.3.3.1   name annotation element

The name annotation element defines the column in the returned SQL ResultSet that the decorated field will be mapped to.

```
create table newMammal(firstName varchar(20), lastName varchar(20),
int age, int kilos, description varchar(50));
```
CODE EXAMPLE 19-17 DDL used to create the newMammal table

```
public class Mammal {

                public String name;

                public String description;

                @ResultColumn(name="kilos") public int weight;

}
```

CODE EXAMPLE 19-18 Mapping a DataSet parameterized type field to a SQL ResultSet column using the ResultColumn annotation

In the above example, the SQL column 'kilos' is mapped to the `DataSet` *data class* field 'weight'. The JDBC programmer would reference the 'kilos' column in the returned ResultSet as `Mammal.weight`.

You may also use the `value` annotation element instead of the `name` annotation element to specify the name of the column. The `value` annotation element would normally only be used when no other annotation elements are specified on the `ResultColumn` annotation.

```
public class Mammal {

                public String name;

                public String description;

                @ResultColumn("kilos") public int weight;

}
```

CODE EXAMPLE 19-19 Using the ResultColumn annotation with the value annotation element

If the `name` annotation element and the `value` annotation element are both specified in a `ResultColumn` annotation, then a `SQLRuntimeException` will be thrown.

19.3.3.2   uniqueIdentifier annotation element

The `uniqueIdentifier` annotation element is used to indicate the fields that are used to uniquely identify a row within a `DataSet` when using the methods `DataSet.modify` or `DataSet.delete`. The value of the `uniqueIdentifier` annotation element is only used when the `sql` annotation element `connect` was set to a value of `false`. If the `uniqueIdentifier` annotation element has a value of `false`, or is not specified, then the field will not be used to identify the row.

If there are no `DataSet` *data class* fields decorated with the `uniqueIdentifier` annotation element set to true, then all fields within the `DataSet` will be used to uniquely identity the row.

The `uniqueIdentifier` annotation element provides functionality similar to the method `CachedRowSet.setKeyColumns`.

19.3.4   AutoGeneratedKeys annotation

The `AutoGeneratedKeys` annotation is used to indicate that a `DataSet` *data class* will be used to store any auto-generated keys that occur as the result of executing a method that is decorated with the `Update` annotation. The *data class* may contain one or more fields that represent the columns to be returned that comprise the auto generated key. The number of columns required in the *data class* to define the auto-generated key depend on the value specified for the `keys` annotation element.

If a value of `GeneratedKeys.RETURN_KEYS_COLUMNS_SPECIFIED` is provided for the `keys` annotation element, the fields in the *data class* represent the columns to be returned as the auto-generated key.

A value of `GeneratedKeys.RETURN_KEYS_DRIVER_DEFINED` for the `keys` annotation element indicates that the JDBC driver determines the columns to return as the auto-generated key. If there are more fields specified than what the JDBC driver determines are to be returned, then a `SQLRuntimeException` is thrown.

Note – Auto generated keys can only be returned when you are accessing the `DataSet` in connected mode.

```
@AutoGeneratedKeys

public class MammalKeys {

     public int key1;

     public int key2;

}

interface MyQueries extends BaseQuery {

     @Update(sql=" insert into
mammals('lastName','firstName','age','weight')values(?2,?3,?1,?4)",

     keys=GeneratedKeys.RETURNED_KEYS_DRIVER_DEFINED)

     DataSet<MammalKeys> addMammal(int age, String lName, String
fName, int weight);

}
```

CODE EXAMPLE 19-20 Retrieving auto-generated keys that are the result of invoking a method decorated by the Update annotation

In the above example, when `MyQueries.addMammal` is invoked, any auto-generated keys are stored in `DataSet<MammalKeys>`. The columns that are returned as the auto-generated key are determined by the JDBC driver.

For additional information on using auto-generated keys with JDBC, please refer to "Retrieving Auto Generated Keys" on page 115.

19.4   DataSet Parameterized Type

A `DataSet` object provides a strongly typed view of a `ResultSet` or a `CachedRowSet`. A `DataSet` object may be accessed in a connected or disconnected mode.

When accessed in a connected mode, the `DataSet` object functions in a similar fashion as an Updatable ResultSet where updates made to the `DataSet` object are applied immediately to the underlying data source.

The behavior of a `DataSet` object used in a disconnected mode is similar to a `CachedRowSet`. Any modifications made to the `DataSet` object are not propagated to the data source until the method `DataSet.sync` is called.

19.4.1   Creating a `DataSet` Parameterized Type Instance

The following are the steps to create an instance of a DataSet:

■ Create a *data class*.

```
public class Mammal {

     public String firstName;

     public String lastName;

     public int    age;

     public int    weight;

     public String description;

}
```

CODE EXAMPLE 19-21 Mammal *data clas*s

■ Create a Query interface that contains one or more methods decorated by a Select annotation.

```
interface MyQueries extends BaseQuery {

     @Select(sql="SELECT lastName, firstName, description FROM
mammals")

     DataSet<Mammal> getMammalNames();

}
```

CODE EXAMPLE 19-22 Query interface

■ Invoke a method on an instantiated Query interface that was decorated by a Select annotation.

```
Connection con = DriverManager.getConnection(url, props);

MyQueries myQueries = con.createQueryObject(MyQueries.class);

DataSet<Mammal> rows = myQueries.getMammalNames();
```

CODE EXAMPLE 19-23 Instaniating a DataSet object

19.4.2   Manipulating a `DataSet`

The following section describes how an application program may modify the contents of a `DataSet` object.

19.4.2.1 Traversing a `DataSet`

The DataSet interface is a subinterface of the `List` interface allowing an application to navigate through a DataSet object.

```
DataSet<Mammal> rows = myQueries.getAllMammals();

for (Mammal m: rows) {

    System.out.println("first name: " + m.firstName);

    System.out.println("last name: " + m.lastName);

    System.out.println("age: " + m.age);

}
```

CODE EXAMPLE 19-24 Traversing the rows in a DataSet

Please refer to the `Collection` and `Iterable` API specification for additional information.

19.4.2.2 Creating a copy of the Current Row

You may obtain a copy of the row that the `DataSet` object is currently positioned at by calling the method `DataSet.getRow`. A call to `DataSet.getRow` does not change the current row position.

```
Mammal mammal = rows.getRow();
```

CODE EXAMPLE 19-25 Creating a copy of the currently positioned row of a DataSet

19.4.2.3 Inserting Rows

You can insert rows into an instance of a `DataSet`. If rows are inserted into a disconnected `DataSet`, the *tableName* annotation element must be specified by the `Select` annotation.

A row is inserted into an instance of a `DataSet` by creating and populating an instance of a *data class* of `DataSet<T>` and passing it as an argument to `DataSet.insert`.

```
public class Contact {
    public String firstName;
    public String lastName;
    public String phoneNumber;
}
interface MyQueries extends BaseQuery {
```

```
    @Select(sql="SELECT lastName, firstName, phoneNumber FROM
contacts")
    DataSet<Contact> getContacts();
}


Connection con = DriverManager.getConnection(url, props);
MyQueries mq = con.createQueryObject(MyQueries.class);
DataSet<Contact> contacts = mq.getContacts();
Contact newRow = new Contact();
newRow.lastName="Doe";
newRow.firstName="Jane";
newRow.phoneNumber= "555 000-0000"
contacts.insert(newRow);
```

CODE EXAMPLE 19-26 Inserting a row into a DataSet

In the above example, an instance of the *data class* Contact is created as the object newRow. The newRow instance is populated and the method DataSet.insert is invoked specifying the newRow object as the parameter to be inserted.

19.4.2.4 Deleting Rows

You may delete rows within a DataSet instance by iterating through the DataSet, positioning to the row to be deleted and executing the delete method. If rows are deleted in a disconnected DataSet, the *tableName* annotation element must be specified in the Select annotation that decorates the method which created the DataSet.


```
Connection con = DriverManager.getConnection(url, props);
MyQueries mq = con.createQueryObject(MyQueries.class);
DataSet<Mammal> rows = mq.getAllMammals();
for (Mammal m: rows) {
    if (m.description.equals("")) {
        rows.delete();
    }
}
```

CODE EXAMPLE 19-27 Deleting the current row from a DataSet

> **Note –** A `SQLRuntimeException` must be thrown if the `delete()` method has been called without positioning to a row within the `DataSet` or if the `DataSet` has been specified to be read-only.

19.4.2.5 Modifying Rows

You may update rows within a `DataSet` instance by iterating through the `DataSet`, positioning to the row to be updated and executing the `modify` method. If rows are modified in a disconnected `DataSet`, the *tableName* annotation element must be specified in the `Select` annotation that decorates the method which created the `DataSet`.

```
Connection con = DriverManager.getConnection(url, props);
MyQueries mq = con.createQueryObject(MyQueries.class);
DataSet<Mammal> rows = mq.getAllMammals();
for (Mammal m: rows) {
   if (m.description.equals("")) {
      m.description ="No value";
      rows.modify();
   }
}
```

CODE EXAMPLE 19-28 Modifying the current row within a DataSet instance

The current positioned row can also be directly modified by calling the `modify` method passing an object representing the replacement row.

```
Mammal newMammal = new Mammal();
newMammal.firstName="Jane";
newMammal.lastName="Doe";
rows.modify( newMammal);
```

CODE EXAMPLE 19-29 Modifying the currently positioned row with a new row

> **Note –** A `SQLRuntimeException` must be thrown if the `modify` method has been called without positioning to a row within the `DataSet` or if the `DataSet` has been specified to be read-only.

19.4.2.6   Creating an empty `DataSet` Object

It is possible to create an empty disconnected `DataSet` that rows may be inserted into. The empty `DataSet` is created by calling a `create` method on a Query interface that consists of a single parameter whose data type is `String` and has a return type of `DataSet<T>`. The parameter to the `create` method represents the name of the table that the `DataSet` will be synchronized to. Once you have an instance of the `DataSet`, create an instance of the class that represents the *data class* of `DataSet<T>`. Update the fields within this instance and invoke the `DataSet.insert` method passing the class instance as an argument. The `DataSet` can then be synchronized with the underlying data store by calling the `DataSet.sync` method.

```
public class Contact {
    public String firstName;
    public String lastName;
    public String phoneNumber;
}

interface MyQueries extends BaseQuery {
   DataSet<Contact> create(String tableName);
}

Connection con = DriverManager.getConnection(url, props);
MyQueries mq = con.createQueryObject(MyQueries.class);
DataSet<Contact> contacts = mq.create("phoneBook");
Contact newRow = new Contact();
newRow.lastName="Doe";
newRow.firstName="Jane";
newRow.phoneNumber= "555 000-0000"
contacts.insert(newRow);
// Now insert the rows into the database
contracts.sync(con);
```

CODE EXAMPLE 19-30 Creating and adding rows to an empty DataSet

19.4.2.7 Closing a `DataSet` Object

A connected `DataSet` object is automatically closed when the `QueryObject` object that produced it is closed. The method `close` can be called explicitly to close a `DataSet` object, thereby releasing any external resources and making it immediately available for garbage collection. A disconnected `DataSet` must be explicitly closed by an application.

19.4.2.8 Synchronizing a `DataSet`

A `Select` annotation may be configured to return a `DataSet` object that will operate in a disconnected environment requiring the `DataSet` to be explicitly synchronized with the underlying data source. The method `DataSet.sync` must be called to propagate all `DataSet` modifications to the data source.

If a `DataSource` was used to create the `DataSet`, the sync() method may be called to update the data source. If a `Connection` was used to create the `DataSet` or the changes are being propagated to a different data source, the `sync(Connection)` method must be called.

If an attempt is made to call the `DataSet.sync` method in a connected environment, a `SQLRuntimeException` is thrown.

If a call to the `DataSet.sync` is unable to propagate all of the changes to the underlying data store, an `SQLDataSetSyncException` is thrown.

19.4.2.9 `SQLDataSetSyncException` processing

An `SQLDataSetSyncException` is thrown when the `DataSet.sync` method is unable to propagate all of the changes made to a `DataSet` to the underlying data store. The method `SQLDataSetSyncException.getDataSetResolver` can be used to obtain a `DataSetResolver` object which provides access to the rows which could not be propagated to the data store along with the underlying cause for the failure. Access to the failed rows is accomplished by examining a `ConflictingRow` object, which is obtained by iterating over an instance of a `DataSetResolver`.

A `ConflictingRow` object provides access to an individual failed row along with the cause of the failure. Each row can be updated to correct the cause of the failure and the `DataSetResolver.sync` method can then be called to propagate the changes to the backend. To propagate an individual row back to the data store, the method `DataSetResolver.syncRow` can be called. A subsequent invocation of the `DataSetResolver.sync` method will not attempt to propagate a row which had previously been updated by a call to the `DataSetResolver.syncRow` method.

CODE EXAMPLE 19-31 demonstrates how to iterate over a `DataSetResolver` object to in order to address failures that occurred during a call to the `DataSet.sync` method:

```
DataSetResolver ds = null;
try {
                // Propagate the changes to the data store
                myDataSet.sync();
} catch (SQLDataSetSyncException dse) {
                // An error occured applying the rows to the backend.
                // Obtain a DataSetResolver instance to see.
                // what failed.
                ds = dse.getDataSetResolver();
}

// Iterate through the DataSetResolver and process each
// ConflictingRow object which represents a failed row
for (ConflictingRow  row : ds) {
                // Grab the Exception and the row that failed to sync
                SQLException sqe = row.getSQLException();
                Mammal badMammal = row.getRow();

                //Update the row to resolve the conflict
                badMammal.setAge(someBetterAge);
}

// Conflicts have been resolved, so propagate the changes
// to the data store
ds.sync();
```

CODE EXAMPLE 19-31 Using a DataSetResolver object to resolve DataSet.sync failures

19.5 QueryObjectFactory Class

The `QueryObjectFactory` class is a utility class that provides access to the `QueryObjectGenerator` implementation provided by Java SE 6.

19.5.1 QueryObjectFactory Methods

The following section provides an overview of the available `QueryObjectFactory` methods.

19.5.1.1 createDefaultQueryObject method

The `createDefaultQueryObject` method is used to create a concrete implementation of a Query interface using the `QueryObjectGenerator` implementation provided by Java SE 6.

```
Connection con = DriverManager.getConnection(url, props);

MyQueries mq =
QueryObjectFactory.createDefaultQueryObject(MyQueries.class, con);
```

CODE EXAMPLE 19-32 Creating a Query object instance using the default QueryObjectGenerator implementation

19.5.1.2 createQueryObject method

The `createQueryObject` method is used to create a concrete implementation of a Query interface. This method takes two parameters, a Query interface and either a `Connection` or `DataSource` object.

An attempt is made to invoke the `createQueryObject` method on the `Connection` or `DataSource` specified as a parameter. If the `Connection` or `DataSource` object does not support the `createQueryObject` method, then the `createDefaultQueryObject` method is called to create the Query object instance.

```
Connection con = DriverManager.getConnection(url, props);

MyQueries mq = QueryObjectFactory.createQueryObject(MyQueries.class,
con);
```

CODE EXAMPLE 19-33 Creating a Query object instance

19.5.2 Managing Transactions

A `QueryObjectGenerator` implementation or a concrete implementation of
Query Interface should not modify the transaction attributes of the `Connection`
used to create the Query object instance. The transaction attributes should be
configured by the JDBC application in a non-managed environment and by the Java
EE application server in a managed environment.

Wrapper Interface

The `Wrapper` interface provides a mechanism for JDBC users to be able to access an instance of a resource which has been wrapped for architectural reasons. This mechanism helps to eliminate the need to use non-standard means to access vendor specific resources.

The following JDBC interfaces are subinterfaces of the `Wrapper` interface:

- `java.sql.Connection`
- `java.sql.DatabaseMetaData`
- `java.sql.ParameterMetaData`
- `java.sql.ResultSet`
- `java.sql.ResultSetMetaData`
- `java.sql.Statement`
- `javax.sql.Datasource`

20.1 Wrapper interface methods

The following methods are available on the Wrapper interface:

- `isWrapperFor`
- `unwrap`

20.1.1 unwrap method

The unwrap method is used to return an object that implements the specified interface allowing access to vendor-specific methods. The object that is returned may either be the object found to implement the specified interface or a proxy for that object.

If the receiver implements the specified interface then that is the object which is returned. If the receiver is a wrapper and the wrapped object implements the interface then the wrapped object or a proxy for the object is returned. Otherwise the returned object is the result of calling unwrap recursively on the wrapped object. If the receiver is not a wrapper and does not implement the specified interface, then an SQLException is thrown.

20.1.2 isWrapperFor method

The isWrapperFor method can be used to determine if the instance implements the specified interface or if the instance is a wrapper for an object that does.

If the object implements the specified interface then a value of true is returned. If this instance is a wrapper, then the method isWrapperFor is recursively called on the wrapped object. If the instance does not implement the interface and is not a wrapper, a value of false is returned.

This method should be implemented as a low-cost operation compared to calling the unwrap method. If the isWrapperFor method returns true, then calling the unwrap method with the same interface must succeed.

CODE EXAMPLE 20-1 provides an example of using the Wrapper interface methods:

```
Statement stmt = conn.createStatement();
Class clzz = Class.forName("oracle.jdbc.OracleStatement");
if(stmt.isWrapperFor(clzz)) {
    OracleStatement os = (OracleStatement)stmt.unwrap(clzz);
    os.defineColumnType(1, Types.NUMBER);
}
```

CODE EXAMPLE 20-1 Using the isWrapper and unwrap methods

Revision History

Appendix TABLE A-1 presents a summary of the revisions made to this specification.

TABLE A-1 Revision History

Revision	Dash	Date	Comments
Early Draft	01	June 2005	First early draft

TABLE A-1 Revision History

Revision	Dash	Date	Comments
Early Draft 2	01	September 2005	Added SQLXML Data type and section on StatementEventListener.
			Added Wrapper interface chapter.
			Updated sections 9.1 and 15.2.5 based on EG and external feedback.
			Updated Table B-6 to indicate that getAsciiStream, getBinaryStream and getCharacterStream support Blob, Clob and NClob
			Updated 13.2.2 to clarify how setXXXStream works when a PreparedStatement is executed more than once.
			Updated 13.2.2.4 and 13.2.2.5 to clarify the PreparedStatement method clearParameters.
			Updated 13.29 the retrieving auto generated keys should be done with autocommit mode disabled
			Updated 10.3 to indicate that JDBC URLs to not adhere to RFC 3986
			Changes to the Ease of Development features: Query annotation renamed Select; Column annotation renamed ResultColumn. Table annotation removed and the tableName annotation element added to the Select annotation. Added support for the SQLRuntimeException and SQLDataSetSyncException. The classes DataSetResolver and ConflictingRow.

TABLE A-1   Revision History

Revision	Dash	Date	Comments
Public Draft	01	December 2005	Addressed EG comments. Added SQLFeatureNotSupportedException.
			Modified 6.2 to indicate that a SQLFeatureNotSupportedException is to be thrown if a given functionality is not supported.
			Added the DatabaseMetaData methods getFunctions and getFunctionParameters.
			Removed DataSet.asRowSet method
			Clarified Section 15.4 regarding what happens when an XMLStreamWriter is not closed for a SQLXML object and what occurs if createXMLStreamWriter is called multiple times for the same SQLXML object.
			Modified the enums for GeneratedKeys in order to have support for JDBC drivers which do not allow a developer to specify the columns to be returned as auto-generated keys. This also required changes to the Ease of Development sections 19.3.2.2 and 19.3.4 .

Data Type Conversion Tables

The tables provided here describe the various mappings and conversions that drivers must support.

TABLE B-1 JDBC Types Mapped to Java Types

This table shows the conceptual correspondence between JDBC types and Java types. A programmer should write code with this mapping in mind. For example, if a value in the database is a SMALLINT, a short should be the data type used in a JDBC application.

All CallableStatement getter methods except for getObject use this mapping. The getObject methods for both the CallableStatement and ResultSet interfaces use the mapping in TABLE B-3.

TABLE B-2 Java Types Mapper to JDBC Types

This table shows the mapping a driver should use for the updater methods in the ResultSet interface and for IN parameters. PreparedStatement setter methods and RowSet setter methods use this table for mapping an IN parameter, which is a Java type, to the JDBC type that will be sent to the database. Note that the setObject methods for these two interfaces use the mapping shown in TABLE B-4.

TABLE B-3 JDBC Types Mapped to Java Object Types

ResultSet.getObject and CallableStatement.getObject use the mapping shown in this table for standard mappings.

TABLE B-4 Java Object Types Mapped to JDBC Types

PreparedStatement.setObject and RowSet.setObject use the mapping shown in this table when no parameter specifying a target JDBC type is provided.

TABLE B-5 Conversions by setObject from Java Object Types to JDBC Types

This table shows which JDBC types may be specified as the target JDBC type to the methods PreparedStatement.setObject and RowSet.setObject.

TABLE B-6 Type Conversions Supported by ResultSet getter Methods

This table shows which JDBC types may be returned by ResultSet getter methods. A bold **X** indicates the method recommended for retrieving a JDBC type. A plain x indicates for which JDBC types it is possible to use a getter method.

This table also shows the conversions used by the SQLInput reader methods, except that they use only the recommended conversions.

JDBC Type	Java Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	byte
SMALLINT	short

TABLE B-1   JDBC Types Mapped to Java Types

JDBC Type	Java Type
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
DISTINCT	mapping of underlying type
STRUCT	java.sql.Struct
REF	java.sql.Ref
DATALINK	java.net.URL
JAVA_OBJECT	underlying Java class
ROWID	java.sql.RowId
NCHAR	String
NVARCHAR	String
LONGNVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

TABLE B-1   JDBC Types Mapped to Java Types

Java Type	JDBC Type
String	CHAR, VARCHAR, LONGVARCHAR, NCHAR, NVARCHAR or LONGNVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT or BOOLEAN
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	BINARY, VARBINARY, or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.sql.Clob	CLOB
java.sql.Blob	BLOB
java.sql.Array	ARRAY
java.sql.Struct	STRUCT
java.sql.Ref	REF
java.net.URL	DATALINK
Java class	JAVA_OBJECT
java.sql.RowId	ROWID
java.sql.NClob	NCLOB
java.sql.SQLXML	SQLXML

TABLE B-2   Standard Mapping from Java Types to JDBC Types

JDBC Type	Java Object Type
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
BOOLEAN	Boolean
TINYINT	Integer
SMALLINT	Integer
INTEGER	Integer
BIGINT	Long
REAL	Float
FLOAT	Double
DOUBLE	Double
BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
DISTINCT	Object type of underlying type
CLOB	java.sql.Clob
BLOB	java.sql.Blob
ARRAY	java.sql.Array
STRUCT	java.sql.Struct or java.sql.SQLData
REF	java.sql.Ref
DATALINK	java.net.URL
JAVA_OBJECT	underlying Java class
ROWID	java.sql.RowId
NCHAR	String
NVARCHAR	String
LONGNVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

TABLE B-3   Mapping from JDBC Types to Java Object Types

Java Object Type	JDBC Type
String	CHAR, VARCHAR, LONGVARCHAR, NCHAR, NVARCHAR or LONGNVARCHAR
java.math.BigDecimal	NUMERIC
Boolean	BIT or BOOLEAN
Integer	INTEGER
Long	BIGINT
Float	REAL
Double	DOUBLE
byte[]	BINARY, VARBINARY, or LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.sql.Clob	CLOB
java.sql.Blob	BLOB
java.sql.Array	ARRAY
java.sql.Struct	STRUCT
java.sql.Ref	REF
java.net.URL	DATALINK
Java class	JAVA_OBJECT
java.sql.RowId	ROWID
java.sql.NClob	NCLOB
java.sql.SQLXML	SQLXML

TABLE B-4   Mapping from Java Object Types to JDBC Types

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	BOOLEAN	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	ARRAY	BLOB	CLOB	STRUCT	REF	DATALINK	JAVA_OBJECT	ROWID	NCHAR	NVARCHAR	LONGNVARCHAR	NCLOB	SQLXML
String	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x									x	x	x		
java.math. BigDecimal	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Boolean	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Integer	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Long	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Float	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
Double	x	x	x	x	x	x	x	x	x	x	x	x	x	x																			
byte[]															x	x	x																
java.sql. Date												x	x	x				x		x													
java.sql. Time												x	x	x					x														
java.sql. Timestamp												x	x	x				x	x	x													
java.sql.Array																					x												
java.sql.Blob																						x											
java.sql.Clob																							x										
java.sql.Struct																								x									
java.sql.Ref																									x								
java.net.URL																										x							
Java class																											x						
java.sql.RowId																												x					
java.sql.NClob																																x	
java.sql.SQLXML																																	x

TABLE B-5   Conversions Performed by `setObject` Between Java Object Types and Target JDBC Types

	TINYINT	SMALLINT	INTEGER	BIGINT	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	BIT	BOOLEAN	CHAR	VARCHAR	LONGVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP	CLOB	BLOB	ARRAY	REF	DATALINK	STRUCT	JAVA_OBJECT	ROWID	NCHAR	NVARCHAR	LONGNCHAR	NCLOB	SQLXML
getByte	X	x	x	x	x	x	x	x	x	x	x	x	x	x														x					
getShort	x	X	x	x	x	x	x	x	x	x	x	x	x	x																			
getInt	x	x	X	x	x	x	x	x	x	x	x	x	x	x																			
getLong	x	x	x	X	x	x	x	x	x	x	x	x	x	x																			
getFloat	x	x	x	x	X	x	x	x	x	x	x	x	x	x																			
getDouble	x	x	x	x	x	X	X	x	x	x	x	x	x	x																			
getBigDecimal	x	x	x	x	x	x	x	X	X	x	x	x	x	x																			
getBoolean	x	x	x	x	x	x	x	x	x	X	X	x	x	x																			
getString	x	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x	x					x				x	x	x		
getBytes															X	X	x																
getDate												x	x	x				X		x													
getTime												x	x	x					X	x													
getTimestamp												x	x	x				x	x	X													
getAsciiStream												x	x	X	x	x	x				x											x	
getBinaryStream															x	x	X					x											
getCharacterStream												x	x	X	x	x	x				x								x	x	X	x	
getClob																					X											X	
getBlob																						X											
getArray																							X										
getRef																								X									
getURL																									X								
getObject	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	X	X	x	x	x	x	x	x
getRowId																												X					
getXML																																	X

TABLE B-6 Use of `ResultSet` getter Methods to Retrieve JDBC Data Types

JDBC Annotations

JDBC Annotations are defined as part of the java.sql package and should be considered used within this namespace.

C.1   AutogeneratedKeys Annotation

The `AutoGeneratedKeys` annotation is used to indicate that a `DataSet` *data class* will be used to store any auto-generated keys that occur as the result of executing a method that is decorated with the `Update` annotation. The *data class* may contain one or more fields that represent the columns that comprise the auto generated key.

C.2   Select Annotation

The `Select` annotation is used to specify a SQL query statement that decorates a Query interface method. A `Select` annotation must include a `sql` annotation element. Additional annotation type elements may be specified to define the configuration of the returned `DataSet` object.

C.3 Update Annoation

The `Update` annotation is used to decorate a Query interface method with a a SQL statement that returns an update count. An `Update` annotation must include a `SQL` annotation type element.

C.4 ResultColumn Annotation

Annotation that allows a specified field of a DataSet of type <T> to be mapped directly to a specified column in the result set returned by invoking a method that is decorated by a Select annotation.

Related Documents

This specification makes reference to the following documents.

Data Management: SQL Call Level Interface (X/Open SQL CLI) Available at `http://www.opengroup.org`.

Distributed Transaction Processes: The XA Specification (X/Open CAE) Available at `http://www.opengroup.org`.

JDBC RowSet 1.0.1 Implementations Specification. Copyright 2004, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdbc`.

JDBC 3.0 Specification (JDBC 3.0). Copyright 2001, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdbc`.

JDBC 2.1 API (JDBC 2.1). Copyright 1998, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdbc`.

JDBC 2.0 Standard Extension API (JDBC extension specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdbc`.

JDBC 1.22 API (JDBC 1.22). Copyright 1998, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jdbc`.

JavaBeans 1.01 Specification (JavaBeans specification). Copyright 1996, 1997, Sun Microsystems, Inc. Available at `http://java.sun.com/beans`.

Java Transaction API, Version 1.0.1 (JTA Specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jta`.

Java Naming and Directory Interface 1.2 Specification (JNDI specification). Copyright 1998, 1999, Sun Microsystems, Inc. Available at `http://java.sun.com/products/jndi`.

Enterprise Java Beans Specification, Version 2.1 (EJB). Copyright 2003, Sun Microsystems, Inc. Available at `http://java.sun.com/products/ejb`.

J2EE Connector Architecture Specification, Version 1.5 Copyright 2003, Sun Microsystems, Inc. Available at http://java.sun.com/j2ee.

The following documents are collectively refered to as SQL:2003:

ISO/IEC 9075-1:2003, Information technology - Database languages - SQL - Part 1: Framework (SQL/Framework).

ISO/IEC 9075-2:2003, Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation).

ISO/IEC 9075-3:2003, Information technology - Database languages - SQL - Part 3: Call-Level Interface (SQL/CLI).

ISO/IEC 9075-4:2003, Information technology - Database languages - SQL - Part 4: Persistent Stored Modules (SQL/PSM).

ISO/IEC 9075-5:2003, Information technology - Database languages - SQL - Part 5: Host Language Bindings (SQL/Bindings).

ISO/IEC 9075-9:2003 Information technology - Database languages - SQL - Part 9: Management of External Data (SQL/MED)

ISO/IEC 9075-14:2003 Information technology - Database languages - SQL - Part 14: XML-Related Specifications (SQL/XML)

The following document is a reference for SQLJ:

ISO/IEC 9075-10:2003, Information technology - Database Languages SQL - Part 10: Object Language Bindings (SQL/OLB)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
650 960-1300

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 650 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
650 960-1300
Intercontinental Sales: 650 688-9000

Scalar Functions

The JDBC API supports escape syntax for numeric, string, time, date, system, and conversion functions on scalar values. These scalar functions may be used in SQL strings as described in Section 13.4.1 "Scalar Functions" on page 13-111. The Open Group CLI specification provides more information on the semantics of the scalar functions. The scalar functions are listed below for reference.

If a DBMS supports a scalar function, the driver should also. Because scalar functions are supported by different DBMSs with slightly different syntax, it is the driver's job either to map them into the appropriate syntax or to implement the functions directly in the driver.

A user should be able to find out which functions are supported by calling metadata methods. For example, the method DatabaseMetaData.getNumericFunctions returns a comma separated list of the Open Group CLI names of the numeric functions supported. Similarly, the method DatabaseMetaData.getStringFunctions returns a list of string functions supported, and so on.

The scalar functions are listed by category:

B.1 NUMERIC FUNCTIONS

Function Name	Function Returns
ABS(number)	Absolute value of number
ACOS(float)	Arccosine, in radians, of float
ASIN(float)	Arcsine, in radians, of float
ATAN(float)	Arctangent, in radians, of float
ATAN2(float1, float2)	Arctangent, in radians, of float2 / float1
CEILING(number)	Smallest integer >= number
COS(float)	Cosine of float radians
COT(float)	Cotangent of float radians

DEGREES(number)	Degrees in number radians
EXP(float)	Exponential function of float
FLOOR(number)	Largest integer <= number
LOG(float)	Base e logarithm of float
LOG10(float)	Base 10 logarithm of float
MOD(integer1, integer2)	Remainder for integer1 / integer2
PI()	The constant pi
POWER(number, power)	number raised to (integer) power
RADIANS(number)	Radians in number degrees
RAND(integer)	Random floating point for seed integer
ROUND(number, places)	number rounded to places places
SIGN(number)	-1 to indicate number is < 0;
	0 to indicate number is = 0;
	1 to indicate number is > 0
SIN(float)	Sine of float radians
SQRT(float)	Square root of float
TAN(float)	Tangent of float radians
TRUNCATE(number, places)	number truncated to places places

B.2   STRING FUNCTIONS

Function Name	**Function Returns**
ASCII(string)	Integer representing the ASCII code value of the leftmost character in string
CHAR(code)	Character with ASCII code value code, where code is between 0 and 255
CONCAT(string1, string2)	Character string formed by appending string2 to string1; if a string is null, the result is DBMS-dependent
DIFFERENCE(string1, string2)	Integer indicating the difference between the values returned by the function SOUNDEX for string1 and string2
INSERT(string1, start, length, string2)	A character string formed by deleting length characters from string1 beginning at start, and inserting string2 into string1 at start
LCASE(string)	Converts all uppercase characters in string to lowercase
LEFT(string, count)	The count leftmost characters from string
LENGTH(string)	Number of characters in string, excluding trailing blanks
LOCATE(string1, string2[, start])	Position in string2 of the first occurrence of string1, searching from the beginning of string2; if start is specified, the search begins from position start. 0 is returned if string2 does not contain string1. Position 1 is the first character in string2.
LTRIM(string)	Characters of string with leading blank spaces removed
REPEAT(string, count)	A character string formed by repeating string count times
REPLACE(string1, string2, string3)	Replaces all occurrences of string2 in string1 with string3
RIGHT(string, count)	The count rightmost characters in string
RTRIM(string)	The characters of string with no trailing blanks
SOUNDEX(string)	A character string, which is data source-dependent, representing the sound of the words in string; this could be a four-digit SOUNDEX code, a phonetic representation of each word, etc.

SPACE(count)	A character string consisting of count spaces
SUBSTRING(string, start, length)	A character string formed by extracting length characters from string beginning at start
UCASE(string)	Converts all lowercase characters in string to uppercase

B.3   TIME and DATE FUNCTIONS

Function Name	Function Returns
CURDATE()	The current date as a date value
CURTIME()	The current local time as a time value
DAYNAME(date)	A character string representing the day component of date; the name for the day is specific to the data source
DAYOFMONTH(date)	An integer from 1 to 31 representing the day of the month in date
DAYOFWEEK(date)	An integer from 1 to 7 representing the day of the week in date; 1 represents Sunday
DAYOFYEAR(date)	An integer from 1 to 366 representing the day of the year in date
HOUR(time)	An integer from 0 to 23 representing the hour component of time
MINUTE(time)	An integer from 0 to 59 representing the minute component of time
MONTH(date)	An integer from 1 to 12 representing the month component of date
MONTHNAME(date)	A character string representing the month component of date; the name for the month is specific to the data source
NOW()	A timestamp value representing the current date and time
QUARTER(date)	An integer from 1 to 4 representing the quarter in date; 1 represents January 1 through March 31
SECOND(time)	An integer from 0 to 59 representing the second component of time
TIMESTAMPADD(interval, count, timestamp)	A timestamp calculated by adding count number of interval(s) to timestamp; interval may be one of the following: SQL_TSI_FRAC_SECOND, SQL_TSI_SECOND, SQL_TSI_MINUTE, SQL_TSI_HOUR, SQL_TSI_DAY, SQL_TSI_WEEK, SQL_TSI_MONTH, SQL_TSI_QUARTER, or SQL_TSI_YEAR
TIMESTAMPDIFF(interval, timestamp1, timestamp2)	An integer representing the number of interval by which timestamp2 is greater than timestamp1; interval may be one of the following: SQL_TSI_FRAC_SECOND, SQL_TSI_SECOND, SQL_TSI_MINUTE, SQL_TSI_HOUR, SQL_TSI_DAY, SQL_TSI_WEEK, SQL_TSI_MONTH, SQL_TSI_QUARTER, or SQL_TSI_YEAR
WEEK(date)	An integer from 1 to 53 representing the week of the year in date
YEAR(date)	An integer representing the year component of date

B.4   SYSTEM FUNCTIONS

Function Name	Function Returns
DATABASE()	Name of the database
IFNULL(expression, value)	value if expression is null;

| | expression if expression is not null |
| USER() | User name in the DBMS |

B.5 CONVERSION FUNCTIONS

Function Name	**Function Returns**
CONVERT(value, SQLtype)	value converted to SQLtype where SQLtype may be one of the following SQL types:
	SQL_BIGINT, SQL_BINARY, SQL_BIT, SQL_BLOB, SQL_BOOLEAN, SQL_CHAR, SQL_CLOB, SQL_DATE, SQL_DECIMAL, SQL_DATALINK, SQL_DOUBLE, SQL_FLOAT, SQL_INTEGER, SQL_LONGVARBINARY, SQL_LONGNVARCHAR, SQL_LONGVARCHAR, SQL_NCHAR, SQL_NCLOB, SQL_NUMERIC, SQL_NVARCHAR, SQL_REAL, SQL_ROWID, SQL_SQLXML, SQL_SMALLINT, SQL_TIME, SQL_TIMESTAMP, SQL_TINYINT, SQL_VARBINARY, or SQL_VARCHAR