Java Tutorials

Updated for Java SE 8

ORACLE

Java Tutorials

The Java Tutorials are a one stop shop for learning the Java language. The tutorials are practical guides for programmers who want to use the Java programming language to create applets and applications. They also serve as a reference for the experienced Java programmer.

Groups of related lessons are organized into "trails"; for example, the "Getting Started" trail, or the "Learning the Java Language" trail. This ebook contains a single trail. You can download other trails via the link on the Java Tutorials home page.

To read an ebook file that you have downloaded, transfer the document to your device.

We would love to hear what you think of this ebook, including any problems that you find. Please send us your feedback.

Legal Notices

This tutorial is a guide to developing applications for the Java Platform, Standard Edition and contains documentation (Tutorial) and sample code. The sample code made available with this Tutorial is licensed separately to you by Oracle under the Berkeley license. If you download any such sample code, you agree to the terms of the Berkeley license.

This Tutorial is provided to you by Oracle under the following license terms containing restrictions on use and disclosure and is protected by intellectual property laws. Oracle grants to you a limited, non-exclusive license to use this Tutorial for information purposes only, as an aid to learning about the Java SE platform. Except as expressly permitted in these license terms, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means this Tutorial. Reverse engineering, disassembly, or decompilation of this Tutorial is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If the Tutorial is licensed on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This Tutorial is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this Tutorial in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use.

THE TUTORIAL IS PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND. ORACLE FURTHER DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT.

IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, PUNITIVE OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA OR DATA USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF ORACLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. ORACLES ENTIRE LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED ONE THOUSAND DOLLARS (U.S. $1,000).

No Technical Support

Oracles technical support organization will not provide technical support, phone support, or updates to you.

The sample code and Tutorial may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Supported Platforms

The `mobi` file format works best on the following devices:

- Kindle Fire
- Kindle DX

The `ePub` file format works best on the following devices:

- iPad
- Nook
- Other eReaders that support the `ePub` format.

For best results when viewing preformatted code blocks, adjust the landscape/portrait orientation and font size of your device to enable the maximum possible viewing area.

Trail: JDBC(TM) Database Access

The JDBC API was designed to keep simple things simple. This means that the JDBC makes everyday database tasks easy. This trail walks you through examples of using JDBC to execute common SQL statements, and perform other objectives common to database applications.
This trail is divided into these lessons:

 **JDBC Introduction** Lists JDBC features, describes JDBC Architecture and reviews SQL commands and Relational Database concepts.

 **JDBC Basics** covers the JDBC API, which is included in the Java SE 6 release.

By the end of the first lesson, you will know how to use the basic JDBC API to create tables, insert values into them, query the tables, retrieve the results of the queries, and update the tables. In this process, you will learn how to use simple statements and prepared statements, and you will see an example of a stored procedure. You will also learn how to perform transactions and how to catch exceptions and warnings.

- Legal Notices
- Supported Platforms

Trail: JDBC(TM) Database Access: Table of Contents

Lesson: JDBC Introduction

The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

JDBC helps you to write Java applications that manage these three programming activities:

1. Connect to a data source, like a database
2. Send queries and update statements to the database
3. Retrieve and process the results received from the database in answer to your query

The following simple code fragment gives a simple example of these three steps:

```
public void connectToAndQueryDatabase(String username, String password) {

    Connection con = DriverManager.getConnection(
                       "jdbc:myDriver:myDatabase",
                       username,
                       password);

    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

    while (rs.next()) {
        int x = rs.getInt("a");
        String s = rs.getString("b");
        float f = rs.getFloat("c");
    }
}
```

This short code fragment instantiates a `DriverManager` object to connect to a database driver and log into the database, instantiates a `Statement` object that carries your SQL language query to the database; instantiates a `ResultSet` object that retrieves the results of your query, and executes a simple `while` loop, which retrieves and displays those results. It's that simple.

JDBC Product Components

JDBC includes four components:

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

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.

2. JDBC Driver Manager The JDBC `DriverManager` class defines objects which can connect Java applications to a JDBC driver. `DriverManager` has traditionally been the backbone of the JDBC architecture. It is quite small and simple.

The Standard Extension packages `javax.naming` and `javax.sql` let you use a `DataSource` object registered with a *Java Naming and Directory Interface* (JNDI) naming service to establish a connection with a data source. You can use either connecting mechanism, but using a `DataSource` object is recommended whenever possible.

3. JDBC Test Suite The JDBC driver test suite helps you to determine that JDBC drivers will run your program. These tests are not comprehensive or exhaustive, but they do exercise many of the important features in the JDBC API.

4. JDBC-ODBC Bridge The Java Software bridge provides JDBC access via ODBC drivers. Note that you need to load ODBC binary code onto each client machine that uses this driver. As a result, the ODBC driver is most appropriate on a corporate network where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

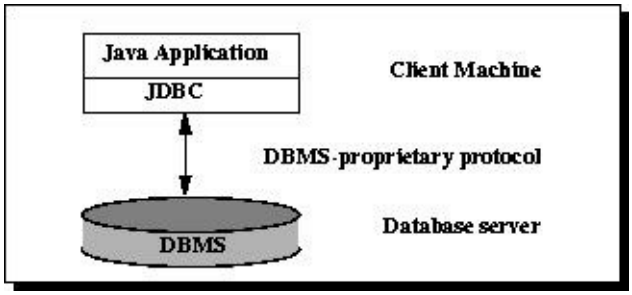
This Trail uses the first two of these these four JDBC components to connect to a database and then build a java program that uses SQL commands to communicate with a test Relational Database. The last two components are used in specialized environments to test web applications, or to communicate with ODBC-aware DBMSs.

JDBC Architecture

Two-tier and Three-tier Processing Models

The JDBC API supports both two-tier and three-tier processing models for database access.

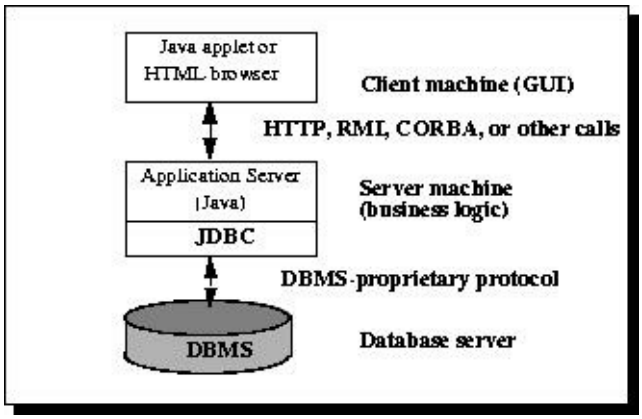
Figure 1: Two-tier Architecture for Data Access.



In the two-tier model, a Java applet or application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 2: Three-tier Architecture for Data Access.



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java

middle tier.

A Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

The `Employees` table illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee.

`Employees` Table

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	Axel	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use `First_Name` and `Last_Name` because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Washington." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Yamaguchi gets a job at this company and the

primary key is `First_Name`, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make the primary key useless as a way of identifying just one row. Note that although using `First_Name` and `Last_Name` is a unique composite key for this example, it might not be unique in a larger database. Note also that the `Employee` table assumes that there can be only one car per employee.

`SELECT` Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the `SELECT` statement.

A `SELECT` statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A `SELECT` statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the `Car_Number` column) follows. The first name and last name are printed for each row that satisfies the requirement because the `SELECT` statement (the first line) specifies the columns `First_Name` and `Last_Name`. The `FROM` clause (the second line) gives the table from which the columns will be selected.

FIRST_NAME	LAST_NAME
Axel	Washington
Florence	Wojokowski

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no `WHERE` clause). Note that `SELECT *` means "SELECT all columns."

```
SELECT *
FROM Employees
```

`WHERE` Clauses

The `WHERE` clause in a `SELECT` statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword `LIKE` is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (`%`) at the end of 'Washington', which signifies that any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in `LIKE` clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Batman', 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a `WHERE` clause that uses the equal sign (=) to compare numbers. It

selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

`WHERE` clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated `WHERE` clauses, but the following code fragment has a `WHERE` clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

A special type of `WHERE` clause involves a join, which is explained in the next section.

Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the make, model, and year of car. This information is stored in another table, `Cars`:

`Cars` Table

Car_Number	Make	Model	Year
5	Honda	Civic DX	1996
12	Toyota	Corolla	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is `Car_Number`, which is the primary key for the table `Cars` and the foreign key in the table Employees. If the 1996 Honda Civic were wrecked and deleted from the `Cars` table, then `Car_Number` 5 would also have to be removed from the Employees table in order to maintain what is called referential integrity. Otherwise, the foreign key column (`Car_Number`) in the `Employees` table would contain an entry that did not refer to anything in `Cars`. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the `Car_Number` column in the table `Employees` because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the make, model, and year of those cars. Note that the `FROM` clause lists both Employees and Cars because the requested data is contained in both tables. Using the table name and a dot (.) before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name,
    Cars.Make, Cars.Model, Cars.Year
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

FIRST_NAME	LAST_NAME	MAKE	MODEL	YEAR
Axel	Washington	Honda	Civic DX	1996
Florence	Wojokowski	Toyota	Corolla	1999

Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.
A list of the more common DML commands follows:

- `SELECT` used to query and display data from a database. The `SELECT` statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are `SELECT` statements.
- `INSERT` adds new rows to a table. `INSERT` is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- `DELETE` removes a specified row or set of rows from a table
- `UPDATE` changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- `CREATE TABLE` creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. `CREATE TABLE` is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- `DROP TABLE` deletes all rows and removes the table definition from the database. 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.
- `ALTER TABLE` adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

Result Sets and Cursors

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.
Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

Transactions

When one user is accessing data in a database, another user may be accessing the same data at the

same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

Stored Procedures

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program, that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database.

The following code is an example of how to create a very simple stored procedure using the Java programming language. Note that the stored procedure is just a static Java method that contains normal JDBC code. It accepts two input parameters and uses them to change an employee's car number.

Do not worry if you do not understand the example at this point. The code example below is presented only to illustrate what a stored procedure looks like. You will learn how to write the code in this example in the tutorials that follow.

```
import java.sql.*;

public class UpdateCar {

    public static void UpdateCarNum(int carNo, int empNo)
        throws SQLException {

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = DriverManager.getConnection(
                      "jdbc:default:connection");

            pstmt = con.prepareStatement(
                        "UPDATE EMPLOYEES " +
                        "SET CAR_NUMBER = ? " +
                        "WHERE EMPLOYEE_NUMBER = ?");

            pstmt.setInt(1, carNo);
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
        }
        finally {
            if (pstmt != null) pstmt.close();
        }
    }
}
```

Metadata

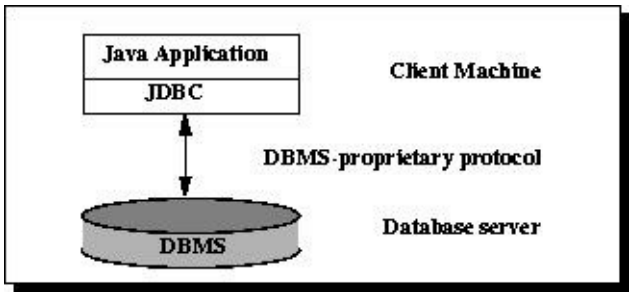
Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface `DatabaseMetaData`, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata.
In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.

Note: See online version of topics in this ebook to download complete source code.

JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access.

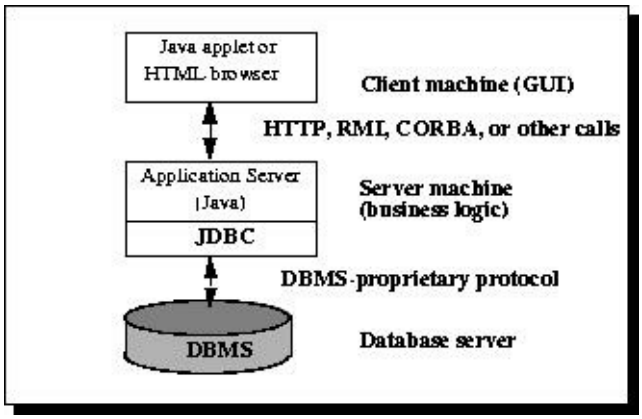
Figure 1: Two-tier Architecture for Data Access.



In the two-tier model, a Java application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration, with the user's machine as the client, and the machine housing the data source as the server. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user. MIS directors find the three-tier model very attractive because the middle tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that it simplifies the deployment of applications. Finally, in many cases, the three-tier architecture can provide performance advantages.

Figure 2: Three-tier Architecture for Data Access.



Until recently, the middle tier has often been written in languages such as C or C++, which offer fast performance. However, with the introduction of optimizing compilers that translate Java bytecode into efficient machine-specific code and technologies such as Enterprise JavaBeans, the Java platform is fast becoming the standard platform for middle-tier development. This is a big plus, making it possible to take advantage of Java's robustness, multithreading, and security features.

With enterprises increasingly using the Java programming language for writing server code, the JDBC API is being used more and more in the middle tier of a three-tier architecture. Some of the features that make JDBC a server technology are its support for connection pooling, distributed transactions, and disconnected rowsets. The JDBC API is also what allows access to a data source from a Java middle tier.

A Relational Database Overview

A database is a means of storing information in such a way that information can be retrieved from it. In simplest terms, a relational database is one that presents information in tables with rows and columns. A table is referred to as a relation in the sense that it is a collection of objects of the same type (rows). Data in a table can be related according to common keys or concepts, and the ability to retrieve related data from a table is the basis for the term relational database. A Database Management System (DBMS) handles the way data is stored, maintained, and retrieved. In the case of a relational database, a Relational Database Management System (RDBMS) performs these tasks. DBMS as used in this book is a general term that includes RDBMS.

Integrity Rules

Relational tables follow certain integrity rules to ensure that the data they contain stay accurate and are always accessible. First, the rows in a relational table should all be distinct. If there are duplicate rows, there can be problems resolving which of two possible selections is the correct one. For most DBMSs, the user can specify that duplicate rows are not allowed, and if that is done, the DBMS will prevent the addition of any rows that duplicate an existing row.

A second integrity rule of the traditional relational model is that column values must not be repeating groups or arrays. A third aspect of data integrity involves the concept of a null value. A database takes care of situations where data may not be available by using a null value to indicate that a value is missing. It does not equate to a blank or zero. A blank is considered equal to another blank, a zero is equal to another zero, but two null values are not considered equal.

When each row in a table is different, it is possible to use one or more columns to identify a particular row. This unique column or group of columns is called a primary key. Any column that is part of a primary key cannot be null; if it were, the primary key containing it would no longer be a complete identifier. This rule is referred to as entity integrity.

The `Employees` table illustrates some of these relational database concepts. It has five columns and six rows, with each row representing a different employee.

Employees Table

Employee_Number	First_name	Last_Name	Date_of_Birth	Car_Number
10001	John	Washington	28-Aug-43	5
10083	Arvid	Sharma	24-Nov-54	null
10120	Jonas	Ginsberg	01-Jan-69	null
10005	Florence	Wojokowski	04-Jul-71	12
10099	Sean	Washington	21-Sep-66	null
10035	Elizabeth	Yamaguchi	24-Dec-59	null

The primary key for this table would generally be the employee number because each one is guaranteed to be different. (A number is also more efficient than a string for making comparisons.) It would also be possible to use `First_Name` and `Last_Name` because the combination of the two also identifies just one row in our sample database. Using the last name alone would not work because there are two employees with the last name of "Washington." In this particular case the first names are all different, so one could conceivably use that column as a primary key, but it is best to avoid using a column where duplicates could occur. If Elizabeth Jones gets a job at this company and the primary key is `First_Name`, the RDBMS will not allow her name to be added (if it has been specified that no duplicates are permitted). Because there is already an Elizabeth in the table, adding a second one would make the primary key useless as a way of identifying just one row. Note that although using `First_Name` and `Last_Name` is a unique composite key for this example, it might not be unique in a larger database. Note also that the `Employees` table assumes that there can be only one car per employee.

SELECT Statements

SQL is a language designed to be used with relational databases. There is a set of basic SQL commands that is considered standard and is used by all RDBMSs. For example, all RDBMSs use the `SELECT` statement.

A `SELECT` statement, also called a query, is used to get information from a table. It specifies one or more column headings, one or more tables from which to select, and some criteria for selection. The RDBMS returns rows of the column entries that satisfy the stated requirements. A `SELECT` statement such as the following will fetch the first and last names of employees who have company cars:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number IS NOT NULL
```

The result set (the set of rows that satisfy the requirement of not having null in the `Car_Number` column) follows. The first name and last name are printed for each row that satisfies the requirement because the `SELECT` statement (the first line) specifies the columns `First_Name` and `Last_Name`. The `FROM` clause (the second line) gives the table from which the columns will be selected.

FIRST_NAME	LAST_NAME
John	Washington
Florence	Wojokowski

The following code produces a result set that includes the whole table because it asks for all of the columns in the table Employees with no restrictions (no `WHERE` clause). Note that `SELECT *` means "`SELECT` all columns."

```
SELECT *
FROM Employees
```

WHERE Clauses

The WHERE clause in a SELECT statement provides the criteria for selecting values. For example, in the following code fragment, values will be selected only if they occur in a row in which the column Last_Name begins with the string 'Washington'.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Last_Name LIKE 'Washington%'
```

The keyword LIKE is used to compare strings, and it offers the feature that patterns containing wildcards can be used. For example, in the code fragment above, there is a percent sign (%) at the end of 'Washington', which signifies that any value containing the string 'Washington' plus zero or more additional characters will satisfy this selection criterion. So 'Washington' or 'Washingtonian' would be matches, but 'Washing' would not be. The other wildcard used in LIKE clauses is an underbar (_), which stands for any one character. For example,

```
WHERE Last_Name LIKE 'Ba_man'
```

would match 'Barman', 'Badman', 'Balman', 'Bagman', 'Bamman', and so on.

The code fragment below has a WHERE clause that uses the equal sign (=) to compare numbers. It selects the first and last name of the employee who is assigned car 12.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Car_Number = 12
```

The next code fragment selects the first and last names of employees whose employee number is greater than 10005:

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number > 10005
```

WHERE clauses can get rather elaborate, with multiple conditions and, in some DBMSs, nested conditions. This overview will not cover complicated WHERE clauses, but the following code fragment has a WHERE clause with two conditions; this query selects the first and last names of employees whose employee number is less than 10100 and who do not have a company car.

```
SELECT First_Name, Last_Name
FROM Employees
WHERE Employee_Number < 10100 and Car_Number IS NULL
```

A special type of WHERE clause involves a join, which is explained in the next section.

Joins

A distinguishing feature of relational databases is that it is possible to get data from more than one table in what is called a join. Suppose that after retrieving the names of employees who have company cars, one wanted to find out who has which car, including the license plate number, mileage, and year of car. This information is stored in another table, `Cars`:

Cars Table

Car_Number	License_Plate	Mileage	Year
5	ABC123	5000	1996
12	DEF123	7500	1999

There must be one column that appears in both tables in order to relate them to each other. This column, which must be the primary key in one table, is called the foreign key in the other table. In this case, the column that appears in two tables is `Car_Number`, which is the primary key for the table `Cars` and the foreign key in the table `Employees`. If the 1996 car with license plate number ABC123 were wrecked and deleted from the `Cars` table, then `Car_Number` 5 would also have to be removed from the `Employees` table in order to maintain what is called referential integrity. Otherwise, the foreign key column (`Car_Number`) in the `Employees` table would contain an entry that did not refer to anything in the `Cars` table. A foreign key must either be null or equal to an existing primary key value of the table to which it refers. This is different from a primary key, which may not be null. There are several null values in the `Car_Number` column in the table `Employees` because it is possible for an employee not to have a company car.

The following code asks for the first and last names of employees who have company cars and for the license plate number, mileage, and year of those cars. Note that the `FROM` clause lists both the `Employees` and `Cars` tables because the requested data is contained in both tables. Using the table name and a dot (`.`) before the column name indicates which table contains the column.

```
SELECT Employees.First_Name, Employees.Last_Name,
    Cars.License_Plate, Cars.Mileage, Cars.Year
FROM Employees, Cars
WHERE Employees.Car_Number = Cars.Car_Number
```

This returns a result set that will look similar to the following:

FIRST_NAME	LAST_NAME	LICENSE_PLATE	MILEAGE	YEAR
John	Washington	ABC123	5000	1996
Florence	Wojokowski	DEF123	7500	1999

Common SQL Commands

SQL commands are divided into categories, the two main ones being Data Manipulation Language (DML) commands and Data Definition Language (DDL) commands. DML commands deal with data, either retrieving it or modifying it to keep it up-to-date. DDL commands create or change tables and other database objects such as views and indexes.
A list of the more common DML commands follows:

- `SELECT` used to query and display data from a database. The `SELECT` statement specifies which columns to include in the result set. The vast majority of the SQL commands used in applications are `SELECT` statements.
- `INSERT` adds new rows to a table. `INSERT` is used to populate a newly created table or to add a new row (or rows) to an already-existing table.
- `DELETE` removes a specified row or set of rows from a table
- `UPDATE` changes an existing value in a column or group of columns in a table

The more common DDL commands follow:

- `CREATE TABLE` creates a table with the column names the user provides. The user also needs to specify a type for the data in each column. Data types vary from one RDBMS to another, so a user might need to use metadata to establish the data types used by a particular database. `CREATE TABLE` is normally used less often than the data manipulation commands because a table is created only once, whereas adding or deleting rows or changing individual values generally occurs more frequently.
- `DROP TABLE` deletes all rows and removes the table definition from the database. 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.
- `ALTER TABLE` adds or removes a column from a table. It also adds or drops table constraints and alters column attributes

Result Sets and Cursors

The rows that satisfy the conditions of a query are called the result set. The number of rows returned in a result set can be zero, one, or many. A user can access the data in a result set one row at a time, and a cursor provides the means to do that. A cursor can be thought of as a pointer into a file that contains the rows of the result set, and that pointer has the ability to keep track of which row is currently being accessed. A cursor allows a user to process each row of a result set from top to bottom and consequently may be used for iterative processing. Most DBMSs create a cursor automatically when a result set is generated.

Earlier JDBC API versions added new capabilities for a result set's cursor, allowing it to move both forward and backward and also allowing it to move to a specified row or to a row whose position is relative to another row.

See Retrieving and Modifying Values from Result Sets for more information.

Transactions

When one user is accessing data in a database, another user may be accessing the same data at the same time. If, for instance, the first user is updating some columns in a table at the same time the second user is selecting columns from that same table, it is possible for the second user to get partly old data and partly updated data. For this reason, DBMSs use transactions to maintain data in a consistent state (data consistency) while allowing more than one user to access a database at the same time (data concurrency).

A transaction is a set of one or more SQL statements that make up a logical unit of work. A transaction ends with either a commit or a rollback, depending on whether there are any problems with data consistency or data concurrency. The commit statement makes permanent the changes resulting from the SQL statements in the transaction, and the rollback statement undoes all changes resulting from the SQL statements in the transaction.

A lock is a mechanism that prohibits two transactions from manipulating the same data at the same time. For example, a table lock prevents a table from being dropped if there is an uncommitted transaction on that table. In some DBMSs, a table lock also locks all of the rows in a table. A row lock prevents two transactions from modifying the same row, or it prevents one transaction from selecting a row while another transaction is still modifying it.

See Using Transactions for more information.

Stored Procedures

A stored procedure is a group of SQL statements that can be called by name. In other words, it is executable code, a mini-program, that performs a particular task that can be invoked the same way one can call a function or method. Traditionally, stored procedures have been written in a DBMS-specific programming language. The latest generation of database products allows stored procedures to be written using the Java programming language and the JDBC API. Stored procedures written in the Java programming language are bytecode portable between DBMSs. Once a stored procedure is written, it can be used and reused because a DBMS that supports stored procedures will, as its name implies, store it in the database. See Using Stored Procedures for information about writing stored procedures.

Metadata

Databases store user data, and they also store information about the database itself. Most DBMSs have a set of system tables, which list tables in the database, column names in each table, primary keys, foreign keys, stored procedures, and so forth. Each DBMS has its own functions for getting information about table layouts and database features. JDBC provides the interface `DatabaseMetaData`, which a driver writer must implement so that its methods return information about the driver and/or DBMS for which the driver is written. For example, a large number of methods return whether or not the driver supports a particular functionality. This interface gives users and tools a standardized way to get metadata. In general, developers writing tools and drivers are the ones most likely to be concerned with metadata.

Lesson: JDBC Basics

In this lesson you will learn the basics of the JDBC API.

- Getting Started sets up a basic database development environment and shows you how to compile and run the JDBC tutorial samples.
- Processing SQL Statements with JDBC outlines the steps required to process any SQL statement. The pages that follow describe these steps in more detail:
  - Establishing a Connection connects you to your database.
  - Connecting with DataSource Objects shows you how to connect to your database with `DataSource` objects, the preferred way of getting a connection to a data source.
  - Handling SQLExceptions shows you how to handle exceptions caused by database errors.
  - Setting Up Tables describes all the database tables used in the JDBC tutorial samples and how to create and populate tables with JDBC API and SQL scripts.
  - Retrieving and Modifying Values from Result Sets develop the process of configuring your database, sending queries, and retrieving data from your database.
  - Using Prepared Statements describes a more flexible way to create database queries.
  - Using Transactions shows you how to control when a database query is actually executed.
- Using RowSet Objects introduces you to `RowSet` objects; these are objects that hold tabular data in a way that make it more flexible and easier to use than result sets. The pages that follow describe the different kinds of `RowSet` objects available:
  - Using JdbcRowSet Objects
  - Using CachedRowSetObjets
  - Using JoinRowSet Objects
  - Using FilteredRowSet Objects
  - Using WebRowSet Objects
- Using Advanced Data Types introduces you to other data types; the pages that follow describe these data types in further detail:
  - Using Large Objects
  - Using SQLXML Objects
  - Using Array Objects
  - Using DISTINCT Data Type
  - Using Structured Objects
  - Using Customized Type Mappings
  - Using Datalink Objects
  - Using RowId Objects
- Using Stored Procedures shows you how to create and use a stored procedure, which is a group of SQL statements that can be called like a Java method with variable input and output parameters.
- Using JDBC with GUI API demonstrates how to integrate JDBC with the Swing API.

Note: See online version of topics in this ebook to download complete source code.

Getting Started

The sample code that comes with this tutorial creates a database that is used by a proprietor of a small coffee house called The Coffee Break, where coffee beans are sold by the pound and brewed coffee is sold by the cup.
The following steps configure a JDBC development environment with which you can compile and run the tutorial samples:

1. Install the latest version of the Java SE SDK on your computer
2. Install your database management system (DBMS) if needed
3. Install a JDBC driver from the vendor of your database
4. Install Apache Ant
5. Install Apache Xalan
6. Download the sample code
7. Modify the `build.xml` file
8. Modify the tutorial properties file
9. Compile and package the samples
10. Create databases, tables, and populate tables
11. Run the samples

Install the latest version of the Java SE SDK on your computer

Install the latest version of the Java SE SDK on your computer.
Ensure that the full directory path of the Java SE SDK `bin` directory is in your `PATH` environment variable so that you can run the Java compiler and the Java application launcher from any directory.

Install your database management system (DBMS) if needed

You may use Java DB, which comes with the latest version of Java SE SDK. This tutorial has been tested for the following DBMS:

- Java DB
- MySQL

Note that if you are using another DBMS, you might have to alter the code of the tutorial samples.

Install a JDBC driver from the vendor of your database

If you are using Java DB, it already comes with a JDBC driver. If you are using MySQL, install the latest version of Connector/J.
Contact the vendor of your database to obtain a JDBC driver for your DBMS.
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 (Open Database Connectivity). Drivers of this type are generally dependent on a native library, which limits their portability. The JDBC-ODBC Bridge is an example of a Type 1 driver.
  Note: The JDBC-ODBC Bridge should be considered a transitional solution. It is not supported by Oracle. Consider using this only if your DBMS does not offer a Java-only JDBC 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. Oracle's OCI (Oracle Call Interface) client-side driver is an example of a Type 2 driver.
- **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.

Check which driver types comes with your DBMS. Java DB comes with two Type 4 drivers, an Embedded driver and a Network Client Driver. MySQL Connector/J is a Type 4 driver.
Installing a JDBC driver generally consists of copying the driver to your computer, then adding the location of it to your class path. In addition, many JDBC drivers other than Type 4 drivers require you to install a client-side API. No other special configuration is usually needed.

Install Apache Ant

These steps use Apache Ant, a Java-based tool, to build, compile, and run the JDBC tutorial samples. Go to the following link to download Apache Ant:
http://ant.apache.org/
Ensure that the Apache Ant executable file is in your `PATH` environment variable so that you can run it from any directory.

Install Apache Xalan

The sample `RSSFeedsTable.java`, which is described in Using SQLXML Objects, requires Apache Xalan if your DBMS is Java DB. The sample uses Apache Xalan-Java. Go to the following link to download it:
http://xml.apache.org/xalan-j/

Download the sample code

The sample code, JDBCTutorial.zip, consists of the following files:

- properties
  - javadb-build-properties.xml
  - javadb-sample-properties.xml
  - mysql-build-properties.xml
  - mysql-sample-properties.xml
- sql
  - javadb
    - create-procedures.sql
    - create-tables.sql
    - drop-tables.sql
    - populate-tables.sql
  - mysql
    - create-procedures.sql
    - create-tables.sql
    - drop-tables.sql
    - populate-tables.sql
- src/com/oracle/tutorial/jdbc
  - CachedRowSetSample.java
  - CityFilter.java
  - ClobSample.java
  - CoffeesFrame.java
  - CoffeesTable.java
  - CoffeesTableModel.java
  - DatalinkSample.java
  - ExampleRowSetListener.java
  - FilteredRowSetSample.java
  - JdbcRowSetSample.java
  - JDBCTutorialUtilities.java
  - JoinSample.java
  - ProductInformationTable.java
  - RSSFeedsTable.java
  - StateFilter.java
  - StoredProcedureJavaDBSample.java
  - StoredProcedureMySQLSample.java
  - SuppliersTable.java
  - WebRowSetSample.java
- txt
  - colombian-description.txt
- xml
  - rss-coffee-industry-news.xml
  - rss-the-coffee-break-blog.xml
- build.xml

Create a directory to contain all the files of the sample. These steps refer to this directory as <*JDBC tutorial directory*>. Unzip the contents of JDBCTutorial.zip into <*JDBC tutorial directory*>.

Modify the build.xml file

The `build.xml` file is the build file that Apache Ant uses to compile and execute the JDBC samples. The files `properties/javadb-build-properties.xml` and `properties/mysql-build-properties.xml` contain additional Apache Ant properties required for Java DB and MySQL, respectively. The files `properties/javadb-sample-properties.xml` and `properties/mysql-sample-properties.xml` contain properties required by the sample.
Modify these XML files as follows:

Modify build.xml

In the `build.xml` file, modify the property `ANTPROPERTIES` to refer to either `properties/javadb-build-properties.xml` or `properties/mysql-build-properties.xml`, depending on your DBMS. For example, if you are using Java DB, your `build.xml` file would contain this:

```
<property
  name="ANTPROPERTIES"
  value="properties/javadb-build-properties.xml"/>

  <import file="${ANTPROPERTIES}"/>
```

Similarly, if you are using MySQL, your `build.xml` file would contain this:

```
<property
  name="ANTPROPERTIES"
  value="properties/mysql-build-properties.xml"/>

  <import file="${ANTPROPERTIES}"/>
```

Modify database-specific properties file

In the `properties/javadb-build-properties.xml` or `properties/mysql-build-properties.xml` file (depending on your DBMS), modify the following properties, as described in the following table:

Property	Description
`JAVAC`	The full path name of your Java compiler, `javac`
`JAVA`	The full path name of your Java runtime executable, `java`
`PROPERTIESFILE`	The name of the properties file, either `properties/javadb-sample-properties.xml` or `properties/mysql-sample-properties.xml`
`MYSQLDRIVER`	The full path name of your MySQL driver. For Connector/J, this is typically *`<Connector/J installation directory>`*`/mysql-connector-java-`*`version-number`*`.jar`.
`JAVADBDRIVER`	The full path name of your Java DB driver. This is typically *`<Java DB installation directory>`*`/lib/derby.jar`.
`XALANDIRECTORY`	The full path name of the directory that contains Apache Xalan.
`CLASSPATH`	The class path that the JDBC tutorial uses. *You do not need to change this value.*
`XALAN`	The full path name of the file `xalan.jar`.
`DB.VENDOR`	A value of either `derby` or `mysql` depending on whether you are using Java DB or MySQL, respectively. The tutorial uses this value to construct the URL required to connect to the DBMS and identify DBMS-specific code and SQL statements.
	The fully qualified class name of the JDBC driver. For Java DB, this is

`DB.DRIVER`	`org.apache.derby.jdbc.EmbeddedDriver`. For MySQL, this is `com.mysql.jdbc.Driver`.
`DB.HOST`	The host name of the computer hosting your DBMS.
`DB.PORT`	The port number of the computer hosting your DBMS.
`DB.SID`	The name of the database the tutorial creates and uses.
`DB.URL.NEWDATABASE`	The connection URL used to connect to your DBMS when creating a new database. *You do not need to change this value.*
`DB.URL`	The connection URL used to connect to your DBMS. *You do not need to change this value.*
`DB.USER`	The name of the user that has access to create databases in the DBMS.
`DB.PASSWORD`	The password of the user specified in `DB.USER`.
`DB.DELIMITER`	The character used to separate SQL statements. *Do not change this value.* It should be the semicolon character (`;`).

Modify the tutorial properties file

The tutorial samples use the values in either the `properties/javadb-sample-properties.xml` file or `properties/mysql-sample-properties.xml` file (depending on your DBMS) to connect to the DBMS and initialize databases and tables, as described in the following table:

Property	Description
`dbms`	A value of either `derby` or `mysql` depending on whether you are using Java DB or MySQL, respectively. The tutorial uses this value to construct the URL required to connect to the DBMS and identify DBMS-specific code and SQL statements.
`jar_file`	The full path name of the JAR file that contains all the class files of this tutorial.
`driver`	The fully qualified class name of the JDBC driver. For Java DB, this is `org.apache.derby.jdbc.EmbeddedDriver`. For MySQL, this is `com.mysql.jdbc.Driver`.
`database_name`	The name of the database the tutorial creates and uses.
`user_name`	The name of the user that has access to create databases in the DBMS.
`password`	The password of the user specified in `user_name`.
`server_name`	The host name of the computer hosting your DBMS.
`port_number`	The port number of the computer hosting your DBMS.

Note: For simplicity in demonstrating the JDBC API, the JDBC tutorial sample code does not perform the password management techniques that a deployed system normally uses. In a production environment, you can follow the Oracle Database password management guidelines and disable any sample accounts. See the section Securing Passwords in Application Design in Managing Security for Application Developers in *Oracle Database Security Guide* for password management guidelines and other security recommendations.

Compile and package the samples

At a command prompt, change the current directory to *<JDBC tutorial directory>*. From this directory, run the following command to compile the samples and package them in a jar file:

```
ant jar
```

Create databases, tables, and populate tables

If you are using MySQL, then run the following command to create a database:

```
ant create-mysql-database
```

Note: No corresponding Ant target exists in the `build.xml` file that creates a database for Java DB. The database URL for Java DB, which is used to establish a database connection, includes the option to create the database (if it does not already exist). See Establishing a Connection for more information.

If you are using either Java DB or MySQL, then from the same directory, run the following command to delete existing sample database tables, recreate the tables, and populate them. For Java DB, this command also creates the database if it does not already exist:

```
ant setup
```

Note: You should run the command `ant setup` every time before you run one of the Java classes in the sample. Many of these samples expect specific data in the contents of the sample's database tables.

Run the samples

Each target in the `build.xml` file corresponds to a Java class or SQL script in the JDBC samples. The following table lists the targets in the `build.xml` file, which class or script each target executes, and other classes or files each target requires:

Ant Target	Class or SQL Script	Other Required Classes or Files
`javadb-create-procedure`	`javadb/create-procedures.sql`; see the `build.xml` file to view other SQL statements that are run	No other required files
`mysql-create-procedure`	`mysql/create-procedures.sql`.	No other required files
`run`	`JDBCTutorialUtilities`	No other required classes
`runct`	`CoffeesTable`	`JDBCTutorialUtilities`
`runst`	`SuppliersTable`	`JDBCTutorialUtilities`
`runjrs`	`JdbcRowSetSample`	`JDBCTutorialUtilities`
`runcrs`	`CachedRowSetSample`, `ExampleRowSetListener`	`JDBCTutorialUtilities`
`runjoin`	`JoinSample`	`JDBCTutorialUtilities`
`runfrs`	`FilteredRowSetSample`	`JDBCTutorialUtilities`, `CityFilter`, `StateFilter`
`runwrs`	`WebRowSetSample`	`JDBCTutorialUtilities`
`runclob`	`ClobSample`	`JDBCTutorialUtilities`, `txt/colombian-description.txt`
`runrss`	`RSSFeedsTable`	`JDBCTutorialUtilities`, the XML files contained in the `xml` directory
`rundl`	`DatalinkSample`	`JDBCTutorialUtilities`
`runspjavadb`	`StoredProcedureJavaDBSample`	`JDBCTutorialUtilities`, `SuppliersTable`, `CoffeesTable`
`runspmysql`	`StoredProcedureMySQLSample`	`JDBCTutorialUtilities`, `SuppliersTable`, `CoffeesTable`
`runframe`	`CoffeesFrame`	`JDBCTutorialUtilities`, `CoffeesTableModel`

For example, to run the class `CoffeesTable`, change the current directory to *`<JDBC tutorial directory>`*, and from this directory, run the following command:

```
ant runct
```

Processing SQL Statements with JDBC

In general, to process any SQL statement with JDBC, you follow these steps:

1. Establishing a connection.
2. Create a statement.
3. Execute the query.
4. Process the `ResultSet` object.
5. Close the connection.

This page uses the following method, `CoffeesTables.viewTable`, from the tutorial sample to demonstrate these steps. This method outputs the contents of the table `COFFEES`. This method will be discussed in more detail later in this tutorial:

```
public static void viewTable(Connection con, String dbName)
    throws SQLException {

    Statement stmt = null;
    String query = "select COF_NAME, SUP_ID, PRICE, " +
                   "SALES, TOTAL " +
                   "from " + dbName + ".COFFEES";
    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + "\t" + supplierID +
                               "\t" + price + "\t" + sales +
                               "\t" + total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Establishing Connections

First, establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. This connection is represented by a `Connection` object. See Establishing a Connection for more information.

Creating Statements

A `Statement` is an interface that represents a SQL statement. You execute `Statement` objects, and they generate `ResultSet` objects, which is a table of data representing a database result set. You need a `Connection` object to create a `Statement` object.
For example, `CoffeesTables.viewTable` creates a `Statement` object with the following code:

```
stmt = con.createStatement();
```

There are three different kinds of statements:

- `Statement`: Used to implement simple SQL statements with no parameters.
- `PreparedStatement`: (Extends `Statement`.) Used for precompiling SQL statements that might contain input parameters. See Using Prepared Statements for more information.
- `CallableStatement`: (Extends `PreparedStatement`.) Used to execute stored procedures that may contain both input and output parameters. See Stored Procedures for more information.

Executing Queries

To execute a query, call an `execute` method from `Statement` such as the following:

- `execute`: Returns `true` if the first object that the query returns is a `ResultSet` object. Use this method if the query could return one or more `ResultSet` objects. Retrieve the `ResultSet` objects returned from the query by repeatedly calling `Statement.getResultSet`.
- `executeQuery`: Returns one `ResultSet` object.
- `executeUpdate`: Returns an integer representing the number of rows affected by the SQL statement. Use this method if you are using `INSERT`, `DELETE`, or `UPDATE` SQL statements.

For example, `CoffeesTables.viewTable` executed a `Statement` object with the following code:

```
ResultSet rs = stmt.executeQuery(query);
```

See Retrieving and Modifying Values from Result Sets for more information.

Processing ResultSet Objects

You access the data in a `ResultSet` object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the `ResultSet` object. Initially, the cursor is positioned before the first row. You call various methods defined in the `ResultSet` object to move the cursor.

For example, `CoffeesTables.viewTable` repeatedly calls the method `ResultSet.next` to move the cursor forward by one row. Every time it calls `next`, the method outputs the data in the row where the cursor is currently positioned:

```
try {
    stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
        String coffeeName = rs.getString("COF_NAME");
        int supplierID = rs.getInt("SUP_ID");
        float price = rs.getFloat("PRICE");
        int sales = rs.getInt("SALES");
        int total = rs.getInt("TOTAL");
        System.out.println(coffeeName + "\t" + supplierID +
                           "\t" + price + "\t" + sales +
                           "\t" + total);
    }
}
// ...
```

See Retrieving and Modifying Values from Result Sets for more information.

Closing Connections

When you are finished using a `Statement`, call the method `Statement.close` to immediately release the resources it is using. When you call this method, its `ResultSet` objects are closed.
For example, the method `CoffeesTables.viewTable` ensures that the `Statement` object is closed at the end of the method, regardless of any `SQLException` objects thrown, by wrapping it in a `finally` block:

```
} finally {
    if (stmt != null) { stmt.close(); }
}
```

JDBC throws an `SQLException` when it encounters an error during an interaction with a data source. See Handling SQL Exceptions for more information.
In JDBC 4.1, which is available in Java SE release 7 and later, you can use a try-with-resources statement to automatically close `Connection`, `Statement`, and `ResultSet` objects, regardless of whether an `SQLException` has been thrown. An automatic resource statement consists of a `try` statement and one or more declared resources. For example, you can modify `CoffeesTables.viewTable` so that its `Statement` object closes automatically, as follows:

```
public static void viewTable(Connection con) throws SQLException {

    String query = "select COF_NAME, SUP_ID, PRICE, " +
                   "SALES, TOTAL " +
                   "from COFFEES";

    try (Statement stmt = con.createStatement()) {

        ResultSet rs = stmt.executeQuery(query);

        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + ", " + supplierID +
                               ", " + price + ", " + sales +
                               ", " + total);
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
}
```

The following statement is an `try`-with-resources statement, which declares one resource, `stmt`, that will be automatically closed when the `try` block terminates:

```
try (Statement stmt = con.createStatement()) {
    // ...
}
```

See The `try`-with-resources Statement in the Essential Classes trail for more information.

Establishing a Connection

First, you need to establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. Typically, a JDBC application connects to a target data source using one of two classes:

- `DriverManager`: This fully implemented class connects an application to a data source, which is specified by a database URL. When this class first attempts to establish a connection, it automatically loads any JDBC 4.0 drivers found within the class path. Note that your application must manually load any JDBC drivers prior to version 4.0.
- `DataSource`: This interface is preferred over `DriverManager` because it allows details about the underlying data source to be transparent to your application. A `DataSource` object's properties are set so that it represents a particular data source. See Connecting with DataSource Objects for more information. For more information about developing applications with the `DataSource` class, see the latest *The Java EE Tutorial*.

Note: The samples in this tutorial use the `DriverManager` class instead of the `DataSource` class because it is easier to use and the samples do not require the features of the `DataSource` class.
This page covers the following topics:

Using the DriverManager Class

Connecting to your DBMS with the `DriverManager` class involves calling the method `DriverManager.getConnection`. The following method, `JDBCTutorialUtilities.getConnection`, establishes a database connection:

```
public Connection getConnection() throws SQLException {

    Connection conn = null;
    Properties connectionProps = new Properties();
    connectionProps.put("user", this.userName);
    connectionProps.put("password", this.password);

    if (this.dbms.equals("mysql")) {
        conn = DriverManager.getConnection(
                    "jdbc:" + this.dbms + "://" +
                    this.serverName +
                    ":" + this.portNumber + "/",
                    connectionProps);
    } else if (this.dbms.equals("derby")) {
        conn = DriverManager.getConnection(
                    "jdbc:" + this.dbms + ":" +
                    this.dbName +
                    ";create=true",
                    connectionProps);
    }
    System.out.println("Connected to database");
    return conn;
}
```

The method `DriverManager.getConnection` establishes a database connection. This method requires a database URL, which varies depending on your DBMS. The following are some examples of database URLs:

1. MySQL: `jdbc:mysql://localhost:3306/`, where `localhost` is the name of the server hosting your database, and `3306` is the port number

2. Java DB: `jdbc:derby:`*`testdb`*`;create=true`, where *`testdb`* is the name of the database to connect to, and `create=true` instructs the DBMS to create the database.

Note: This URL establishes a database connection with the Java DB Embedded Driver. Java DB also includes a Network Client Driver, which uses a different URL.

This method specifies the user name and password required to access the DBMS with a `Properties` object.

Note:

- Typically, in the database URL, you also specify the name of an existing database to which you want to connect. For example, the URL `jdbc:mysql://localhost:3306/mysql` represents the database URL for the MySQL database named `mysql`. The samples in this tutorial use a URL that does not specify a specific database because the samples create a new database.
- In previous versions of JDBC, to obtain a connection, you first had to initialize your JDBC driver by calling the method `Class.forName`. This methods required an object of type `java.sql.Driver`. Each JDBC driver contains one or more classes that implements the interface `java.sql.Driver`. The drivers for Java DB are `org.apache.derby.jdbc.EmbeddedDriver` and `org.apache.derby.jdbc.ClientDriver`, and the one for MySQL Connector/J is `com.mysql.jdbc.Driver`. See the documentation of your DBMS driver to obtain the name of the class that implements the interface `java.sql.Driver`.

  Any JDBC 4.0 drivers that are found in your class path are automatically loaded. (However, you must manually load any drivers prior to JDBC 4.0 with the method `Class.forName`.)

The method returns a `Connection` object, which represents a connection with the DBMS or a specific database. Query the database through this object.

Specifying Database Connection URLs

A database connection URL is a string that your DBMS JDBC driver uses to connect to a database. It can contain information such as where to search for the database, the name of the database to connect to, and configuration properties. The exact syntax of a database connection URL is specified by your DBMS.

Java DB Database Connection URLs

The following is the database connection URL syntax for Java DB:

```
jdbc:derby:[subsubprotocol:][databaseName]
    [;attribute=value]*
```

- *`subsubprotocol`* specifies where Java DB should search for the database, either in a directory, in memory, in a class path, or in a JAR file. It is typically omitted.
- *`databaseName`* is the name of the database to connect to.
- *`attribute=value`* represents an optional, semicolon-separated list of attributes. These attributes enable you to instruct Java DB to perform various tasks, including the following:
  - Create the database specified in the connection URL.
  - Encrypt the database specified in the connection URL.
  - Specify directories to store logging and trace information.
  - Specify a user name and password to connect to the database.

See *Java DB Developer's Guide* and *Java DB Reference Manual* from Java DB Technical Documentation for more information.

MySQL Connector/J Database URL

The following is the database connection URL syntax for MySQL Connector/J:

```
jdbc:mysql://[host][,failoverhost...]
    [:port]/[database]
    [?propertyName1][=propertyValue1]
    [&propertyName2][=propertyValue2]...
```

- *`host`*`:`*`port`* is the host name and port number of the computer hosting your database. If not specified, the default values of *`host`* and *`port`* are 127.0.0.1 and 3306, respectively.
- *`database`* is the name of the database to connect to. If not specified, a connection is made with no default database.
- *`failover`* is the name of a standby database (MySQL Connector/J supports failover).
- *`propertyName=propertyValue`* represents an optional, ampersand-separated list of properties. These attributes enable you to instruct MySQL Connector/J to perform various tasks.

See *MySQL Reference Manual* for more information.

Connecting with DataSource Objects

This section covers `DataSource` objects, which are the preferred means of getting a connection to a data source. In addition to their other advantages, which will be explained later, `DataSource` objects can provide connection pooling and distributed transactions. This functionality is essential for enterprise database computing. In particular, it is integral to Enterprise JavaBeans (EJB) technology.
This section shows you how to get a connection using the `DataSource` interface and how to use distributed transactions and connection pooling. Both of these involve very few code changes in your JDBC application.
The work performed to deploy the classes that make these operations possible, which a system administrator usually does with a tool (such as Apache Tomcat or Oracle WebLogic Server), varies with the type of `DataSource` object that is being deployed. As a result, most of this section is devoted to showing how a system administrator sets up the environment so that programmers can use a `DataSource` object to get connections.
The following topics are covered:

Using DataSource Objects to Get a Connection

In Establishing a Connection, you learned how to get a connection using the `DriverManager` class. This section shows you how to use a `DataSource` object to get a connection to your data source, which is the preferred way.

Objects instantiated by classes that implement the `DataSource` represent a particular DBMS or some other data source, such as a file. A `DataSource` object represents a particular DBMS or some other data source, such as a file. If a company uses more than one data source, it will deploy a separate `DataSource` object for each of them. The `DataSource` interface is implemented by a driver vendor. It can be implemented in three different ways:

- A basic `DataSource` implementation produces standard `Connection` objects that are not pooled or used in a distributed transaction.
- A `DataSource` implementation that supports connection pooling produces `Connection` objects that participate in connection pooling, that is, connections that can be recycled.
- A `DataSource` implementation that supports distributed transactions produces `Connection` objects that can be used in a distributed transaction, that is, a transaction that accesses two or more DBMS servers.

A JDBC driver should include at least a basic `DataSource` implementation. For example, the Java DB JDBC driver includes the implementation `org.apache.derby.jdbc.ClientDataSource` and for MySQL, `com.mysql.jdbc.jdbc2.optional.MysqlDataSource`. If your client runs on Java 8 compact profile 2, then the Java DB JDBC driver is `org.apache.derby.jdbc.BasicClientDataSource40`. The sample for this tutorial requires compact profile 3 or greater.

A `DataSource` class that supports distributed transactions typically also implements support for connection pooling. For example, a `DataSource` class provided by an EJB vendor almost always supports both connection pooling and distributed transactions.

Suppose that the owner of the thriving chain of The Coffee Break shops, from the previous examples, has decided to expand further by selling coffee over the Internet. With the large amount of online business expected, the owner will definitely need connection pooling. Opening and closing connections involves a great deal of overhead, and the owner anticipates that this online ordering system will necessitate a sizable number of queries and updates. With connection pooling, a pool of connections can be used over and over again, avoiding the expense of creating a new connection for every database access. In addition, the owner now has a second DBMS that contains data for the recently acquired coffee roasting company. This means that the owner will want to be able to write distributed transactions that use both the old DBMS server and the new one.

The chain owner has reconfigured the computer system to serve the new, larger customer base. The owner has purchased the most recent JDBC driver and an EJB application server that works with it to be able to use distributed transactions and get the increased performance that comes with connection pooling. Many JDBC drivers are available that are compatible with the recently purchased EJB server. The owner now has a three-tier architecture, with a new EJB application server and JDBC driver in the middle tier and the two DBMS servers as the third tier. Client computers making requests are the first tier.

Deploying Basic DataSource Objects

The system administrator needs to deploy `DataSource` objects so that The Coffee Break's programming team can start using them. Deploying a `DataSource` object consists of three tasks:

1. Creating an instance of the `DataSource` class
2. Setting its properties
3. Registering it with a naming service that uses the Java Naming and Directory Interface (JNDI) API

First, consider the most basic case, which is to use a basic implementation of the `DataSource` interface, that is, one that does not support connection pooling or distributed transactions. In this case there is only one `DataSource` object that needs to be deployed. A basic implementation of `DataSource` produces the same kind of connections that the `DriverManager` class produces.

Creating Instance of DataSource Class and Setting its Properties

Suppose a company that wants only a basic implementation of `DataSource` has bought a driver from the JDBC vendor DB Access, Inc. This driver includes the class `com.dbaccess.BasicDataSource` that implements the `DataSource` interface. The following code excerpt creates an instance of the class `BasicDataSource` and sets its properties. After the instance of `BasicDataSource` is deployed, a programmer can call the method `DataSource.getConnection` to get a connection to the company's database, `CUSTOMER_ACCOUNTS`. First, the system administrator creates the `BasicDataSource` object *`ds`* using the default constructor. The system administrator then sets three properties. Note that the following code is typically be executed by a deployment tool:

```
com.dbaccess.BasicDataSource ds = new com.dbaccess.BasicDataSource();
ds.setServerName("grinder");
ds.setDatabaseName("CUSTOMER_ACCOUNTS");
ds.setDescription("Customer accounts database for billing");
```

The variable *`ds`* now represents the database `CUSTOMER_ACCOUNTS` installed on the server. Any connection produced by the `BasicDataSource` object *`ds`* will be a connection to the database `CUSTOMER_ACCOUNTS`.

Registering DataSource Object with Naming Service That Uses JNDI API

With the properties set, the system administrator can register the `BasicDataSource` object with a JNDI (Java Naming and Directory Interface) naming service. The particular naming service that is used is usually determined by a system property, which is not shown here. The following code excerpt registers the `BasicDataSource` object and binds it with the logical name `jdbc/billingDB`:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/billingDB", ds);
```

This code uses the JNDI API. The first line creates an `InitialContext` object, which serves as the starting point for a name, similar to root directory in a file system. The second line associates, or binds, the `BasicDataSource` object *`ds`* to the logical name `jdbc/billingDB`. In the next code excerpt, you give the naming service this logical name, and it returns the `BasicDataSource` object. The logical name can be any string. In this case, the company decided to use the name `billingDB` as the logical name for the `CUSTOMER_ACCOUNTS` database.

In the previous example, `jdbc` is a subcontext under the initial context, just as a directory under the root directory is a subdirectory. The name `jdbc/billingDB` is like a path name, where the last item in the path is analogous to a file name. In this case, `billingDB` is the logical name that is given to the `BasicDataSource` object *`ds`*. The subcontext `jdbc` is reserved for logical names to be bound to `DataSource` objects, so `jdbc` will always be the first part of a logical name for a data source.

Using Deployed DataSource Object

After a basic `DataSource` implementation is deployed by a system administrator, it is ready for a programmer to use. This means that a programmer can give the logical data source name that was bound to an instance of a `DataSource` class, and the JNDI naming service will return an instance of that `DataSource` class. The method `getConnection` can then be called on that `DataSource` object to get a connection to the data source it represents. For example, a programmer might write the following two lines of code to get a `DataSource` object that produces a connection to the database `CUSTOMER_ACCOUNTS`.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/billingDB");
```

The first line of code gets an initial context as the starting point for retrieving a `DataSource` object. When you supply the logical name `jdbc/billingDB` to the method `lookup`, the method returns the `DataSource` object that the system administrator bound to `jdbc/billingDB` at deployment time. Because the return value of the method `lookup` is a Java `Object`, we must cast it to the more specific `DataSource` type before assigning it to the variable *`ds`*.

The variable *`ds`* is an instance of the class `com.dbaccess.BasicDataSource` that implements the `DataSource` interface. Calling the method *`ds`*`.getConnection` produces a connection to the `CUSTOMER_ACCOUNTS` database.

```
Connection con = ds.getConnection("fernanda","brewed");
```

The `getConnection` method requires only the user name and password because the variable *`ds`* has the rest of the information necessary for establishing a connection with the `CUSTOMER_ACCOUNTS` database, such as the database name and location, in its properties.

Advantages of DataSource Objects

Because of its properties, a `DataSource` object is a better alternative than the `DriverManager` class for getting a connection. Programmers no longer have to hard code the driver name or JDBC URL in their applications, which makes them more portable. Also, `DataSource` properties make maintaining code much simpler. If there is a change, the system administrator can update data source properties and not be concerned about changing every application that makes a connection to the data source. For example, if the data source were moved to a different server, all the system administrator would have to do is set the `serverName` property to the new server name.

Aside from portability and ease of maintenance, using a `DataSource` object to get connections can offer other advantages. When the `DataSource` interface is implemented to work with a `ConnectionPoolDataSource` implementation, all of the connections produced by instances of that `DataSource` class will automatically be pooled connections. Similarly, when the `DataSource` implementation is implemented to work with an `XADataSource` class, all of the connections it produces will automatically be connections that can be used in a distributed transaction. The next section shows how to deploy these types of `DataSource` implementations.

Deploying Other DataSource Implementations

A system administrator or another person working in that capacity can deploy a `DataSource` object so that the connections it produces are pooled connections. To do this, he or she first deploys a `ConnectionPoolDataSource` object and then deploys a `DataSource` object implemented to work with it. The properties of the `ConnectionPoolDataSource` object are set so that it represents the data source to which connections will be produced. After the `ConnectionPoolDataSource` object has been registered with a JNDI naming service, the `DataSource` object is deployed. Generally only two properties must be set for the `DataSource` object: `description` and `dataSourceName`. The value given to the `dataSourceName` property is the logical name identifying the `ConnectionPoolDataSource` object previously deployed, which is the object containing the properties needed to make the connection.

With the `ConnectionPoolDataSource` and `DataSource` objects deployed, you can call the method `DataSource.getConnection` on the `DataSource` object and get a pooled connection. This connection will be to the data source specified in the `ConnectionPoolDataSource` object's properties.

The following example describes how a system administrator for The Coffee Break would deploy a `DataSource` object implemented to provide pooled connections. The system administrator would typically use a deployment tool, so the code fragments shown in this section are the code that a deployment tool would execute.

To get better performance, The Coffee Break company has bought a JDBC driver from DB Access, Inc. that includes the class `com.dbaccess.ConnectionPoolDS`, which implements the `ConnectionPoolDataSource` interface. The system administrator creates create an instance of this class, sets its properties, and registers it with a JNDI naming service. The Coffee Break has bought its `DataSource` class, `com.applogic.PooledDataSource`, from its EJB server vendor, Application Logic, Inc. The class `com.applogic.PooledDataSource` implements connection pooling by using the underlying support provided by the `ConnectionPoolDataSource` class `com.dbaccess.ConnectionPoolDS`.

The `ConnectionPoolDataSource` object must be deployed first. The following code creates an instance of `com.dbaccess.ConnectionPoolDS` and sets its properties:

```
com.dbaccess.ConnectionPoolDS cpds = new com.dbaccess.ConnectionPoolDS();
cpds.setServerName("creamer");
cpds.setDatabaseName("COFFEEBREAK");
cpds.setPortNumber(9040);
cpds.setDescription("Connection pooling for " + "COFFEEBREAK DBMS");
```

After the `ConnectionPoolDataSource` object has been deployed, the system administrator deploys the `DataSource` object. The following code registers the `com.dbaccess.ConnectionPoolDS` object *`cpds`* with a JNDI naming service. Note that the logical name being associated with the *`cpds`* variable has the subcontext `pool` added under the subcontext `jdbc`, which is similar to adding a subdirectory to another subdirectory in a hierarchical file system. The logical name of any instance of the class `com.dbaccess.ConnectionPoolDS` will always begin with `jdbc/pool`. Oracle recommends putting all `ConnectionPoolDataSource` objects under the subcontext `jdbc/pool`:

```
Context ctx = new InitialContext();
ctx.bind("jdbc/pool/fastCoffeeDB", cpds);
```

Next, the `DataSource` class that is implemented to interact with the *`cpds`* variable and other instances of the `com.dbaccess.ConnectionPoolDS` class is deployed. The following code creates an instance of this class and sets its properties. Note that only two properties are set for this instance of `com.applogic.PooledDataSource`. The `description` property is set because it is always required. The other property that is set, `dataSourceName`, gives the logical JNDI name for *`cpds`*, which is an instance of the `com.dbaccess.ConnectionPoolDS` class. In other words, *`cpds`* represents the `ConnectionPoolDataSource` object that will implement connection pooling for the `DataSource`

object.

The following code, which would probably be executed by a deployment tool, creates a `PooledDataSource` object, sets its properties, and binds it to the logical name `jdbc/fastCoffeeDB`:

```
com.applogic.PooledDataSource ds = new com.applogic.PooledDataSource();
ds.setDescription("produces pooled connections to COFFEEBREAK");
ds.setDataSourceName("jdbc/pool/fastCoffeeDB");
Context ctx = new InitialContext();
ctx.bind("jdbc/fastCoffeeDB", ds);
```

At this point, a `DataSource` object is deployed from which an application can get pooled connections to the database `COFFEEBREAK`.

Getting and Using Pooled Connections

A *connection pool* is a cache of database connection objects. The objects represent physical database connections that can be used by an application to connect to a database. At run time, the application requests a connection from the pool. If the pool contains a connection that can satisfy the request, it returns the connection to the application. If no connections are found, a new connection is created and returned to the application. The application uses the connection to perform some work on the database and then returns the object back to the pool. The connection is then available for the next connection request.

Connection pools promote the reuse of connection objects and reduce the number of times that connection objects are created. Connection pools significantly improve performance for database-intensive applications because creating connection objects is costly both in terms of time and resources.

Now that these `DataSource` and `ConnectionPoolDataSource` objects are deployed, a programmer can use the `DataSource` object to get a pooled connection. The code for getting a pooled connection is just like the code for getting a nonpooled connection, as shown in the following two lines:

```
ctx = new InitialContext();
ds = (DataSource)ctx.lookup("jdbc/fastCoffeeDB");
```

The variable *`ds`* represents a `DataSource` object that produces pooled connections to the database `COFFEEBREAK`. You need to retrieve this `DataSource` object only once because you can use it to produce as many pooled connections as needed. Calling the method `getConnection` on the *`ds`* variable automatically produces a pooled connection because the `DataSource` object that the *`ds`* variable represents was configured to produce pooled connections.

Connection pooling is generally transparent to the programmer. There are only two things you need to do when you are using pooled connections:

1. Use a `DataSource` object rather than the `DriverManager` class to get a connection. In the following line of code, *`ds`* is a `DataSource` object implemented and deployed so that it will create pooled connections and `username` and `password` are variables that represent the credentials of the user that has access to the database:

```
Connection con = ds.getConnection(username, password);
```

2. Use a `finally` statement to close a pooled connection. The following `finally` block would appear after the `try/catch` block that applies to the code in which the pooled connection was used:

```
try {
    Connection con = ds.getConnection(username, password);
    // ... code to use the pooled
    // connection con
} catch (Exception ex {
    // ... code to handle exceptions
} finally {
    if (con != null) con.close();
}
```

Otherwise, an application using a pooled connection is identical to an application using a regular connection. The only other thing an application programmer might notice when connection pooling is being done is that performance is better.

The following sample code gets a `DataSource` object that produces connections to the database `COFFEEBREAK` and uses it to update a price in the table `COFFEES`:

```
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class ConnectionPoolingBean implements SessionBean {

    // ...

    public void ejbCreate() throws CreateException {
        ctx = new InitialContext();
        ds = (DataSource)ctx.lookup("jdbc/fastCoffeeDB");
    }

    public void updatePrice(float price, String cofName,
```

```
                        String username, String password)
        throws SQLException{

        Connection con;
        PreparedStatement pstmt;
        try {
            con = ds.getConnection(username, password);
            con.setAutoCommit(false);
            pstmt = con.prepareStatement("UPDATE COFFEES " +
                        "SET PRICE = ? " +
                        "WHERE COF_NAME = ?");
            pstmt.setFloat(1, price);
            pstmt.setString(2, cofName);
            pstmt.executeUpdate();

            con.commit();
            pstmt.close();

        } finally {
            if (con != null) con.close();
        }
    }

    private DataSource ds = null;
    private Context ctx = null;
}
```

The connection in this code sample participates in connection pooling because the following are true:

- An instance of a class implementing `ConnectionPoolDataSource` has been deployed.
- An instance of a class implementing `DataSource` has been deployed, and the value set for its `dataSourceName` property is the logical name that was bound to the previously deployed `ConnectionPoolDataSource` object.

Note that although this code is very similar to code you have seen before, it is different in the following ways:

- It imports the `javax.sql`, `javax.ejb`, and `javax.naming` packages in addition to `java.sql`. The `DataSource` and `ConnectionPoolDataSource` interfaces are in the `javax.sql` package, and the JNDI constructor `InitialContext` and method `Context.lookup` are part of the `javax.naming` package. This particular example code is in the form of an EJB component that uses API from the `javax.ejb` package. The purpose of this example is to show that you use a pooled connection the same way you use a nonpooled connection, so you need not worry about understanding the EJB API.
- It uses a `DataSource` object to get a connection instead of using the `DriverManager` facility.
- It uses a `finally` block to ensure that the connection is closed.

Getting and using a pooled connection is similar to getting and using a regular connection. When someone acting as a system administrator has deployed a `ConnectionPoolDataSource` object and a `DataSource` object properly, an application uses that `DataSource` object to get a pooled connection. An application should, however, use a `finally` block to close the pooled connection. For simplicity, the preceding example used a `finally` block but no `catch` block. If an exception is thrown by a method in the `try` block, it will be thrown by default, and the `finally` clause will be executed in any case.

Deploying Distributed Transactions

`DataSource` objects can be deployed to get connections that can be used in distributed transactions. As with connection pooling, two different class instances must be deployed: an `XADataSource` object and a `DataSource` object that is implemented to work with it.

Suppose that the EJB server that The Coffee Break entrepreneur bought includes the `DataSource` class `com.applogic.TransactionalDS`, which works with an `XADataSource` class such as `com.dbaccess.XATransactionalDS`. The fact that it works with any `XADataSource` class makes the EJB server portable across JDBC drivers. When the `DataSource` and `XADataSource` objects are deployed, the connections produced will be able to participate in distributed transactions. In this case, the class `com.applogic.TransactionalDS` is implemented so that the connections produced are also pooled connections, which will usually be the case for `DataSource` classes provided as part of an EJB server implementation.

The `XADataSource` object must be deployed first. The following code creates an instance of `com.dbaccess.XATransactionalDS` and sets its properties:

```
com.dbaccess.XATransactionalDS xads = new com.dbaccess.XATransactionalDS();
xads.setServerName("creamer");
xads.setDatabaseName("COFFEEBREAK");
xads.setPortNumber(9040);
xads.setDescription("Distributed transactions for COFFEEBREAK DBMS");
```

The following code registers the `com.dbaccess.XATransactionalDS` object *`xads`* with a JNDI naming service. Note that the logical name being associated with *`xads`* has the subcontext `xa` added under `jdbc`. Oracle recommends that the logical name of any instance of the class `com.dbaccess.XATransactionalDS` always begin with `jdbc/xa`.

```
Context ctx = new InitialContext();
ctx.bind("jdbc/xa/distCoffeeDB", xads);
```

Next, the `DataSource` object that is implemented to interact with *`xads`* and other `XADataSource` objects is deployed. Note that the `DataSource` class, `com.applogic.TransactionalDS`, can work with an `XADataSource` class from any JDBC driver vendor. Deploying the `DataSource` object involves creating an instance of the `com.applogic.TransactionalDS` class and setting its properties. The `dataSourceName` property is set to `jdbc/xa/distCoffeeDB`, the logical name associated with `com.dbaccess.XATransactionalDS`. This is the `XADataSource` class that implements the distributed transaction capability for the `DataSource` class. The following code deploys an instance of the `DataSource` class:

```
com.applogic.TransactionalDS ds = new com.applogic.TransactionalDS();
ds.setDescription("Produces distributed transaction " +
                  "connections to COFFEEBREAK");
ds.setDataSourceName("jdbc/xa/distCoffeeDB");
Context ctx = new InitialContext();
ctx.bind("jdbc/distCoffeeDB", ds);
```

Now that instances of the classes `com.applogic.TransactionalDS` and `com.dbaccess.XATransactionalDS` have been deployed, an application can call the method `getConnection` on instances of the `TransactionalDS` class to get a connection to the `COFFEEBREAK` database that can be used in distributed transactions.

Using Connections for Distributed Transactions

To get a connection that can be used for distributed transactions, must use a `DataSource` object that has been properly implemented and deployed, as shown in the section Deploying Distributed Transactions. With such a `DataSource` object, call the method `getConnection` on it. After you have the connection, use it just as you would use any other connection. Because `jdbc/distCoffeesDB` has been associated with an `XADataSource` object in a JNDI naming service, the following code produces a `Connection` object that can be used in distributed transactions:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/distCoffeesDB");
Connection con = ds.getConnection();
```

There are some minor but important restrictions on how this connection is used while it is part of a distributed transaction. A transaction manager controls when a distributed transaction begins and when it is committed or rolled back; therefore, application code should never call the methods `Connection.commit` or `Connection.rollback`. An application should likewise never call `Connection.setAutoCommit(true)`, which enables the auto-commit mode, because that would also interfere with the transaction manager's control of the transaction boundaries. This explains why a new connection that is created in the scope of a distributed transaction has its auto-commit mode disabled by default. Note that these restrictions apply only when a connection is participating in a distributed transaction; there are no restrictions while the connection is not part of a distributed transaction.

For the following example, suppose that an order of coffee has been shipped, which triggers updates to two tables that reside on different DBMS servers. The first table is a new `INVENTORY` table, and the second is the `COFFEES` table. Because these tables are on different DBMS servers, a transaction that involves both of them will be a distributed transaction. The code in the following example, which obtains a connection, updates the `COFFEES` table, and closes the connection, is the second part of a distributed transaction.

Note that the code does not explicitly commit or roll back the updates because the scope of the distributed transaction is being controlled by the middle tier server's underlying system infrastructure. Also, assuming that the connection used for the distributed transaction is a pooled connection, the application uses a `finally` block to close the connection. This guarantees that a valid connection will be closed even if an exception is thrown, thereby ensuring that the connection is returned to the connection pool to be recycled.

The following code sample illustrates an enterprise Bean, which is a class that implements the methods that can be called by a client computer. The purpose of this example is to demonstrate that application code for a distributed transaction is no different from other code except that it does not call the `Connection` methods `commit`, `rollback`, or `setAutoCommit(true)`. Therefore, you do not need to worry about understanding the EJB API that is used.

```
import java.sql.*;
import javax.sql.*;
import javax.ejb.*;
import javax.naming.*;

public class DistributedTransactionBean implements SessionBean {

    // ...

    public void ejbCreate() throws CreateException {

        ctx = new InitialContext();
        ds = (DataSource)ctx.lookup("jdbc/distCoffeesDB");
    }

    public void updateTotal(int incr, String cofName, String username,
                            String password)
        throws SQLException {

        Connection con;
        PreparedStatement pstmt;

        try {
            con = ds.getConnection(username, password);
            pstmt = con.prepareStatement("UPDATE COFFEES " +
                        "SET TOTAL = TOTAL + ? " +
```

```
                    "WHERE COF_NAME = ?");
            pstmt.setInt(1, incr);
            pstmt.setString(2, cofName);
            pstmt.executeUpdate();
            stmt.close();
        } finally {
            if (con != null) con.close();
        }
    }

    private DataSource ds = null;
    private Context ctx = null;
}
```

Handling SQLExceptions

This page covers the following topics:

Overview of SQLException

When JDBC encounters an error during an interaction with a data source, it throws an instance of `SQLException` as opposed to `Exception`. (A data source in this context represents the database to which a `Connection` object is connected.) The `SQLException` instance contains the following information that can help you determine the cause of the error:

- A description of the error. Retrieve the `String` object that contains this description by calling the method `SQLException.getMessage`.
- A SQLState code. These codes and their respective meanings have been standardized by ISO/ANSI and Open Group (X/Open), although some codes have been reserved for database vendors to define for themselves. This `String` object consists of five alphanumeric characters. Retrieve this code by calling the method `SQLException.getSQLState`.
- An error code. This is an integer value identifying the error that caused the `SQLException` instance to be thrown. Its value and meaning are implementation-specific and might be the actual error code returned by the underlying data source. Retrieve the error by calling the method `SQLException.getErrorCode`.
- A cause. A `SQLException` instance might have a causal relationship, which consists of one or more `Throwable` objects that caused the `SQLException` instance to be thrown. To navigate this chain of causes, recursively call the method `SQLException.getCause` until a `null` value is returned.
- A reference to any *chained* exceptions. If more than one error occurs, the exceptions are referenced through this chain. Retrieve these exceptions by calling the method `SQLException.getNextException` on the exception that was thrown.

Retrieving Exceptions

The following method, `JDBCTutorialUtilities.printSQLException` outputs the SQLState, error code, error description, and cause (if there is one) contained in the `SQLException` as well as any other exception chained to it:

```
public static void printSQLException(SQLException ex) {

    for (Throwable e : ex) {
        if (e instanceof SQLException) {
            if (ignoreSQLException(
                ((SQLException)e).
                getSQLState()) == false) {

                e.printStackTrace(System.err);
                System.err.println("SQLState: " +
                    ((SQLException)e).getSQLState());

                System.err.println("Error Code: " +
                    ((SQLException)e).getErrorCode());

                System.err.println("Message: " + e.getMessage());

                Throwable t = ex.getCause();
                while(t != null) {
                    System.out.println("Cause: " + t);
                    t = t.getCause();
                }
            }
        }
    }
}
```

For example, if you call the method `CoffeesTable.dropTable` with Java DB as your DBMS, the table `COFFEES` does not exist, *and* you remove the call to `JDBCTutorialUtilities.ignoreSQLException`, the output will be similar to the following:

```
SQLState: 42Y55
Error Code: 30000
Message: 'DROP TABLE' cannot be performed on
'TESTDB.COFFEES' because it does not exist.
```

Instead of outputting `SQLException` information, you could instead first retrieve the `SQLState` then process the `SQLException` accordingly. For example, the method `JDBCTutorialUtilities.ignoreSQLException` returns `true` if the `SQLState` is equal to code `42Y55` (and you are using Java DB as your DBMS), which causes `JDBCTutorialUtilities.printSQLException` to ignore the `SQLException`:

```
public static boolean ignoreSQLException(String sqlState) {

    if (sqlState == null) {
        System.out.println("The SQL state is not defined!");
        return false;
    }

    // X0Y32: Jar file already exists in schema
    if (sqlState.equalsIgnoreCase("X0Y32"))
        return true;

    // 42Y55: Table already exists in schema
    if (sqlState.equalsIgnoreCase("42Y55"))
        return true;

    return false;
}
```

Retrieving Warnings

`SQLWarning` objects are a subclass of `SQLException` that deal with database access warnings. Warnings do not stop the execution of an application, as exceptions do; they simply alert the user that something did not happen as planned. For example, a warning might let you know that a privilege you attempted to revoke was not revoked. Or a warning might tell you that an error occurred during a requested disconnection.

A warning can be reported on a `Connection` object, a `Statement` object (including `PreparedStatement` and `CallableStatement` objects), or a `ResultSet` object. Each of these classes has a `getWarnings` method, which you must invoke in order to see the first warning reported on the calling object. If `getWarnings` returns a warning, you can call the `SQLWarning` method `getNextWarning` on it to get any additional warnings. Executing a statement automatically clears the warnings from a previous statement, so they do not build up. This means, however, that if you want to retrieve warnings reported on a statement, you must do so before you execute another statement.

The following methods from `JDBCTutorialUtilities` illustrate how to get complete information about any warnings reported on `Statement` or `ResultSet` objects:

```
public static void getWarningsFromResultSet(ResultSet rs)
    throws SQLException {
    JDBCTutorialUtilities.printWarnings(rs.getWarnings());
}

public static void getWarningsFromStatement(Statement stmt)
    throws SQLException {
    JDBCTutorialUtilities.printWarnings(stmt.getWarnings());
}

public static void printWarnings(SQLWarning warning)
    throws SQLException {

    if (warning != null) {
        System.out.println("\n---Warning---\n");

    while (warning != null) {
        System.out.println("Message: " + warning.getMessage());
        System.out.println("SQLState: " + warning.getSQLState());
        System.out.print("Vendor error code: ");
        System.out.println(warning.getErrorCode());
        System.out.println("");
        warning = warning.getNextWarning();
    }
}
```

The most common warning is a `DataTruncation` warning, a subclass of `SQLWarning`. All `DataTruncation` objects have a SQLState of `01004`, indicating that there was a problem with reading or writing data. `DataTruncation` methods let you find out in which column or parameter data was truncated, whether the truncation was on a read or write operation, how many bytes should have been transferred, and how many bytes were actually transferred.

Categorized SQLExceptions

Your JDBC driver might throw a subclass of `SQLException` that corresponds to a common SQLState or a common error state that is not associated with a specific SQLState class value. This enables you to write more portable error-handling code. These exceptions are subclasses of one of the following classes:

- `SQLNonTransientException`
- `SQLTransientException`
- `SQLRecoverableException`

See the latest Javadoc of the `java.sql` package or the documentation of your JDBC driver for more information about these subclasses.

Other Subclasses of SQLException

The following subclasses of `SQLException` can also be thrown:

- `BatchUpdateException` is thrown when an error occurs during a batch update operation. In addition to the information provided by `SQLException`, `BatchUpdateException` provides the update counts for all statements that were executed before the error occurred.
- `SQLClientInfoException` is thrown when one or more client information properties could not be set on a Connection. In addition to the information provided by `SQLException`, `SQLClientInfoException` provides a list of client information properties that were not set.

Setting Up Tables

This page describes all the tables used in the JDBC tutorial and how to create them:

COFFEES Table

The COFFEES table stores information about the coffees available for sale at The Coffee Break:

COF_NAME	SUP_ID	PRICE	SALES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

The following describes each of the columns in the COFFEES table:

- COF_NAME: Stores the coffee name. Holds values with a SQL type of VARCHAR with a maximum length of 32 characters. Because the names are different for each type of coffee sold, the name uniquely identifies a particular coffee and serves as the primary key.
- SUP_ID: Stores a number identifying the coffee supplier. Holds values with a SQL type of INTEGER. It is defined as a foreign key that references the column SUP_ID in the SUPPLIERS table. Consequently, the DBMS will enforce that each value in this column matches one of the values in the corresponding column in the SUPPLIERS table.
- PRICE: Stores the cost of the coffee per pound. Holds values with a SQL type of FLOAT because it needs to hold values with decimal points. (Note that money values would typically be stored in a SQL type DECIMAL or NUMERIC, but because of differences among DBMSs and to avoid incompatibility with earlier versions of JDBC, the tutorial uses the more standard type FLOAT.)
- SALES: Stores the number of pounds of coffee sold during the current week. Holds values with a SQL type of INTEGER.
- TOTAL: Stores the number of pounds of coffee sold to date. Holds values with a SQL type of INTEGER.

SUPPLIERS Table

The SUPPLIERS stores information about each of the suppliers:

SUP_ID	SUP_NAME	STREET	CITY	STATE	ZIP
101	Acme, Inc.	99 Market Street	Groundsville	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

The following describes each of the columns in the SUPPLIERS table:

- SUP_ID: Stores a number identifying the coffee supplier. Holds values with a SQL type of INTEGER. It is the primary key in this table.
- SUP_NAME: Stores the name of the coffee supplier.
- STREET, CITY, STATE, and ZIP: These columns store the address of the coffee supplier.

COF_INVENTORY Table

The table `COF_INVENTORY` stores information about the amount of coffee stored in each warehouse:

WAREHOUSE_ID	COF_NAME	SUP_ID	QUAN	DATE_VAL
1234	House_Blend	49	0	2006_04_01
1234	House_Blend_Decaf	49	0	2006_04_01
1234	Colombian	101	0	2006_04_01
1234	French_Roast	49	0	2006_04_01
1234	Espresso	150	0	2006_04_01
1234	Colombian_Decaf	101	0	2006_04_01

The following describes each of the columns in the `COF_INVENTORY` table:

- `WAREHOUSE_ID`: Stores a number identifying a warehouse.
- `COF_NAME`: Stores the name of a particular type of coffee.
- `SUP_ID`: Stores a number identifying a supplier.
- `QUAN`: Stores a number indicating the amount of merchandise available.
- `DATE`: Stores a timestamp value indicating the last time the row was updated.

MERCH_INVENTORY Table

The table MERCH_INVENTORY stores information about the amount of non-coffee merchandise in stock:

ITEM_ID	ITEM_NAME	SUP_ID	QUAN	DATE
00001234	Cup_Large	00456	28	2006_04_01
00001235	Cup_Small	00456	36	2006_04_01
00001236	Saucer	00456	64	2006_04_01
00001287	Carafe	00456	12	2006_04_01
00006931	Carafe	00927	3	2006_04_01
00006935	PotHolder	00927	88	2006_04_01
00006977	Napkin	00927	108	2006_04_01
00006979	Towel	00927	24	2006_04_01
00004488	CofMaker	08732	5	2006_04_01
00004490	CofGrinder	08732	9	2006_04_01
00004495	EspMaker	08732	4	2006_04_01
00006914	Cookbook	00927	12	2006_04_01

The following describes each of the columns in the MERCH_INVENTORY table:

- ITEM_ID: Stores a number identifying an item.
- ITEM_NAME: Stores the name of an item.
- SUP_ID: Stores a number identifying a supplier.
- QUAN: Stores a number indicating the amount of that item available.
- DATE: Stores a timestamp value indicating the last time the row was updated.

COFFEE_HOUSES Table

The table COFFEE_HOUSES stores locations of coffee houses:

STORE_ID	CITY	COFFEE	MERCH	TOTAL
10023	Mendocino	3450	2005	5455
33002	Seattle	4699	3109	7808
10040	SF	5386	2841	8227
32001	Portland	3147	3579	6726
10042	SF	2863	1874	4710
10024	Sacramento	1987	2341	4328
10039	Carmel	2691	1121	3812
10041	LA	1533	1007	2540
33005	Olympia	2733	1550	4283
33010	Seattle	3210	2177	5387
10035	SF	1922	1056	2978
10037	LA	2143	1876	4019
10034	San_Jose	1234	1032	2266
32004	Eugene	1356	1112	2468

The following describes each of the columns in the COFFEE_HOUSES table:

- STORE_ID: Stores a number identifying a coffee house. It indicates, among other things, the state in which the coffee house is located. A value beginning with 10, for example, means that the state is California. STORE_ID values beginning with 32 indicate Oregon, and those beginning with 33 indicate the state of Washington.
- CITY: Stores the name of the city in which the coffee house is located.
- COFFEE: Stores a number indicating the amount of coffee sold.
- MERCH: Stores a number indicating the amount of coffee sold.
- TOTAL: Stores a number indicating the total amount of coffee and merchandise sold.

DATA_REPOSITORY Table

The table DATA_REPOSITORY stores URLs that reference documents and other data of interest to The Coffee Break. The script `populate_tables.sql` does not add any data to this table. The following describes each of the columns in this table:

- `DOCUMENT_NAME`: Stores a string that identifies the URL.
- `URL`: Stores a URL.

Creating Tables

You can create tables with Apache Ant or JDBC API.

Creating Tables with Apache Ant

To create the tables used with the tutorial sample code, run the following command in the directory *<JDBC tutorial directory>*:

```
ant setup
```

This command runs several Ant targets, including the following, `build-tables` (from the `build.xml` file):

```
<target name="build-tables"
  description="Create database tables">
  <sql
    driver="${DB.DRIVER}"
    url="${DB.URL}"
    userid="${DB.USER}"
    password="${DB.PASSWORD}"
    classpathref="CLASSPATH"
    delimiter="${DB.DELIMITER}"
    autocommit="false" onerror="abort">
    <transaction src=
  "./sql/${DB.VENDOR}/create-tables.sql"/>
  </sql>
</target>
```

The sample specifies values for the following `sql` Ant task parameters:

Parameter	Description
`driver`	Fully qualified class name of your JDBC driver. This sample uses `org.apache.derby.jdbc.EmbeddedDriver` for Java DB and `com.mysql.jdbc.Driver` for MySQL Connector/J.
`url`	Database connection URL that your DBMS JDBC driver uses to connect to a database.
`userid`	Name of a valid user in your DBMS.
`password`	Password of the user specified in `userid`
`classpathref`	Full path name of the JAR file that contains the class specified in `driver`
`delimiter`	String or character that separates SQL statements. This sample uses the semicolon (`;`).
`autocommit`	Boolean value; if set to `false`, all SQL statements are executed as one transaction.
`onerror`	Action to perform when a statement fails; possible values are `continue`, `stop`, and `abort`. The value `abort` specifies that if an error occurs, the transaction is aborted.

The sample stores the values of these parameters in a separate file. The build file `build.xml` retrieves these values with the `import` task:

```
<import file="${ANTPROPERTIES}"/>
```

The `transaction` element specifies a file that contains SQL statements to execute. The file `create-tables.sql` contains SQL statements that create all the tables described on this page. For example, the following excerpt from this file creates the tables `SUPPLIERS` and `COFFEES`:

```
create table SUPPLIERS
    (SUP_ID integer NOT NULL,
    SUP_NAME varchar(40) NOT NULL,
    STREET varchar(40) NOT NULL,
    CITY varchar(20) NOT NULL,
    STATE char(2) NOT NULL,
    ZIP char(5),
    PRIMARY KEY (SUP_ID));

create table COFFEES
    (COF_NAME varchar(32) NOT NULL,
    SUP_ID int NOT NULL,
    PRICE numeric(10,2) NOT NULL,
    SALES integer NOT NULL,
    TOTAL integer NOT NULL,
```

```
    PRIMARY KEY (COF_NAME),
    FOREIGN KEY (SUP_ID)
        REFERENCES SUPPLIERS (SUP_ID));
```

Note: The file `build.xml` contains another target named `drop-tables` that deletes the tables used by the tutorial. The `setup` target runs `drop-tables` before running the `build-tables` target.

Creating Tables with JDBC API

The following method, `SuppliersTable.createTable`, creates the `SUPPLIERS` table:

```
public void createTable() throws SQLException {
    String createString =
        "create table " + dbName +
        ".SUPPLIERS " +
        "(SUP_ID integer NOT NULL, " +
        "SUP_NAME varchar(40) NOT NULL, " +
        "STREET varchar(40) NOT NULL, " +
        "CITY varchar(20) NOT NULL, " +
        "STATE char(2) NOT NULL, " +
        "ZIP char(5), " +
        "PRIMARY KEY (SUP_ID))";

    Statement stmt = null;
    try {
        stmt = con.createStatement();
        stmt.executeUpdate(createString);
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

The following method, `CoffeesTable.createTable`, creates the `COFFEES` table:

```
  public void createTable() throws SQLException {
    String createString =
        "create table " + dbName +
        ".COFFEES " +
        "(COF_NAME varchar(32) NOT NULL, " +
        "SUP_ID int NOT NULL, " +
        "PRICE float NOT NULL, " +
        "SALES integer NOT NULL, " +
        "TOTAL integer NOT NULL, " +
        "PRIMARY KEY (COF_NAME), " +
        "FOREIGN KEY (SUP_ID) REFERENCES " +
        dbName + ".SUPPLIERS (SUP_ID))";

    Statement stmt = null;
    try {
        stmt = con.createStatement();
        stmt.executeUpdate(createString);
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

In both methods, `con` is a `Connection` object and `dbName` is the name of the database in which you are creating the table.

To execute the SQL query, such as those specified by the `String createString`, use a `Statement` object. To create a `Statement` object, call the method `Connection.createStatement` from an existing `Connection` object. To execute a SQL query, call the method `Statement.executeUpdate`.

All `Statement` objects are closed when the connection that created them is closed. However, it is good coding practice to explicitly close `Statement` objects as soon as you are finished with them. This allows any external resources that the statement is using to be released immediately. Close a statement by calling the method `Statement.close`. Place this statement in a `finally` to ensure that it closes even if the normal program flow is interrupted because an exception (such as `SQLException`) is thrown.

Note: You must create the `SUPPLIERS` table before the `COFFEES` because `COFFEES` contains a foreign key, `SUP_ID` that references `SUPPLIERS`.

Populating Tables

Similarly, you can insert data into tables with Apache Ant or JDBC API.

Populating Tables with Apache Ant

In addition to creating the tables used by this tutorial, the command `ant setup` also populates these tables. This command runs the Ant target `populate-tables`, which runs the SQL script `populate-tables.sql`.
The following is an excerpt from `populate-tables.sql` that populates the tables `SUPPLIERS` and `COFFEES`:

```
insert into SUPPLIERS values(
    49, 'Superior Coffee', '1 Party Place',
    'Mendocino', 'CA', '95460');
insert into SUPPLIERS values(
    101, 'Acme, Inc.', '99 Market Street',
    'Groundsville', 'CA', '95199');
insert into SUPPLIERS values(
    150, 'The High Ground',
    '100 Coffee Lane', 'Meadows', 'CA', '93966');
insert into COFFEES values(
    'Colombian', 00101, 7.99, 0, 0);
insert into COFFEES values(
    'French_Roast', 00049, 8.99, 0, 0);
insert into COFFEES values(
    'Espresso', 00150, 9.99, 0, 0);
insert into COFFEES values(
    'Colombian_Decaf', 00101, 8.99, 0, 0);
insert into COFFEES values(
    'French_Roast_Decaf', 00049, 9.99, 0, 0);
```

Populating Tables with JDBC API

The following method, `SuppliersTable.populateTable`, inserts data into the table:

```
public void populateTable() throws SQLException {

    Statement stmt = null;
    try {
        stmt = con.createStatement();
        stmt.executeUpdate(
            "insert into " + dbName +
            ".SUPPLIERS " +
            "values(49, 'Superior Coffee', " +
            "'1 Party Place', " +
            "'Mendocino', 'CA', '95460')");

        stmt.executeUpdate(
            "insert into " + dbName +
            ".SUPPLIERS " +
            "values(101, 'Acme, Inc.', " +
            "'99 Market Street', " +
            "'Groundsville', 'CA', '95199')");

        stmt.executeUpdate(
            "insert into " + dbName +
            ".SUPPLIERS " +
            "values(150, " +
            "'The High Ground', " +
            "'100 Coffee Lane', " +
            "'Meadows', 'CA', '93966')");
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

The following method, `CoffeesTable.populateTable`, inserts data into the table:

```
public void populateTable() throws SQLException {

    Statement stmt = null;
    try {
        stmt = con.createStatement();
        stmt.executeUpdate(
            "insert into " + dbName +
            ".COFFEES " +
            "values('Colombian', 00101, " +
            "7.99, 0, 0)");

        stmt.executeUpdate(
            "insert into " + dbName +
            ".COFFEES " +
            "values('French_Roast', " +
            "00049, 8.99, 0, 0)");

        stmt.executeUpdate(
            "insert into " + dbName +
            ".COFFEES " +
            "values('Espresso', 00150, 9.99, 0, 0)");

        stmt.executeUpdate(
            "insert into " + dbName +
            ".COFFEES " +
            "values('Colombian_Decaf', " +
            "00101, 8.99, 0, 0)");
```

```
        stmt.executeUpdate(
            "insert into " + dbName +
            ".COFFEES " +
            "values('French_Roast_Decaf', " +
            "00049, 9.99, 0, 0)");
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) {
          stmt.close();
        }
    }
}
```

Retrieving and Modifying Values from Result Sets

The following method, `CoffeesTable.viewTable` outputs the contents of the `COFFEES` tables, and demonstrates the use of `ResultSet` objects and cursors:

```
public static void viewTable(Connection con, String dbName)
    throws SQLException {

    Statement stmt = null;
    String query =
        "select COF_NAME, SUP_ID, PRICE, " +
        "SALES, TOTAL " +
        "from " + dbName + ".COFFEES";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString("COF_NAME");
            int supplierID = rs.getInt("SUP_ID");
            float price = rs.getFloat("PRICE");
            int sales = rs.getInt("SALES");
            int total = rs.getInt("TOTAL");
            System.out.println(coffeeName + "\t" + supplierID +
                               "\t" + price + "\t" + sales +
                               "\t" + total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

A `ResultSet` object is a table of data representing a database result set, which is usually generated by executing a statement that queries the database. For example, the `CoffeeTables.viewTable` method creates a `ResultSet`, `rs`, when it executes the query through the `Statement` object, `stmt`. Note that a `ResultSet` object can be created through any object that implements the `Statement` interface, including `PreparedStatement`, `CallableStatement`, and `RowSet`.

You access the data in a `ResultSet` object through a cursor. Note that this cursor is not a database cursor. This cursor is a pointer that points to one row of data in the `ResultSet`. Initially, the cursor is positioned before the first row. The method `ResultSet.next` moves the cursor to the next row. This method returns `false` if the cursor is positioned after the last row. This method repeatedly calls the `ResultSet.next` method with a `while` loop to iterate through all the data in the `ResultSet`.

This page covers the following topics:

ResultSet Interface

The `ResultSet` interface provides methods for retrieving and manipulating the results of executed queries, and `ResultSet` objects can have different functionality and characteristics. These characteristics are type, concurrency, and cursor *holdability*.

ResultSet Types

The type of a `ResultSet` object determines the level of its functionality in two areas: the ways in which the cursor can be manipulated, and how concurrent changes made to the underlying data source are reflected by the `ResultSet` object.
The sensitivity of a `ResultSet` object is determined by one of three different `ResultSet` types:

- `TYPE_FORWARD_ONLY`: The result set cannot be scrolled; 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 generates 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.
- `TYPE_SCROLL_INSENSITIVE`: The result can be scrolled; 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.
- `TYPE_SCROLL_SENSITIVE`: The result can be scrolled; 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`.
Note: Not all databases and JDBC drivers support all `ResultSet` types. The method `DatabaseMetaData.supportsResultSetType` returns `true` if the specified `ResultSet` type is supported and `false` otherwise.

ResultSet Concurrency

The concurrency of a `ResultSet` object determines what level of update functionality is supported.
There are two concurrency levels:

- `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`.
Note: Not all JDBC drivers and databases support concurrency. The method `DatabaseMetaData.supportsResultSetConcurrency` returns `true` if the specified concurrency level is supported by the driver and `false` otherwise.
The method `CoffeesTable.modifyPrices` demonstrates how to use a `ResultSet` object whose concurrency level is `CONCUR_UPDATABLE`.

Cursor 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 behavior. 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`:

- `HOLD_CURSORS_OVER_COMMIT`: `ResultSet` cursors are not closed; they are *holdable*: they are held open when the method `commit` is called. Holdable cursors might be ideal if your application uses mostly read-only `ResultSet` objects.
- `CLOSE_CURSORS_AT_COMMIT`: `ResultSet` objects (cursors) are closed when the `commit` method is called. Closing cursors when this method is called can result in better performance for some applications.

The default cursor holdability varies depending on your DBMS.
Note: Not all JDBC drivers and databases support holdable and non-holdable cursors. The following `method`, `JDBCTutorialUtilities.cursorHoldabilitySupport`, outputs the default cursor holdability of `ResultSet` objects and whether `HOLD_CURSORS_OVER_COMMIT` and `CLOSE_CURSORS_AT_COMMIT` are supported:

```
public static void cursorHoldabilitySupport(Connection conn)
    throws SQLException {

    DatabaseMetaData dbMetaData = conn.getMetaData();
    System.out.println("ResultSet.HOLD_CURSORS_OVER_COMMIT = " +
        ResultSet.HOLD_CURSORS_OVER_COMMIT);

    System.out.println("ResultSet.CLOSE_CURSORS_AT_COMMIT = " +
        ResultSet.CLOSE_CURSORS_AT_COMMIT);

    System.out.println("Default cursor holdability: " +
        dbMetaData.getResultSetHoldability());

    System.out.println("Supports HOLD_CURSORS_OVER_COMMIT? " +
        dbMetaData.supportsResultSetHoldability(
            ResultSet.HOLD_CURSORS_OVER_COMMIT));

    System.out.println("Supports CLOSE_CURSORS_AT_COMMIT? " +
        dbMetaData.supportsResultSetHoldability(
            ResultSet.CLOSE_CURSORS_AT_COMMIT));
}
```

Retrieving Column Values from Rows

The `ResultSet` interface declares getter methods (for example, `getBoolean` and `getLong`) for retrieving column values from the current row. You can retrieve values using either the index number of the column or the alias or name of the column. The column index is usually more efficient. Columns are numbered from 1. For maximum portability, result set columns within each row should be read in left-to-right order, and each column should be read only once.

For example, the following method, `CoffeesTable.alternateViewTable`, retrieves column values by number:

```
public static void alternateViewTable(Connection con)
    throws SQLException {

    Statement stmt = null;
    String query =
        "select COF_NAME, SUP_ID, PRICE, " +
        "SALES, TOTAL from COFFEES";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            String coffeeName = rs.getString(1);
            int supplierID = rs.getInt(2);
            float price = rs.getFloat(3);
            int sales = rs.getInt(4);
            int total = rs.getInt(5);
            System.out.println(coffeeName + "\t" + supplierID +
                               "\t" + price + "\t" + sales +
                               "\t" + total);
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Strings used as input to getter methods are case-insensitive. When a getter method is called with a string and more than one column has the same alias or name as the string, the value of the first matching column is returned. The option to use a string as opposed to an integer is designed to be used when column aliases and names are used in the SQL query that generated the result set. For columns that are *not* explicitly named in the query (for example, `select * from COFFEES`) it is best to use column numbers. If column names are used, the developer should guarantee that they uniquely refer to the intended columns by using column aliases. A column alias effectively renames the column of a result set. To specify a column alias, use the SQL `AS` clause in the `SELECT` statement.

The getter method of the appropriate type retrieves the value in each column. For example, in the method `CoffeeTables.viewTable`, the first column in each row of the `ResultSet rs` is `COF_NAME`, which stores a value of SQL type `VARCHAR`. The method for retrieving a value of SQL type `VARCHAR` is `getString`. The second column in each row stores a value of SQL type `INTEGER`, and the method for retrieving values of that type is `getInt`.

Note that although the method `getString` is recommended for retrieving the SQL types `CHAR` and `VARCHAR`, it is possible to retrieve any of the basic SQL types with it. Getting all values with `getString` can be very useful, but it also has its limitations. For instance, if it is used to retrieve a numeric type, `getString` converts the numeric value to a Java `String` object, and the value has to be converted back to a numeric type before it can be operated on as a number. In cases where the value is treated as a string anyway, there is no drawback. Furthermore, if you want an application to retrieve values of any standard SQL type other than SQL3 types, use the `getString` method.

Cursors

As mentioned previously, you access the data in a `ResultSet` object through a cursor, which points to one row in the `ResultSet` object. However, when a `ResultSet` object is first created, the cursor is positioned before the first row. The method `CoffeeTables.viewTable` moves the cursor by calling the `ResultSet.next` method. There are other methods available 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 backward 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.
- `absolute(int row)`: Positions the cursor on the row specified by the parameter `row`.

Note that the default sensitivity of a `ResultSet` is `TYPE_FORWARD_ONLY`, which means that it cannot be scrolled; you cannot call any of these methods that move the cursor, except `next`, if your `ResultSet` cannot be scrolled. The method `CoffeesTable.modifyPrices`, described in the following section, demonstrates how you can move the cursor of a `ResultSet`.

Updating Rows in ResultSet Objects

You cannot update a default `ResultSet` object, and you can only move its cursor forward. However, you can create `ResultSet` objects that can be scrolled (the cursor can move backwards or move to an absolute position) and updated.

The following method, `CoffeesTable.modifyPrices`, multiplies the `PRICE` column of each row by the argument `percentage`:

```
public void modifyPrices(float percentage) throws SQLException {

    Statement stmt = null;
    try {
        stmt = con.createStatement();
        stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                   ResultSet.CONCUR_UPDATABLE);
        ResultSet uprs = stmt.executeQuery(
            "SELECT * FROM " + dbName + ".COFFEES");

        while (uprs.next()) {
            float f = uprs.getFloat("PRICE");
            uprs.updateFloat( "PRICE", f * percentage);
            uprs.updateRow();
        }

    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

The field `ResultSet.TYPE_SCROLL_SENSITIVE` creates a `ResultSet` object whose cursor can move both forward and backward relative to the current position and to an absolute position. The field `ResultSet.CONCUR_UPDATABLE` creates a `ResultSet` object that can be updated. See the `ResultSet` Javadoc for other fields you can specify to modify the behavior of `ResultSet` objects.

The method `ResultSet.updateFloat` updates the specified column (in this example, `PRICE` with the specified `float` value in the row where the cursor is positioned. `ResultSet` contains various updater methods that enable you to update column values of various data types. However, none of these updater methods modifies the database; you must call the method `ResultSet.updateRow` to update the database.

Using Statement Objects for Batch Updates

`Statement`, `PreparedStatement` and `CallableStatement` objects have a list of commands that is associated with them. This list may contain statements for updating, inserting, or deleting a row; and it may also contain DDL statements such as `CREATE TABLE` and `DROP TABLE`. It cannot, however, contain a statement that would produce a `ResultSet` object, such as a `SELECT` statement. In other words, the list can contain only statements that produce an update count.

The list, which is associated with a `Statement` object at its creation, is initially empty. You can add SQL commands to this list with the method `addBatch` and empty it with the method `clearBatch`. When you have finished adding statements to the list, call the method `executeBatch` to send them all to the database to be executed as a unit, or batch.

For example, the following method `CoffeesTable.batchUpdate` adds four rows to the `COFFEES` table with a batch update:

```
public void batchUpdate() throws SQLException {

    Statement stmt = null;
    try {
        this.con.setAutoCommit(false);
        stmt = this.con.createStatement();

        stmt.addBatch(
            "INSERT INTO COFFEES " +
            "VALUES('Amaretto', 49, 9.99, 0, 0)");

        stmt.addBatch(
            "INSERT INTO COFFEES " +
            "VALUES('Hazelnut', 49, 9.99, 0, 0)");

        stmt.addBatch(
            "INSERT INTO COFFEES " +
            "VALUES('Amaretto_decaf', 49, " +
            "10.99, 0, 0)");

        stmt.addBatch(
            "INSERT INTO COFFEES " +
            "VALUES('Hazelnut_decaf', 49, " +
            "10.99, 0, 0)");

        int [] updateCounts = stmt.executeBatch();
        this.con.commit();

    } catch(BatchUpdateException b) {
        JDBCTutorialUtilities.printBatchUpdateException(b);
    } catch(SQLException ex) {
        JDBCTutorialUtilities.printSQLException(ex);
    } finally {
        if (stmt != null) { stmt.close(); }
        this.con.setAutoCommit(true);
    }
}
```

The following line disables auto-commit mode for the `Connection` object con so that the transaction will not be automatically committed or rolled back when the method `executeBatch` is called.

```
this.con.setAutoCommit(false);
```

To allow for correct error handling, you should always disable auto-commit mode before beginning a batch update.

The method `Statement.addBatch` adds a command to the list of commands associated with the `Statement` object `stmt`. In this example, these commands are all `INSERT INTO` statements, each one adding a row consisting of five column values. The values for the columns `COF_NAME` and `PRICE` are the name of the coffee and its price, respectively. The second value in each row is 49 because that is the identification number for the supplier, Superior Coffee. The last two values, the entries for the columns `SALES` and `TOTAL`, all start out being zero because there have been no sales yet. (`SALES` is the number of pounds of this row's coffee sold in the current week; `TOTAL` is the total of all the cumulative sales of this coffee.)

The following line sends the four SQL commands that were added to its list of commands to the database to be executed as a batch:

```
int [] updateCounts = stmt.executeBatch();
```

Note that `stmt` uses the method `executeBatch` to send the batch of insertions, not the method `executeUpdate`, which sends only one command and returns a single update count. The DBMS executes the commands in the order in which they were added to the list of commands, so it will first

add the row of values for Amaretto, then add the row for Hazelnut, then Amaretto decaf, and finally Hazelnut decaf. If all four commands execute successfully, the DBMS will return an update count for each command in the order in which it was executed. The update counts that indicate how many rows were affected by each command are stored in the array `updateCounts`.
If all four of the commands in the batch are executed successfully, `updateCounts` will contain four values, all of which are 1 because an insertion affects one row. The list of commands associated with `stmt` will now be empty because the four commands added previously were sent to the database when `stmt` called the method `executeBatch`. You can at any time explicitly empty this list of commands with the method `clearBatch`.
The `Connection.commit` method makes the batch of updates to the `COFFEES` table permanent. This method needs to be called explicitly because the auto-commit mode for this connection was disabled previously.
The following line enables auto-commit mode for the current `Connection` object.

```
this.con.setAutoCommit(true);
```

Now each statement in the example will automatically be committed after it is executed, and it no longer needs to invoke the method `commit`.

Performing Parameterized Batch Update

It is also possible to have a parameterized batch update, as shown in the following code fragment, where `con` is a `Connection` object:

```
con.setAutoCommit(false);
PreparedStatement pstmt = con.prepareStatement(
                          "INSERT INTO COFFEES VALUES( " +
                          "?, ?, ?, ?, ?)");
pstmt.setString(1, "Amaretto");
pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

pstmt.setString(1, "Hazelnut");
pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

// ... and so on for each new
// type of coffee

int [] updateCounts = pstmt.executeBatch();
con.commit();
con.setAutoCommit(true);
```

Handling Batch Update Exceptions

You will get a `BatchUpdateException` when you call the method `executeBatch` if (1) one of the SQL statements you added to the batch produces a result set (usually a query) or (2) one of the SQL statements in the batch does not execute successfully for some other reason.
You should not add a query (a `SELECT` statement) to a batch of SQL commands because the method `executeBatch`, which returns an array of update counts, expects an update count from each SQL statement that executes successfully. This means that only commands that return an update count (commands such as `INSERT INTO`, `UPDATE`, `DELETE`) or that return 0 (such as `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`) can be successfully executed as a batch with the `executeBatch` method.
A `BatchUpdateException` contains an array of update counts that is similar to the array returned by the method `executeBatch`. In both cases, the update counts are in the same order as the commands that produced them. This tells you how many commands in the batch executed successfully and which ones they are. For example, if five commands executed successfully, the array will contain five numbers: the first one being the update count for the first command, the second one being the update count for the second command, and so on.

`BatchUpdateException` is derived from `SQLException`. This means that you can use all of the methods available to an `SQLException` object with it. The following method, `JDBCTutorialUtilities.printBatchUpdateException` prints all of the `SQLException` information plus the update counts contained in a `BatchUpdateException` object. Because `BatchUpdateException.getUpdateCounts` returns an array of `int`, the code uses a `for` loop to print each of the update counts:

```
public static void printBatchUpdateException(BatchUpdateException b) {

    System.err.println("----BatchUpdateException----");
    System.err.println("SQLState:  " + b.getSQLState());
    System.err.println("Message:  " + b.getMessage());
    System.err.println("Vendor:  " + b.getErrorCode());
    System.err.print("Update counts:  ");
    int [] updateCounts = b.getUpdateCounts();

    for (int i = 0; i < updateCounts.length; i++) {
        System.err.print(updateCounts[i] + "   ");
    }
}
```

Inserting Rows in ResultSet Objects

Note: Not all JDBC drivers support inserting new rows with the `ResultSet` interface. If you attempt to insert a new row and your JDBC driver database does not support this feature, a `SQLFeatureNotSupportedException` exception is thrown.

The following method, `CoffeesTable.insertRow`, inserts a row into the `COFFEES` through a `ResultSet` object:

```
public void insertRow(String coffeeName, int supplierID,
                      float price, int sales, int total)
    throws SQLException {

    Statement stmt = null;
    try {
        stmt = con.createStatement(
            ResultSet.TYPE_SCROLL_SENSITIVE
            ResultSet.CONCUR_UPDATABLE);

        ResultSet uprs = stmt.executeQuery(
            "SELECT * FROM " + dbName +
            ".COFFEES");

        uprs.moveToInsertRow();
        uprs.updateString("COF_NAME", coffeeName);
        uprs.updateInt("SUP_ID", supplierID);
        uprs.updateFloat("PRICE", price);
        uprs.updateInt("SALES", sales);
        uprs.updateInt("TOTAL", total);

        uprs.insertRow();
        uprs.beforeFirst();
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

This example calls the `Connection.createStatement` method with two arguments, `ResultSet.TYPE_SCROLL_SENSITIVE` and `ResultSet.CONCUR_UPDATABLE`. The first value enables the cursor of the `ResultSet` object to be moved both forward and backward. The second value, `ResultSet.CONCUR_UPDATABLE`, is required if you want to insert rows into a `ResultSet` object; it specifies that it can be updatable.

The same stipulations for using strings in getter methods also apply to updater methods.

The method `ResultSet.moveToInsertRow` moves the cursor to the insert row. The insert row is a special row associated with an updatable result set. It is essentially a buffer where a new row can be constructed by calling the updater methods prior to inserting the row into the result set. For example, this method calls the method `ResultSet.updateString` to update the insert row's `COF_NAME` column to `Kona`.

The method `ResultSet.insertRow` inserts the contents of the insert row into the `ResultSet` object and into the database.

Note: After inserting a row with the `ResultSet.insertRow`, you should move the cursor to a row other than the insert row. For example, this example moves it to before the first row in the result set with the method `ResultSet.beforeFirst`. Unexpected results can occur if another part of your application uses the same result set and the cursor is still pointing to the insert row.

Using Prepared Statements

This page covers the following topics:

Overview of Prepared Statements

Sometimes it is more convenient to use a `PreparedStatement` object for sending SQL statements to the database. This special type of statement is derived from the more general class, `Statement`, that you already know.

If you want to execute a `Statement` object many times, it usually reduces execution time to use a `PreparedStatement` object instead.

The main feature of a `PreparedStatement` object is that, unlike a `Statement` object, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled. As a result, the `PreparedStatement` object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the `PreparedStatement` is executed, the DBMS can just run the `PreparedStatement` SQL statement without having to compile it first.

Although `PreparedStatement` objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it. Examples of this are in the following sections.

The following method, `CoffeesTable.updateCoffeeSales`, stores the number of pounds of coffee sold in the current week in the `SALES` column for each type of coffee, and updates the total number of pounds of coffee sold in the `TOTAL` column for each type of coffee:

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)
    throws SQLException {

    PreparedStatement updateSales = null;
    PreparedStatement updateTotal = null;

    String updateString =
        "update " + dbName + ".COFFEES " +
        "set SALES = ? where COF_NAME = ?";

    String updateStatement =
        "update " + dbName + ".COFFEES " +
        "set TOTAL = TOTAL + ? " +
        "where COF_NAME = ?";

    try {
        con.setAutoCommit(false);
        updateSales = con.prepareStatement(updateString);
        updateTotal = con.prepareStatement(updateStatement);

        for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
            updateSales.setInt(1, e.getValue().intValue());
            updateSales.setString(2, e.getKey());
            updateSales.executeUpdate();
            updateTotal.setInt(1, e.getValue().intValue());
            updateTotal.setString(2, e.getKey());
            updateTotal.executeUpdate();
            con.commit();
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
        if (con != null) {
            try {
                System.err.print("Transaction is being rolled back");
                con.rollback();
            } catch(SQLException excep) {
                JDBCTutorialUtilities.printSQLException(excep);
            }
        }
    } finally {
        if (updateSales != null) {
            updateSales.close();
        }
        if (updateTotal != null) {
            updateTotal.close();
        }
        con.setAutoCommit(true);
    }
}
```

Creating a PreparedStatement Object

The following creates a `PreparedStatement` object that takes two input parameters:

```
String updateString =
    "update " + dbName + ".COFFEES " +
    "set SALES = ? where COF_NAME = ?";
updateSales = con.prepareStatement(updateString);
```

Supplying Values for PreparedStatement Parameters

You must supply values in place of the question mark placeholders (if there are any) before you can execute a `PreparedStatement` object. Do this by calling one of the setter methods defined in the `PreparedStatement` class. The following statements supply the two question mark placeholders in the `PreparedStatement` named `updateSales`:

```
updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());
```

The first argument for each of these setter methods specifies the question mark placeholder. In this example, `setInt` specifies the first placeholder and `setString` specifies the second placeholder.

After a parameter has been set with a value, it retains that value until it is reset to another value, or the method `clearParameters` is called. Using the `PreparedStatement` object `updateSales`, the following code fragment illustrates reusing a prepared statement after resetting the value of one of its parameters and leaving the other one the same:

```
// changes SALES column of French Roast
//row to 100

updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();

// changes SALES column of Espresso row to 100
// (the first parameter stayed 100, and the second
// parameter was reset to "Espresso")

updateSales.setString(2, "Espresso");
updateSales.executeUpdate();
```

Using Loops to Set Values

You can often make coding easier by using a `for` loop or a `while` loop to set values for input parameters.

The `CoffeesTable.updateCoffeeSales` method uses a for-each loop to repeatedly set values in the `PreparedStatement` objects `updateSales` and `updateTotal`:

```
for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {

    updateSales.setInt(1, e.getValue().intValue());
    updateSales.setString(2, e.getKey());

    // ...
}
```

The method `CoffeesTable.updateCoffeeSales` takes one argument, `HashMap`. Each element in the `HashMap` argument contains the name of one type of coffee and the number of pounds of that type of coffee sold during the current week. The for-each loop iterates through each element of the `HashMap` argument and sets the appropriate question mark placeholders in `updateSales` and `updateTotal`.

Executing PreparedStatement Objects

As with `Statement` objects, to execute a `PreparedStatement` object, call an execute statement: `executeQuery` if the query returns only one `ResultSet` (such as a `SELECT` SQL statement), `executeUpdate` if the query does not return a `ResultSet` (such as an `UPDATE` SQL statement), or `execute` if the query might return more than one `ResultSet` object. Both `PreparedStatement` objects i n `CoffeesTable.updateCoffeeSales` contain `UPDATE` SQL statements, so both are executed by calling `executeUpdate`:

```
updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());
updateSales.executeUpdate();

updateTotal.setInt(1, e.getValue().intValue());
updateTotal.setString(2, e.getKey());
updateTotal.executeUpdate();
con.commit();
```

No arguments are supplied to `executeUpdate` when they are used to execute `updateSales` and `updateTotals`; both `PreparedStatement` objects already contain the SQL statement to be executed.

Note: At the beginning of `CoffeesTable.updateCoffeeSales`, the auto-commit mode is set to false:

```
con.setAutoCommit(false);
```

Consequently, no SQL statements are committed until the method `commit` is called. For more information about the auto-commit mode, see Transactions.

Return Values for the executeUpdate Method

Whereas `executeQuery` returns a `ResultSet` object containing the results of the query sent to the DBMS, the return value for `executeUpdate` is an `int` value that indicates how many rows of a table were updated. For instance, the following code shows the return value of `executeUpdate` being assigned to the variable `n`:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it
```

The table `COFFEES` is updated; the value 50 replaces the value in the column `SALES` in the row for `Espresso`. That update affects one row in the table, so `n` is equal to 1.

When the method `executeUpdate` is used to execute a DDL (data definition language) statement, such as in creating a table, it returns the `int` value of 0. Consequently, in the following code fragment, which executes the DDL statement used to create the table `COFFEES`, `n` is assigned a value of 0:

```
// n = 0
int n = executeUpdate(createTableCoffees);
```

Note that when the return value for `executeUpdate` is 0, it can mean one of two things:

- The statement executed was an update statement that affected zero rows.
- The statement executed was a DDL statement.

Using Transactions

There are times when you do not want one statement to take effect unless another one completes. For example, when the proprietor of The Coffee Break updates the amount of coffee sold each week, the proprietor will also want to update the total amount sold to date. However, the amount sold per week and the total amount sold should be updated at the same time; otherwise, the data will be inconsistent. The way to be sure that either both actions occur or neither action occurs is to use a transaction. A transaction is a set of one or more statements that is executed as a unit, so either all of the statements are executed, or none of the statements is executed.
This page covers the following topics

Disabling Auto-Commit Mode

When a connection is created, it is in auto-commit mode. This means that each individual SQL statement is treated as a transaction and is automatically committed right after it is executed. (To be more precise, the default is for a SQL statement to be committed when it is completed, not when it is executed. A statement is completed when all of its result sets and update counts have been retrieved. In almost all cases, however, a statement is completed, and therefore committed, right after it is executed.)

The way to allow two or more statements to be grouped into a transaction is to disable the auto-commit mode. This is demonstrated in the following code, where `con` is an active connection:

```
con.setAutoCommit(false);
```

Committing Transactions

After the auto-commit mode is disabled, no SQL statements are committed until you call the method `commit` explicitly. All statements executed after the previous call to the method `commit` are included in the current transaction and committed together as a unit. The following method, `CoffeesTable.updateCoffeeSales`, in which `con` is an active connection, illustrates a transaction:

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)
    throws SQLException {

    PreparedStatement updateSales = null;
    PreparedStatement updateTotal = null;

    String updateString =
        "update " + dbName + ".COFFEES " +
        "set SALES = ? where COF_NAME = ?";

    String updateStatement =
        "update " + dbName + ".COFFEES " +
        "set TOTAL = TOTAL + ? " +
        "where COF_NAME = ?";

    try {
        con.setAutoCommit(false);
        updateSales = con.prepareStatement(updateString);
        updateTotal = con.prepareStatement(updateStatement);

        for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
            updateSales.setInt(1, e.getValue().intValue());
            updateSales.setString(2, e.getKey());
            updateSales.executeUpdate();
            updateTotal.setInt(1, e.getValue().intValue());
            updateTotal.setString(2, e.getKey());
            updateTotal.executeUpdate();
            con.commit();
        }
    } catch (SQLException e ) {
        JDBCTutorialUtilities.printSQLException(e);
        if (con != null) {
            try {
                System.err.print("Transaction is being rolled back");
                con.rollback();
            } catch(SQLException excep) {
                JDBCTutorialUtilities.printSQLException(excep);
            }
        }
    } finally {
        if (updateSales != null) {
            updateSales.close();
        }
        if (updateTotal != null) {
            updateTotal.close();
        }
        con.setAutoCommit(true);
    }
}
```

In this method, the auto-commit mode is disabled for the connection `con`, which means that the two prepared statements `updateSales` and `updateTotal` are committed together when the method `commit` is called. Whenever the `commit` method is called (either automatically when auto-commit mode is enabled or explicitly when it is disabled), all changes resulting from statements in the transaction are made permanent. In this case, that means that the `SALES` and `TOTAL` columns for Colombian coffee have been changed to `50` (if `TOTAL` had been `0` previously) and will retain this value until they are changed with another update statement.

The statement `con.setAutoCommit(true);` enables auto-commit mode, which means that each statement is once again committed automatically when it is completed. Then, you are back to the default state where you do not have to call the method `commit` yourself. It is advisable to disable the auto-commit mode only during the transaction mode. This way, you avoid holding database locks for multiple statements, which increases the likelihood of conflicts with other users.

Using Transactions to Preserve Data Integrity

In addition to grouping statements together for execution as a unit, transactions can help to preserve the integrity of the data in a table. For instance, imagine that an employee was supposed to enter new coffee prices in the table `COFFEES` but delayed doing it for a few days. In the meantime, prices rose, and today the owner is in the process of entering the higher prices. The employee finally gets around to entering the now outdated prices at the same time that the owner is trying to update the table. After inserting the outdated prices, the employee realizes that they are no longer valid and calls the `Connection` method `rollback` to undo their effects. (The method `rollback` aborts a transaction and restores values to what they were before the attempted update.) At the same time, the owner is executing a `SELECT` statement and printing the new prices. In this situation, it is possible that the owner will print a price that had been rolled back to its previous value, making the printed price incorrect.

This kind of situation can be avoided by using transactions, providing some level of protection against conflicts that arise when two users access data at the same time.

To avoid conflicts during a transaction, a DBMS uses locks, mechanisms for blocking access by others to the data that is being accessed by the transaction. (Note that in auto-commit mode, where each statement is a transaction, locks are held for only one statement.) After a lock is set, it remains in force until the transaction is committed or rolled back. For example, a DBMS could lock a row of a table until updates to it have been committed. The effect of this lock would be to prevent a user from getting a dirty read, that is, reading a value before it is made permanent. (Accessing an updated value that has not been committed is considered a *dirty read* because it is possible for that value to be rolled back to its previous value. If you read a value that is later rolled back, you will have read an invalid value.)

How locks are set is determined by what is called a transaction isolation level, which can range from not supporting transactions at all to supporting transactions that enforce very strict access rules.

One example of a transaction isolation level is `TRANSACTION_READ_COMMITTED`, which will not allow a value to be accessed until after it has been committed. In other words, if the transaction isolation level is set to `TRANSACTION_READ_COMMITTED`, the DBMS does not allow dirty reads to occur. The interface `Connection` includes five values that represent the transaction isolation levels you can use in JDBC:

Isolation Level	Transactions	Dirty Reads	Non-Repeatable Reads	Phantom Reads
`TRANSACTION_NONE`	Not supported	*Not applicable*	*Not applicable*	*Not applicable*
`TRANSACTION_READ_COMMITTED`	Supported	Prevented	Allowed	Allowed
`TRANSACTION_READ_UNCOMMITTED`	Supported	Allowed	Allowed	Allowed
`TRANSACTION_REPEATABLE_READ`	Supported	Prevented	Prevented	Allowed
`TRANSACTION_SERIALIZABLE`	Supported	Prevented	Prevented	Prevented

A *non-repeatable read* occurs when transaction A retrieves a row, transaction B subsequently updates the row, and transaction A later retrieves the same row again. Transaction A retrieves the same row twice but sees different data.

A *phantom read* occurs when transaction A retrieves a set of rows satisfying a given condition, transaction B subsequently inserts or updates a row such that the row now meets the condition in transaction A, and transaction A later repeats the conditional retrieval. Transaction A now sees an

additional row. This row is referred to as a phantom.

Usually, you do not need to do anything about the transaction isolation level; you can just use the default one for your DBMS. The default transaction isolation level depends on your DBMS. For example, for Java DB, it is `TRANSACTION_READ_COMMITTED`. JDBC allows you to find out what transaction isolation level your DBMS is set to (using the `Connection` method `getTransactionIsolation`) and also allows you to set it to another level (using the `Connection` method `setTransactionIsolation`).

Note: A JDBC driver might not support all transaction isolation levels. If a driver does not support the isolation level specified in an invocation of `setTransactionIsolation`, the driver can substitute a higher, more restrictive transaction isolation level. If a driver cannot substitute a higher transaction level, it throws a `SQLException`. Use the method `DatabaseMetaData.supportsTransactionIsolationLevel` to determine whether or not the driver supports a given level.

Setting and Rolling Back to Savepoints

The method `Connection.setSavepoint`, sets a `Savepoint` object within the current transaction. The `Connection.rollback` method is overloaded to take a `Savepoint` argument.
The following method, `CoffeesTable.modifyPricesByPercentage`, raises the price of a particular coffee by a percentage, `priceModifier`. However, if the new price is greater than a specified price, `maximumPrice`, then the price is reverted to the original price:

```
public void modifyPricesByPercentage(
    String coffeeName,
    float priceModifier,
    float maximumPrice)
    throws SQLException {

    con.setAutoCommit(false);

    Statement getPrice = null;
    Statement updatePrice = null;
    ResultSet rs = null;
    String query =
        "SELECT COF_NAME, PRICE FROM COFFEES " +
        "WHERE COF_NAME = '" + coffeeName + "'";

    try {
        Savepoint save1 = con.setSavepoint();
        getPrice = con.createStatement(
                       ResultSet.TYPE_SCROLL_INSENSITIVE,
                       ResultSet.CONCUR_READ_ONLY);
        updatePrice = con.createStatement();

        if (!getPrice.execute(query)) {
            System.out.println(
                "Could not find entry " +
                "for coffee named " +
                coffeeName);
        } else {
            rs = getPrice.getResultSet();
            rs.first();
            float oldPrice = rs.getFloat("PRICE");
            float newPrice = oldPrice + (oldPrice * priceModifier);
            System.out.println(
                "Old price of " + coffeeName +
                " is " + oldPrice);

            System.out.println(
                "New price of " + coffeeName +
                " is " + newPrice);

            System.out.println(
                "Performing update...");

            updatePrice.executeUpdate(
                "UPDATE COFFEES SET PRICE = " +
                newPrice +
                " WHERE COF_NAME = '" +
                coffeeName + "'");

            System.out.println(
                "\nCOFFEES table after " +
                "update:");

            CoffeesTable.viewTable(con);

            if (newPrice > maximumPrice) {
                System.out.println(
                    "\nThe new price, " +
                    newPrice +
                    ", is greater than the " +
                    "maximum price, " +
                    maximumPrice +
                    ". Rolling back the " +
                    "transaction...");

                con.rollback(save1);

                System.out.println(
                    "\nCOFFEES table " +
                    "after rollback:");

                CoffeesTable.viewTable(con);
            }
            con.commit();
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (getPrice != null) { getPrice.close(); }
        if (updatePrice != null) {
            updatePrice.close();
        }
        con.setAutoCommit(true);
    }
}
```

The following statement specifies that the cursor of the `ResultSet` object generated from the `getPrice` query is closed when the `commit` method is called. Note that if your DBMs does not support `ResultSet.CLOSE_CURSORS_AT_COMMIT`, then this constant is ignored:

```
getPrice = con.prepareStatement(query, ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

The method begins by creating a `Savepoint` with the following statement:

```
Savepoint save1 = con.setSavepoint();
```

The method checks if the new price is greater than the `maximumPrice` value. If so, the method rolls back the transaction with the following statement:

```
con.rollback(save1);
```

Consequently, when the method commits the transaction by calling the `Connection.commit` method, it will not commit any rows whose associated `Savepoint` has been rolled back; it will commit all the other updated rows.

Releasing Savepoints

The method `Connection.releaseSavepoint` takes a `Savepoint` object as a parameter and removes it from the current transaction.
After a savepoint has been released, attempting to reference it in a rollback operation causes a `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.

When to Call Method rollback

As mentioned earlier, calling the method `rollback` terminates a transaction and returns any values that were modified to their previous values. If you are trying to execute one or more statements in a transaction and get a `SQLException`, call the method `rollback` to end the transaction and start the transaction all over again. That is the only way to know what has been committed and what has not been committed. Catching a `SQLException` tells you that something is wrong, but it does not tell you what was or was not committed. Because you cannot count on the fact that nothing was committed, calling the method `rollback` is the only way to be certain.

The method `CoffeesTable.updateCoffeeSales` demonstrates a transaction and includes a `catch` block that invokes the method `rollback`. If the application continues and uses the results of the transaction, this call to the `rollback` method in the `catch` block prevents the use of possibly incorrect data.

Using RowSet Objects

A JDBC `RowSet` object holds tabular data in a way that makes it more flexible and easier to use than a result set.
Oracle has defined five `RowSet` interfaces for some of the more popular uses of a `RowSet`, and standard reference are available for these `RowSet` interfaces. In this tutorial you will learn how to use these reference implementations.
These versions of the `RowSet` interface and their implementations have been provided as a convenience for programmers. Programmers are free write their own versions of the `javax.sql.RowSet` interface, to extend the implementations of the five `RowSet` interfaces, or to write their own implementations. However, many programmers will probably find that the standard reference implementations already fit their needs and will use them as is.
This section introduces you to the `RowSet` interface and the following interfaces that extend this interface:

- `JdbcRowSet`
- `CachedRowSet`
- `WebRowSet`
- `JoinRowSet`
- `FilteredRowSet`

The following topics are covered:

What Can RowSet Objects Do?

All `RowSet` objects are derived from the `ResultSet` interface and therefore share its capabilities. What makes JDBC `RowSet` objects special is that they add these new capabilities:

- Function as JavaBeans Component
- Add Scrollability or Updatability

Function as JavaBeans Component

All `RowSet` objects are JavaBeans components. This means that they have the following:

- Properties
- JavaBeans Notification Mechanism

Properties

All `RowSet` objects have properties. A property is a field that has corresponding getter and setter methods. Properties are exposed to builder tools (such as those that come with the IDEs JDveloper and Eclipse) that enable you to visually manipulate beans. For more information, see the Properties lesson in the JavaBeans trail.

JavaBeans Notification Mechanism

`RowSet` objects use the JavaBeans event model, in which registered components are notified when certain events occur. For all `RowSet` objects, three events trigger notifications:

- A cursor movement
- The update, insertion, or deletion of a row
- A change to the entire `RowSet` contents

The notification of an event goes to all *listeners*, components that have implemented the `RowSetListener` interface and have had themselves added to the `RowSet` object's list of components to be notified when any of the three events occurs.
A listener could be a GUI component such as a bar graph. If the bar graph is tracking data in a `RowSet` object, the listener would want to know the new data values whenever the data changed. The listener would therefore implement the `RowSetListener` methods to define what it will do when a particular event occurs. Then the listener also must be added to the `RowSet` object's list of listeners. The following line of code registers the bar graph component `bg` with the `RowSet` object `rs`.

```
rs.addListener(bg);
```

Now `bg` will be notified each time the cursor moves, a row is changed, or all of `rs` gets new data.

Add Scrollability or Updatability

Some DBMSs do not support result sets that can be scrolled (scrollable), and some do not support result sets that can be updated (updatable). If a driver for that DBMS does not add the ability to scroll or update result sets, you can use a `RowSet` object to do it. A `RowSet` object is scrollable and updatable

by default, so by populating a `RowSet` object with the contents of a result set, you can effectively make the result set scrollable and updatable.

Kinds of RowSet Objects

A `RowSet` object is considered either connected or disconnected. A *connected* `RowSet` object uses a JDBC driver to make a connection to a relational database and maintains that connection throughout its life span. A *disconnected* `RowSet` object makes a connection to a data source only to read in data from a `ResultSet` object or to write data back to the data source. After reading data from or writing data to its data source, the `RowSet` object disconnects from it, thus becoming "disconnected." During much of its life span, a disconnected `RowSet` object has no connection to its data source and operates independently. The next two sections tell you what being connected or disconnected means in terms of what a `RowSet` object can do.

Connected RowSet Objects

Only one of the standard `RowSet` implementations is a connected `RowSet` object: `JdbcRowSet`. Always being connected to a database, a `JdbcRowSet` object is most similar to a `ResultSet` object and is often used as a wrapper to make an otherwise non-scrollable and read-only `ResultSet` object scrollable and updatable.
As a JavaBeans component, a `JdbcRowSet` object can be used, for example, in a GUI tool to select a JDBC driver. A `JdbcRowSet` object can be used this way because it is effectively a wrapper for the driver that obtained its connection to the database.

Disconnected RowSet Objects

The other four implementations are disconnected `RowSet` implementations. Disconnected `RowSet` objects have all the capabilities of connected `RowSet` objects plus they have the additional capabilities available only to disconnected `RowSet` objects. For example, not having to maintain a connection to a data source makes disconnected `RowSet` objects far more lightweight than a `JdbcRowSet` object or a `ResultSet` object. Disconnected `RowSet` objects are also serializable, and the combination of being both serializable and lightweight makes them ideal for sending data over a network. They can even be used for sending data to thin clients such as PDAs and mobile phones.
The `CachedRowSet` interface defines the basic capabilities available to all disconnected `RowSet` objects. The other three are extensions of the `CachedRowSet` interface, which provide more specialized capabilities. The following information shows how they are related:
A `CachedRowSet` object has all the capabilities of a `JdbcRowSet` object plus it can also do the following:

- Obtain a connection to a data source and execute a query
- Read the data from the resulting `ResultSet` object and populate itself with that data
- Manipulate data and make changes to data while it is disconnected
- Reconnect to the data source to write changes back to it
- Check for conflicts with the data source and resolve those conflicts

A `WebRowSet` object has all the capabilities of a `CachedRowSet` object plus it can also do the following:

- Write itself as an XML document
- Read an XML document that describes a `WebRowSet` object

A `JoinRowSet` object has all the capabilities of a `WebRowSet` object (and therefore also those of a `CachedRowSet` object) plus it can also do the following:

- Form the equivalent of a `SQL JOIN` without having to connect to a data source

A `FilteredRowSet` object likewise has all the capabilities of a `WebRowSet` object (and therefore also a `CachedRowSet` object) plus it can also do the following:

- Apply filtering criteria so that only selected data is visible. This is equivalent to executing a query on a `RowSet` object without having to use a query language or connect to a data source.

Using JdbcRowSet Objects

A `JdbcRowSet` object is an enhanced `ResultSet` object. It maintains a connection to its data source, just as a `ResultSet` object does. The big difference is that it has a set of properties and a listener notification mechanism that make it a JavaBeans component.
One of the main uses of a `JdbcRowSet` object is to make a `ResultSet` object scrollable and updatable when it does not otherwise have those capabilities.
This section covers the following topics:

Creating JdbcRowSet Objects

You can create a `JdbcRowSet` object in various ways:

- By using the reference implementation constructor that takes a `ResultSet` object
- By using the reference implementation constructor that takes a `Connection` object
- By using the reference implementation default constructor
- By using an instance of `RowSetFactory`, which is created from the class `RowSetProvider`

Note: Alternatively, you can use the constructor from the `JdbcRowSet` implementation of your JDBC driver. However, implementations of the `RowSet` interface will differ from the reference implementation. These implementations will have different names and constructors. For example, the Oracle JDBC driver's implementation of the `JdbcRowSet` interface is named `oracle.jdbc.rowset.OracleJDBCRowSet`.

Passing ResultSet Objects

The simplest way to create a `JdbcRowSet` object is to produce a `ResultSet` object and pass it to the `JdbcRowSetImpl` constructor. Doing this not only creates a `JdbcRowSet` object but also populates it with the data in the `ResultSet` object.
Note: The `ResultSet` object that is passed to the `JdbcRowSetImpl` constructor must be scrollable.
As an example, the following code fragment uses the `Connection` object `con` to create a `Statement` object, `stmt`, which then executes a query. The query produces the `ResultSet` object `rs`, which is passed to the constructor to create a new `JdbcRowSet` object initialized with the data in `rs`:

```
stmt = con.createStatement(
           ResultSet.TYPE_SCROLL_SENSITIVE,
           ResultSet.CONCUR_UPDATABLE);
rs = stmt.executeQuery("select * from COFFEES");
jdbcRs = new JdbcRowSetImpl(rs);
```

A `JdbcRowSet` object created with a `ResultSet` object serves as a wrapper for the `ResultSet` object. Because the `RowSet` object `rs` is scrollable and updatable, `jdbcRs` is also scrollable and updatable. If you have run the method `createStatement` without any arguments, `rs` would not be scrollable or updatable, and neither would `jdbcRs`.

Passing Connection Objects

The first statement in the following code excerpt from `JdbcRowSetSample` creates a `JdbcRowSet` object that connects to the database with the `Connection` object `con`:

```
jdbcRs = new JdbcRowSetImpl(con);
jdbcRs.setCommand("select * from COFFEES");
jdbcRs.execute();
```

The object `jdbcRs` contains no data until you specify a SQL statement with the method `setCommand`, then run the method `execute`.
The object `jdbcRs` is scrollable and updatable; by default, `JdbcRowSet` and all other `RowSet` objects are scrollable and updatable unless otherwise specified. See Default JdbcRowSet Objects for more information about `JdbcRowSet` properties you can specify.

Using the Default Constructor

The first statement in the following code excerpt creates an empty `JdbcRowSet` object.

```
public void createJdbcRowSet(String username, String password) {

    jdbcRs = new JdbcRowSetImpl();
    jdbcRs.setCommand("select * from COFFEES");
    jdbcRs.setUrl("jdbc:myDriver:myAttribute");
```

```
    jdbcRs.setUsername(username);
    jdbcRs.setPassword(password);
    jdbcRs.execute();
    // ...
}
```

The object `jdbcRs` contains no data until you specify a SQL statement with the method `setCommand`, specify how the `JdbcResultSet` object connects the database, and then run the method `execute`.
All of the reference implementation constructors assign the default values for the properties listed in the section Default JdbcRowSet Objects.

Using the RowSetFactory Interface

With RowSet 1.1, which is part of Java SE 7 and later, you can use an instance of `RowSetFactory` to create a `JdbcRowSet` object. For example, the following code excerpt uses an instance of the `RowSetFactory` interface to create the `JdbcRowSet` object, `jdbcRs`:

```
public void createJdbcRowSetWithRowSetFactory(
    String username, String password)
    throws SQLException {

    RowSetFactory myRowSetFactory = null;
    JdbcRowSet jdbcRs = null;
    ResultSet rs = null;
    Statement stmt = null;

    try {
        myRowSetFactory = RowSetProvider.newFactory();
        jdbcRs = myRowSetFactory.createJdbcRowSet();

        jdbcRs.setUrl("jdbc:myDriver:myAttribute");
        jdbcRs.setUsername(username);
        jdbcRs.setPassword(password);

        jdbcRs.setCommand("select * from COFFEES");
        jdbcRs.execute();

        // ...
    }
}
```

The following statement creates the `RowSetProvider` object `myRowSetFactory` with the default `RowSetFactory` implementation, `com.sun.rowset.RowSetFactoryImpl`:

```
myRowSetFactory = RowSetProvider.newFactory();
```

Alternatively, if your JDBC driver has its own `RowSetFactory` implementation, you may specify it as an argument of the `newFactory` method.
The following statements create the `JdbcRowSet` object `jdbcRs` and configure its database connection properties:

```
jdbcRs = myRowSetFactory.createJdbcRowSet();
jdbcRs.setUrl("jdbc:myDriver:myAttribute");
jdbcRs.setUsername(username);
jdbcRs.setPassword(password);
```

The `RowSetFactory` interface contains methods to create the different types of `RowSet` implementations available in RowSet 1.1 and later:

- `createCachedRowSet`
- `createFilteredRowSet`
- `createJdbcRowSet`
- `createJoinRowSet`
- `createWebRowSet`

Default JdbcRowSet Objects

When you create a `JdbcRowSet` object with the default constructor, the new `JdbcRowSet` object will have the following properties:

- `type`: `ResultSet.TYPE_SCROLL_INSENSITIVE` (has a scrollable cursor)
- `concurrency`: `ResultSet.CONCUR_UPDATABLE` (can be updated)
- `escapeProcessing`: `true` (the driver will do escape processing; when escape processing is enabled, the driver will scan for any escape syntax and translate it into code that the particular database understands)
- `maxRows`: `0` (no limit on the number of rows)
- `maxFieldSize`: `0` (no limit on the number of bytes for a column value; applies only to columns that store `BINARY`, `VARBINARY`, `LONGVARBINARY`, `CHAR`, `VARCHAR`, and `LONGVARCHAR` values)
- `queryTimeout`: `0` (has no time limit for how long it takes to execute a query)
- `showDeleted`: `false` (deleted rows are not visible)
- `transactionIsolation`: `Connection.TRANSACTION_READ_COMMITTED` (reads only data that has been committed)
- `typeMap`: `null` (the type map associated with a `Connection` object used by this `RowSet` object is `null`)

The main thing you must remember from this list is that a `JdbcRowSet` and all other `RowSet` objects are scrollable and updatable unless you set different values for those properties.

Setting Properties

The section Default JdbcRowSet Objects lists the properties that are set by default when a new `JdbcRowSet` object is created. If you use the default constructor, you must set some additional properties before you can populate your new `JdbcRowSet` object with data.
In order to get its data, a `JdbcRowSet` object first needs to connect to a database. The following four properties hold information used in obtaining a connection to a database.

- `username`: the name a user supplies to a database as part of gaining access
- `password`: the user's database password
- `url`: the JDBC URL for the database to which the user wants to connect
- `dataSourceName`: the name used to retrieve a `DataSource` object that has been registered with a JNDI naming service

Which of these properties you set depends on how you are going to make a connection. The preferred way is to use a `DataSource` object, but it may not be practical for you to register a `DataSource` object with a JNDI naming service, which is generally done by a system administrator. Therefore, the code examples all use the `DriverManager` mechanism to obtain a connection, for which you use the `url` property and not the `dataSourceName` property.
Another property that you must set is the `command` property. This property is the query that determines what data the `JdbcRowSet` object will hold. For example, the following line of code sets the `command` property with a query that produces a `ResultSet` object containing all the data in the table `COFFEES`:

```
jdbcRs.setCommand("select * from COFFEES");
```

After you have set the `command` property and the properties necessary for making a connection, you are ready to populate the `jdbcRs` object with data by calling the `execute` method.

```
jdbcRs.execute();
```

The `execute` method does many things for you in the background:

- It makes a connection to the database using the values you assigned to the `url`, `username`, and `password` properties.
- It executes the query you set in the `command` property.
- It reads the data from the resulting `ResultSet` object into the `jdbcRs` object.

Using JdbcRowSet Objects

You update, insert, and delete a row in a `JdbcRowSet` object the same way you update, insert, and delete a row in an updatable `ResultSet` object. Similarly, you navigate a `JdbcRowSet` object the same way you navigate a scrollable `ResultSet` object.
The Coffee Break chain of coffee houses acquired another chain of coffee houses and now has a legacy database that does not support scrolling or updating of a result set. In other words, any `ResultSet` object produced by this legacy database does not have a scrollable cursor, and the data in it cannot be modified. However, by creating a `JdbcRowSet` object populated with the data from a `ResultSet` object, you can, in effect, make the `ResultSet` object scrollable and updatable.
As mentioned previously, a `JdbcRowSet` object is by default scrollable and updatable. Because its contents are identical to those in a `ResultSet` object, operating on the `JdbcRowSet` object is equivalent to operating on the `ResultSet` object itself. And because a `JdbcRowSet` object has an ongoing connection to the database, changes it makes to its own data are also made to the data in the database.
This section covers the following topics:

Navigating JdbcRowSet Objects

A `ResultSet` object that is not scrollable can use only the `next` method to move its cursor forward, and it can move the cursor only forward from the first row to the last row. A default `JdbcRowSet` object, however, can use all of the cursor movement methods defined in the `ResultSet` interface.
A `JdbcRowSet` object can call the method `next`, and it can also call any of the other `ResultSet` cursor movement methods. For example, the following lines of code move the cursor to the fourth row in the `jdbcRs` object and then back to the third row:

```
jdbcRs.absolute(4);
jdbcRs.previous();
```

The method `previous` is analogous to the method `next` in that it can be used in a `while` loop to traverse all of the rows in order. The difference is that you must move the cursor to a position after the last row, and `previous` moves the cursor toward the beginning.

Updating Column Values

You update data in a `JdbcRowSet` object the same way you update data in a `ResultSet` object.
Assume that the Coffee Break owner wants to raise the price for a pound of Espresso coffee. If the owner knows that Espresso is in the third row of the `jdbcRs` object, the code for doing this might look like the following:

```
jdbcRs.absolute(3);
jdbcRs.updateFloat("PRICE", 10.99f);
jdbcRs.updateRow();
```

The code moves the cursor to the third row and changes the value for the column `PRICE` to 10.99, and then updates the database with the new price.
Calling the method `updateRow` updates the database because `jdbcRs` has maintained its connection to the database. For disconnected `RowSet` objects, the situation is different.

Inserting Rows

If the owner of the Coffee Break chain wants to add one or more coffees to what he offers, the owner will need to add one row to the `COFFEES` table for each new coffee, as is done in the following code fragment from `JdbcRowSetSample`. Notice that because the `jdbcRs` object is always connected to the database, inserting a row into a `JdbcRowSet` object is the same as inserting a row into a `ResultSet` object: You move to the cursor to the insert row, use the appropriate updater method to set a value for each column, and call the method `insertRow`:

```
jdbcRs.moveToInsertRow();
jdbcRs.updateString("COF_NAME", "HouseBlend");
jdbcRs.updateInt("SUP_ID", 49);
jdbcRs.updateFloat("PRICE", 7.99f);
jdbcRs.updateInt("SALES", 0);
jdbcRs.updateInt("TOTAL", 0);
jdbcRs.insertRow();

jdbcRs.moveToInsertRow();
jdbcRs.updateString("COF_NAME", "HouseDecaf");
jdbcRs.updateInt("SUP_ID", 49);
jdbcRs.updateFloat("PRICE", 8.99f);
jdbcRs.updateInt("SALES", 0);
jdbcRs.updateInt("TOTAL", 0);
jdbcRs.insertRow();
```

When you call the method `insertRow`, the new row is inserted into the `jdbcRs` object and is also inserted into the database. The preceding code fragment goes through this process twice, so two new rows are inserted into the `jdbcRs` object and the database.

Deleting Rows

As is true with updating data and inserting a new row, deleting a row is just the same for a `JdbcRowSet` object as for a `ResultSet` object. The owner wants to discontinue selling French Roast decaffeinated coffee, which is the last row in the `jdbcRs` object. In the following lines of code, the first line moves the cursor to the last row, and the second line deletes the last row from the `jdbcRs` object and from the database:

```
jdbcRs.last();
jdbcRs.deleteRow();
```

Code Sample

The sample `JdbcRowSetSample` does the following:

- Creates a new `JdbcRowSet` object initialized with the `ResultSet` object that was produced by the execution of a query that retrieves all the rows in the `COFFEES` table
- Moves the cursor to the third row of the `COFFEES` table and updates the `PRICE` column in that row
- Inserts two new rows, one for `HouseBlend` and one for `HouseDecaf`
- Moves the cursor to the last row and deletes it

Using CachedRowSetObjects

A `CachedRowSet` object is special in that it can operate without being connected to its data source, that is, it is a *disconnected* `RowSet` object. It gets its name from the fact that it stores (caches) its data in memory so that it can operate on its own data rather than on the data stored in a database.
The `CachedRowSet` interface is the superinterface for all disconnected `RowSet` objects, so everything demonstrated here also applies to `WebRowSet`, `JoinRowSet`, and `FilteredRowSet` objects.
Note that although the data source for a `CachedRowSet` object (and the `RowSet` objects derived from it) is almost always a relational database, a `CachedRowSet` object is capable of getting data from any data source that stores its data in a tabular format. For example, a flat file or spreadsheet could be the source of data. This is true when the `RowSetReader` object for a disconnected `RowSet` object is implemented to read data from such a data source. The reference implementation of the `CachedRowSet` interface has a `RowSetReader` object that reads data from a relational database, so in this tutorial, the data source is always a database.
The following topics are covered:

Setting Up CachedRowSet Objects

Setting up a `CachedRowSet` object involves the following:

- Creating CachedRowSet Objects
- Setting CachedRowSet Properties
- Setting Key Columns

Creating CachedRowSet Objects

You can create a new `CachedRowSet` object in the different ways:

- Using the Default Constructor
- Using an instance of `RowSetFactory`, which is created from the class `RowSetProvider`: See Using the RowSetFactory Interface in Using JdbcRowSet Objects for more information.

Note: Alternatively, you can use the constructor from the `CachedRowSet` implementation of your JDBC driver. However, implementations of the `RowSet` interface will differ from the reference implementation. These implementations will have different names and constructors. For example, the Oracle JDBC driver's implementation of the `CachedRowSet` interface is named `oracle.jdbc.rowset.OracleCachedRowSet`.

Using the Default Constructor

One of the ways you can create a `CachedRowSet` object is by calling the default constructor defined in the reference implementation, as is done in the following line of code:

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

The object `crs` has the same default values for its properties that a `JdbcRowSet` object has when it is first created. In addition, it has been assigned an instance of the default `SyncProvider` implementation, `RIOptimisticProvider`.

A `SyncProvider` object supplies a `RowSetReader` object (a *reader*) and a `RowSetWriter` object (a *writer*), which a disconnected `RowSet` object needs in order to read data from its data source or to write data back to its data source. What a reader and writer do is explained later in the sections What Reader Does and What Writer Does. One thing to keep in mind is that readers and writers work entirely in the background, so the explanation of how they work is for your information only. Having some background on readers and writers should help you understand what some of the methods defined in the `CachedRowSet` interface do in the background.

Setting CachedRowSet Properties

Generally, the default values for properties are fine as they are, but you may change the value of a property by calling the appropriate setter method. There are some properties without default values that you must set yourself.

In order to get data, a disconnected `RowSet` object must be able to connect to a data source and have some means of selecting the data it is to hold. The following properties hold information necessary to obtain a connection to a database.

- `username`: The name a user supplies to a database as part of gaining access

- `password`: The user's database password
- `url`: The JDBC URL for the database to which the user wants to connect
- `datasourceName`: The name used to retrieve a DataSource object that has been registered with a JNDI naming service

Which of these properties you must set depends on how you are going to make a connection. The preferred way is to use a `DataSource` object, but it may not be practical for you to register a `DataSource` object with a JNDI naming service, which is generally done by a system administrator. Therefore, the code examples all use the `DriverManager` mechanism to obtain a connection, for which you use the `url` property and not the `datasourceName` property.
The following lines of code set the `username`, `password`, and `url` properties so that a connection can be obtained using the `DriverManager` class. (You will find the JDBC URL to set as the value for the `url` property in the documentation for your JDBC driver.)

```
public void setConnectionProperties(
    String username, String password) {
    crs.setUsername(username);
    crs.setPassword(password);
    crs.setUrl("jdbc:mySubprotocol:mySubname");
    // ...
```

Another property that you must set is the `command` property. In the reference implementation, data is read into a `RowSet` object from a `ResultSet` object. The query that produces that `ResultSet` object is the value for the `command` property. For example, the following line of code sets the `command` property with a query that produces a `ResultSet` object containing all the data in the table `MERCH_INVENTORY`:

```
crs.setCommand("select * from MERCH_INVENTORY");
```

Setting Key Columns

If you are going make any updates to the `crs` object and want those updates saved in the database, you must set one more piece of information: the key columns. Key columns are essentially the same as a primary key because they indicate one or more columns that uniquely identify a row. The difference is that a primary key is set on a table in the database, whereas key columns are set on a particular `RowSet` object. The following lines of code set the key columns for `crs` to the first column:

```
int [] keys = {1};
crs.setKeyColumns(keys);
```

The first column in the table `MERCH_INVENTORY` is `ITEM_ID`. It can serve as the key column because every item identifier is different and therefore uniquely identifies one row and only one row in the table `MERCH_INVENTORY`. In addition, this column is specified as a primary key in the definition of the `MERCH_INVENTORY` table. The method `setKeyColumns` takes an array to allow for the fact that it may take two or more columns to identify a row uniquely.
As a point of interest, the method `setKeyColumns` does not set a value for a property. In this case, it sets the value for the field `keyCols`. Key columns are used internally, so after setting them, you do nothing more with them. You will see how and when key columns are used in the section Using SyncResolver Objects.

Populating CachedRowSet Objects

Populating a disconnected `RowSet` object involves more work than populating a connected `RowSet` object. Fortunately, the extra work is done in the background. After you have done the preliminary work to set up the `CachedRowSet` object `crs`, the following line of code populates `crs`:

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

The data in `crs` is the data in the `ResultSet` object produced by executing the query in the command property.

What is different is that the `CachedRowSet` implementation for the `execute` method does a lot more than the `JdbcRowSet` implementation. Or more correctly, the `CachedRowSet` object's reader, to which the method execute delegates its tasks, does a lot more.

Every disconnected `RowSet` object has a `SyncProvider` object assigned to it, and this `SyncProvider` object is what provides the `RowSet` object's *reader* (a `RowSetReader` object). When the `crs` object was created, it was used as the default `CachedRowSetImpl` constructor, which, in addition to setting default values for properties, assigns an instance of the `RIOptimisticProvider` implementation as the default `SyncProvider` object.

What Reader Does

When an application calls the method `execute`, a disconnected `RowSet` object's reader works behind the scenes to populate the `RowSet` object with data. A newly created `CachedRowSet` object is not connected to a data source and therefore must obtain a connection to that data source in order to get data from it. The reference implementation of the default `SyncProvider` object (`RIOptimisticProvider`) provides a reader that obtains a connection by using the values set for the user name, password, and either the JDBC URL or the data source name, whichever was set more recently. Then the reader executes the query set for the command. It reads the data in the `ResultSet` object produced by the query, populating the `CachedRowSet` object with that data. Finally, the reader closes the connection.

Updating CachedRowSet Object

In the Coffee Break scenario, the owner wants to streamline operations. The owner decides to have employees at the warehouse enter inventory directly into a PDA (personal digital assistant), thereby avoiding the error-prone process of having a second person do the data entry. A `CachedRowSet` object is ideal in this situation because it is lightweight, serializable, and can be updated without a connection to the data source.

The owner will have the application development team create a GUI tool for the PDA that warehouse employees will use for entering inventory data. Headquarters will create a `CachedRowSet` object populated with the table showing the current inventory and send it using the Internet to the PDAs. When a warehouse employee enters data using the GUI tool, the tool adds each entry to an array, which the `CachedRowSet` object will use to perform the updates in the background. Upon completion of the inventory, the PDAs send their new data back to headquarters, where the data is uploaded to the main server.

This section covers the following topics:

Updating Column Values

Updating data in a `CachedRowSet` object is just the same as updating data in a `JdbcRowSet` object. For example, the following code fragment from `CachedRowSetSample.java` increments the value in the column `QUAN` by 1 in the row whose `ITEM_ID` column has an item identifier of `12345`:

```
while (crs.next()) {
    System.out.println(
        "Found item " + crs.getInt("ITEM_ID") +
        ": " + crs.getString("ITEM_NAME"));
    if (crs.getInt("ITEM_ID") == 1235) {
        int currentQuantity = crs.getInt("QUAN") + 1;
        System.out.println("Updating quantity to " +
          currentQuantity);
        crs.updateInt("QUAN", currentQuantity + 1);
        crs.updateRow();
        // Synchronizing the row
        // back to the DB
        crs.acceptChanges(con);
    }
```

Inserting and Deleting Rows

Just as with updating a column value, the code for inserting and deleting rows in a `CachedRowSet` object is the same as for a `JdbcRowSet` object.

The following excerpt from `CachedRowSetSample.java` inserts a new row into the `CachedRowSet` object `crs`:

```
crs.moveToInsertRow();
crs.updateInt("ITEM_ID", newItemId);
crs.updateString("ITEM_NAME", "TableCloth");
crs.updateInt("SUP_ID", 927);
crs.updateInt("QUAN", 14);
Calendar timeStamp;
timeStamp = new GregorianCalendar();
timeStamp.set(2006, 4, 1);
crs.updateTimestamp(
    "DATE_VAL",
    new Timestamp(timeStamp.getTimeInMillis()));
crs.insertRow();
crs.moveToCurrentRow();
```

If headquarters has decided to stop stocking a particular item, it would probably remove the row for that coffee itself. However, in the scenario, a warehouse employee using a PDA also has the capability of removing it. The following code fragment finds the row where the value in the `ITEM_ID` column is `12345` and deletes it from the `CachedRowSet` `crs`:

```
while (crs.next()) {
    if (crs.getInt("ITEM_ID") == 12345) {
```

```
        crs.deleteRow();
        break;
    }
}
```

Updating Data Sources

There is a major difference between making changes to a `JdbcRowSet` object and making changes to a `CachedRowSet` object. Because a `JdbcRowSet` object is connected to its data source, the methods `updateRow`, `insertRow`, and `deleteRow` can update both the `JdbcRowSet` object and the data source. In the case of a disconnected `RowSet` object, however, these methods update the data stored in the `CachedRowSet` object's memory but cannot affect the data source. A disconnected `RowSet` object must call the method `acceptChanges` in order to save its changes to the data source. In the inventory scenario, back at headquarters, an application will call the method `acceptChanges` to update the database with the new values for the column `QUAN`.

```
crs.acceptChanges();
```

What Writer Does

Like the method `execute`, the method `acceptChanges` does its work invisibly. Whereas the method `execute` delegates its work to the `RowSet` object's reader, the method `acceptChanges` delegates its tasks to the `RowSet` object's writer. In the background, the writer opens a connection to the database, updates the database with the changes made to the `RowSet` object, and then closes the connection.

Using Default Implementation

The difficulty is that a *conflict* can arise. A conflict is a situation in which another party has updated a value in the database that corresponds to a value that was updated in a `RowSet` object. Which value should persist in the database? What the writer does when there is a conflict depends on how it is implemented, and there are many possibilities. At one end of the spectrum, the writer does not even check for conflicts and just writes all changes to the database. This is the case with the `RIXMLProvider` implementation, which is used by a `WebRowSet` object. At the other end, the writer ensures that there are no conflicts by setting database locks that prevent others from making changes.
The writer for the `crs` object is the one provided by the default `SyncProvider` implementation, `RIOptimisticProvider`. The `RIOPtimisticProvider` implementation gets its name from the fact that it uses an optimistic concurrency model. This model assumes that there will be few, if any, conflicts and therefore sets no database locks. The writer checks to see if there are any conflicts, and if there is none, it writes the changes made to the `crs` object to the database, and those changes become persistent. If there are any conflicts, the default is not to write the new `RowSet` values to the database.
In the scenario, the default behavior works very well. Because no one at headquarters is likely to change the value in the `QUAN` column of `COF_INVENTORY`, there will be no conflicts. As a result, the values entered into the `crs` object at the warehouse will be written to the database and thus will be persistent, which is the desired outcome.

Using SyncResolver Objects

In other situations, however, it is possible for conflicts to exist. To accommodate these situations, the `RIOPtimisticProvider` implementation provides an option that lets you look at the values in conflict and decide which ones should be persistent. This option is the use of a `SyncResolver` object.

When the writer has finished looking for conflicts and has found one or more, it creates a `SyncResolver` object containing the database values that caused the conflicts. Next, the method `acceptChanges` throws a `SyncProviderException` object, which an application may catch and use to retrieve the `SyncResolver` object. The following lines of code retrieve the `SyncResolver` object `resolver`:

```
try {
    crs.acceptChanges();
} catch (SyncProviderException spe) {
    SyncResolver resolver = spe.getSyncResolver();
}
```

The object `resolver` is a `RowSet` object that replicates the `crs` object except that it contains only the values in the database that caused a conflict. All other column values are null.

With the `resolver` object, you can iterate through its rows to locate the values that are not null and are therefore values that caused a conflict. Then you can locate the value at the same position in the `crs` object and compare them. The following code fragment retrieves `resolver` and uses the `SyncResolver` method `nextConflict` to iterate through the rows that have conflicting values. The object `resolver` gets the status of each conflicting value, and if it is `UPDATE_ROW_CONFLICT`, meaning that the `crs` was attempting an update when the conflict occurred, the `resolver` object gets the row number of that value. Then the code moves the cursor for the `crs` object to the same row. Next, the code finds the column in that row of the `resolver` object that contains a conflicting value, which will be a value that is not null. After retrieving the value in that column from both the `resolver` and `crs` objects, you can compare the two and decide which one you want to become persistent. Finally, the code sets that value in both the `crs` object and the database using the method `setResolvedValue`, as shown in the following code:

```
try {
    crs.acceptChanges();
} catch (SyncProviderException spe) {
    SyncResolver resolver = spe.getSyncResolver();

    // value in crs
    Object crsValue;

    // value in the SyncResolver object
    Object resolverValue;

    // value to be persistent
    Object resolvedValue;

    while (resolver.nextConflict()) {
        if (resolver.getStatus() ==
            SyncResolver.UPDATE_ROW_CONFLICT) {
            int row = resolver.getRow();
            crs.absolute(row);
            int colCount =
                crs.getMetaData().getColumnCount();
            for (int j = 1; j <= colCount; j++) {
                if (resolver.getConflictValue(j)
                    != null) {
                    crsValue = crs.getObject(j);
                    resolverValue =
                        resolver.getConflictValue(j);

                    // ...
                    // compare crsValue and
                    // resolverValue to
                    // determine the value to be
                    // persistent

                    resolvedValue = crsValue;
                    resolver.setResolvedValue(
                        j, resolvedValue);
                }
            }
        }
    }
}
```

Notifying Listeners

Being a JavaBeans component means that a `RowSet` object can notify other components when certain things happen to it. For example, if data in a `RowSet` object changes, the `RowSet` object can notify interested parties of that change. The nice thing about this notification mechanism is that, as an application programmer, all you have to do is add or remove the components that will be notified.
This section covers the following topics:

Setting Up Listeners

A *listener* for a `RowSet` object is a component that implements the following methods from the `RowSetListener` interface:

- `cursorMoved`: Defines what the listener will do, if anything, when the cursor in the `RowSet` object moves.
- `rowChanged`: Defines what the listener will do, if anything, when one or more column values in a row have changed, a row has been inserted, or a row has been deleted.
- `rowSetChanged`: Defines what the listener will do, if anything, when the `RowSet` object has been populated with new data.

An example of a component that might want to be a listener is a `BarGraph` object that graphs the data in a `RowSet` object. As the data changes, the `BarGraph` object can update itself to reflect the new data.
As an application programmer, the only thing you must do to take advantage of the notification mechanism is to add or remove listeners. The following line of code means that every time the cursor for the `crs` objects moves, values in `crs` are changed, or `crs` as a whole gets new data, the `BarGraph` object `bar` will be notified:

```
crs.addRowSetListener(bar);
```

You can also stop notifications by removing a listener, as is done in the following line of code:

```
crs.removeRowSetListener(bar);
```

Using the Coffee Break scenario, assume that headquarters checks with the database periodically to get the latest price list for the coffees it sells online. In this case, the listener is the `PriceList` object `priceList` at the Coffee Break web site, which must implement the `RowSetListener` methods `cursorMoved`, `rowChanged`, and `rowSetChanged`. The implementation of the `cursorMoved` method could be to do nothing because the position of the cursor does not affect the `priceList` object. The implementations for the `rowChanged` and `rowSetChanged` methods, on the other hand, must ascertain what changes have been made and update `priceList` accordingly.

How Notification Works

In the reference implementation, methods that cause any of the `RowSet` events automatically notify all registered listeners. For example, any method that moves the cursor also calls the method `cursorMoved` on each of the listeners. Similarly, the method `execute` calls the method `rowSetChanged` on all listeners, and `acceptChanges` calls `rowChanged` on all listeners.

Sending Large Amounts of Data

The sample code `CachedRowSetSample.testCachedRowSet` demonstrates how data can be sent in smaller pieces.

Using JoinRowSet Objects

A `JoinRowSet` implementation lets you create a SQL `JOIN` between `RowSet` objects when they are not connected to a data source. This is important because it saves the overhead of having to create one or more connections.
The following topics are covered:

The `JoinRowSet` interface is a subinterface of the `CachedRowSet` interface and thereby inherits the capabilities of a `CachedRowSet` object. This means that a `JoinRowSet` object is a disconnected `RowSet` object and can operate without always being connected to a data source.

Creating JoinRowSet Objects

A `JoinRowSet` object serves as the holder of a SQL `JOIN`. The following line of code shows to create a `JoinRowSet` object:

```
JoinRowSet jrs = new JoinRowSetImpl();
```

The variable `jrs` holds nothing until `RowSet` objects are added to it.

Note: Alternatively, you can use the constructor from the `JoinRowSet` implementation of your JDBC driver. However, implementations of the `RowSet` interface will differ from the reference implementation. These implementations will have different names and constructors. For example, the Oracle JDBC driver's implementation of the `JoinRowSet` interface is named `oracle.jdbc.rowset.OracleJoinRowSet`.

Adding RowSet Objects

Any `RowSet` object can be added to a `JoinRowSet` object as long as it can be part of a SQL `JOIN`. A `JdbcRowSet` object, which is always connected to its data source, can be added, but typically it forms part of a `JOIN` by operating with the data source directly instead of becoming part of a `JOIN` by being added to a `JoinRowSet` object. The point of providing a `JoinRowSet` implementation is to make it possible for disconnected `RowSet` objects to become part of a `JOIN` relationship.

The owner of The Coffee Break chain of coffee houses wants to get a list of the coffees he buys from Acme, Inc. In order to do this, the owner will have to get information from two tables, `COFFEES` and `SUPPLIERS`. In the database world before `RowSet` technology, programmers would send the following query to the database:

```
String query =
    "SELECT COFFEES.COF_NAME " +
    "FROM COFFEES, SUPPLIERS " +
    "WHERE SUPPLIERS.SUP_NAME = Acme.Inc. " +
    "and " +
    "SUPPLIERS.SUP_ID = COFFEES.SUP_ID";
```

In the world of `RowSet` technology, you can accomplish the same result without having to send a query to the data source. You can add `RowSet` objects containing the data in the two tables to a `JoinRowSet` object. Then, because all the pertinent data is in the `JoinRowSet` object, you can perform a query on it to get the desired data.

The following code fragment from `JoinSample.testJoinRowSet` creates two `CachedRowSet` objects, `coffees` populated with the data from the table `COFFEES`, and `suppliers` populated with the data from the table `SUPPLIERS`. The `coffees` and `suppliers` objects have to make a connection to the database to execute their commands and get populated with data, but after that is done, they do not have to reconnect again in order to form a `JOIN`.

```
coffees = new CachedRowSetImpl();
coffees.setCommand("SELECT * FROM COFFEES");
coffees.setUsername(settings.userName);
coffees.setPassword(settings.password);
coffees.setUrl(settings.urlString);
coffees.execute();

suppliers = new CachedRowSetImpl();
suppliers.setCommand("SELECT * FROM SUPPLIERS");
suppliers.setUsername(settings.userName);
suppliers.setPassword(settings.password);
suppliers.setUrl(settings.urlString);
suppliers.execute();
```

Managing Match Columns

Looking at the SUPPLIERS table, you can see that Acme, Inc. has an identification number of 101. The coffees in the table COFFEES with the supplier identification number of 101 are Colombian and Colombian_Decaf. The joining of information from both tables is possible because the two tables have the column SUP_ID in common. In JDBC RowSet technology, SUP_ID, the column on which the JOIN is based, is called the *match column*.

Each RowSet object added to a JoinRowSet object must have a match column, the column on which the JOIN is based. There are two ways to set the match column for a RowSet object. The first way is to pass the match column to the JoinRowSet method addRowSet, as shown in the following line of code:

```
jrs.addRowSet(coffees, 2);
```

This line of code adds the coffees CachedRowSet to the jrs object and sets the second column of coffees (SUP_ID) as the match column. The line of code could also have used the column name rather that the column number.

```
jrs.addRowSet(coffees, "SUP_ID");
```

At this point, jrs has only coffees in it. The next RowSet object added to jrs will have to be able to form a JOIN with coffees, which is true of suppliers because both tables have the column SUP_ID. The following line of code adds suppliers to jrs and sets the column SUP_ID as the match column.

```
jrs.addRowSet(suppliers, 1);
```

Now jrs contains a JOIN between coffees and suppliers from which the owner can get the names of the coffees supplied by Acme, Inc. Because the code did not set the type of JOIN, jrs holds an inner JOIN, which is the default. In other words, a row in jrs combines a row in coffees and a row in suppliers. It holds the columns in coffees plus the columns in suppliers for rows in which the value in the COFFEES.SUP_ID column matches the value in SUPPLIERS.SUP_ID. The following code prints out the names of coffees supplied by Acme, Inc., where the String supplierName is equal to Acme, Inc. Note that this is possible because the column SUP_NAME, which is from suppliers, and COF_NAME, which is from coffees, are now both included in the JoinRowSet object jrs.

```
System.out.println("Coffees bought from " + supplierName + ": ");

while (jrs.next()) {
    if (jrs.getString("SUP_NAME").equals(supplierName)) {
        String coffeeName = jrs.getString(1);
        System.out.println("     " + coffeeName);
    }
}
```

This will produce output similar to the following:

```
Coffees bought from Acme, Inc.:
     Colombian
     Colombian_Decaf
```

The JoinRowSet interface provides constants for setting the type of JOIN that will be formed, but currently the only type that is implemented is JoinRowSet.INNER_JOIN.

Using FilteredRowSet Objects

A `FilteredRowSet` object lets you cut down the number of rows that are visible in a `RowSet` object so that you can work with only the data that is relevant to what you are doing. You decide what limits you want to set on your data (how you want to "filter" the data) and apply that filter to a `FilteredRowSet` object. In other words, the `FilteredRowSet` object makes visible only the rows of data that fit within the limits you set. A `JdbcRowSet` object, which always has a connection to its data source, can do this filtering with a query to the data source that selects only the columns and rows you want to see. The query's `WHERE` clause defines the filtering criteria. A `FilteredRowSet` object provides a way for a disconnected `RowSet` object to do this filtering without having to execute a query on the data source, thus avoiding having to get a connection to the data source and sending queries to it.
For example, assume that the Coffee Break chain of coffee houses has grown to hundreds of stores throughout the United States of America, and all of them are listed in a table called `COFFEE_HOUSES`. The owner wants to measure the success of only the stores in California with a coffee house comparison application that does not require a persistent connection to the database system. This comparison will look at the profitability of selling merchandise versus selling coffee drinks plus various other measures of success, and it will rank California stores by coffee drink sales, merchandise sales, and total sales. Because the table `COFFEE_HOUSES` has hundreds of rows, these comparisons will be faster and easier if the amount of data being searched is cut down to only those rows where the value in the column `STORE_ID` indicates California.
This is exactly the kind of problem that a `FilteredRowSet` object addresses by providing the following capabilities:

- Ability to limit the rows that are visible according to set criteria
- Ability to select which data is visible without being connected to a data source

The following topics are covered:

Defining Filtering Criteria in Predicate Objects

To set the criteria for which rows in a `FilteredRowSet` object will be visible, you define a class that implements the `Predicate` interface. An object created with this class is initialized with the following:

- The high end of the range within which values must fall
- The low end of the range within which values must fall
- The column name or column number of the column with the value that must fall within the range of values set by the high and low boundaries

Note that the range of values is inclusive, meaning that a value at the boundary is included in the range. For example, if the range has a high of 100 and a low of 50, a value of 50 is considered to be within the range. A value of 49 is not. Likewise, 100 is within the range, but 101 is not.
In line with the scenario where the owner wants to compare California stores, an implementation of the `Predicate` interface that filters for Coffee Break coffee houses located in California must be written. There is no one right way to do this, which means there is a lot of latitude in how the implementation is written. For example, you could name the class and its members whatever you want and implement a constructor and the three evaluate methods in any way that accomplishes the desired results.
The table listing all of the coffee houses, named `COFFEE_HOUSES`, has hundreds of rows. To make things more manageable, this example uses a table with far fewer rows, which is enough to demonstrate how filtering is done.
A value in the column `STORE_ID` is an `int` value that indicates, among other things, the state in which the coffee house is located. A value beginning with 10, for example, means that the state is California. `STORE_ID` values beginning with 32 indicate Oregon, and those beginning with 33 indicate the state of Washington.
The following class `StateFilter` implements the `Predicate` interface:

```
public class StateFilter implements Predicate {

    private int lo;
    private int hi;
    private String colName = null;
    private int colNumber = -1;

    public StateFilter(int lo, int hi, int colNumber) {
        this.lo = lo;
        this.hi = hi;
        this.colNumber = colNumber;
    }

    public StateFilter(int lo, int hi, String colName) {
        this.lo = lo;
        this.hi = hi;
        this.colName = colName;
    }

    public boolean evaluate(Object value, String columnName) {
        boolean evaluation = true;
        if (columnName.equalsIgnoreCase(this.colName)) {
            int columnValue = ((Integer)value).intValue();
            if ((columnValue >= this.lo)
                &&
                (columnValue <= this.hi)) {
                evaluation = true;
            } else {
                evaluation = false;
            }
        }
        return evaluation;
    }

    public boolean evaluate(Object value, int columnNumber) {

        boolean evaluation = true;

        if (this.colNumber == columnNumber) {
            int columnValue = ((Integer)value).intValue();
            if ((columnValue >= this.lo)
                &&
                (columnValue <= this.hi)) {
                evaluation = true;
            } else {
                evaluation = false;
            }
        }
```

```
        return evaluation;
    }

    public boolean evaluate(RowSet rs) {

        CachedRowSet frs = (CachedRowSet)rs;
        boolean evaluation = false;

        try {
            int columnValue = -1;

            if (this.colNumber > 0) {
                columnValue = frs.getInt(this.colNumber);
            } else if (this.colName != null) {
                columnValue = frs.getInt(this.colName);
            } else {
                return false;
            }

            if ((columnValue >= this.lo)
                &&
                (columnValue <= this.hi)) {
                evaluation = true;
            }
        } catch (SQLException e) {
            JDBCTutorialUtilities.printSQLException(e);
            return false;
        } catch (NullPointerException npe) {
            System.err.println("NullPointerException caught");
            return false;
        }
        return evaluation;
    }
}
```

This is a very simple implementation that checks the value in the column specified by either `colName` or `colNumber` to see if it is in the range of `lo` to `hi`, inclusive. The following line of code, from `FilteredRowSetSample`, creates a filter that allows only the rows where the `STORE_ID` column value indicates a value between 10000 and 10999, which indicates a California location:

```
StateFilter myStateFilter = new StateFilter(10000, 10999, 1);
```

Note that the `StateFilter` class just defined applies to one column. It is possible to have it apply to two or more columns by making each of the parameters arrays instead of single values. For example, the constructor for a `Filter` object could look like the following:

```
public Filter2(Object [] lo, Object [] hi, Object [] colNumber) {
    this.lo = lo;
    this.hi = hi;
    this.colNumber = colNumber;
}
```

The first element in the `colNumber` object gives the first column in which the value will be checked against the first element in `lo` and the first element in `hi`. The value in the second column indicated by `colNumber` will be checked against the second elements in `lo` and `hi`, and so on. Therefore, the number of elements in the three arrays should be the same. The following code is what an implementation of the method `evaluate(RowSet rs)` might look like for a `Filter2` object, in which the parameters are arrays:

```
public boolean evaluate(RowSet rs) {
    CachedRowSet crs = (CachedRowSet)rs;
    boolean bool1;
    boolean bool2;
    for (int i = 0; i < colNumber.length; i++) {

        if ((rs.getObject(colNumber[i] >= lo [i]) &&
            (rs.getObject(colNumber[i] <= hi[i]) {
            bool1 = true;
        } else {
            bool2 = true;
        }

        if (bool2) {
            return false;
        } else {
            return true;
        }
    }
}
```

The advantage of using a `Filter2` implementation is that you can use parameters of any `Object` type and can check one column or multiple columns without having to write another implementation. However, you must pass an `Object` type, which means that you must convert a primitive type to its `Object` type. For example, if you use an `int` value for `lo` and `hi`, you must convert the `int` value to an `Integer` object before passing it to the constructor. `String` objects are already `Object` types, so you do not have to convert them.

Creating FilteredRowSet Objects

The reference implementation for the `FilteredRowSet` interface, `FilteredRowSetImpl`, includes a default constructor, which is used in the following line of code to create the empty `FilteredRowSet` object `frs:.`

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

The implementation extends the `BaseRowSet` abstract class, so the `frs` object has the default properties defined in `BaseRowSet`. This means that `frs` is scrollable, updatable, does not show deleted rows, has escape processing turned on, and so on. Also, because the `FilteredRowSet` interface is a subinterface of `CachedRowSet`, `Joinable`, and `WebRowSet`, the `frs` object has the capabilities of each. It can operate as a disconnected `RowSet` object, can be part of a `JoinRowSet` object, and can read and write itself in XML format.

Note: Alternatively, you can use the constructor from the `WebRowSet` implementation of your JDBC driver. However, implementations of the `RowSet` interface will differ from the reference implementation. These implementations will have different names and constructors. For example, the Oracle JDBC driver's implementation of the `WebRowSet` interface is named `oracle.jdbc.rowset.OracleWebRowSet`.

You can use an instance of `RowSetFactory`, which is created from the class `RowSetProvider`, to create a `FilteredRowSet` object. See Using the RowSetFactory Interface in Using JdbcRowSet Objects for more information.

Like other disconnected `RowSet` objects, the `frs` object must populate itself with data from a tabular data source, which is a relational database in the reference implementation. The following code fragment from `FilteredRowSetSample` sets the properties necessary to connect to a database to execute its command. Note that this code uses the `DriverManager` class to make a connection, which is done for convenience. Usually, it is better to use a `DataSource` object that has been registered with a naming service that implements the Java Naming and Directory Interface (JNDI):

```
frs.setCommand("SELECT * FROM COFFEE_HOUSES");
frs.setUsername(settings.userName);
frs.setPassword(settings.password);
frs.setUrl(settings.urlString);
```

The following line of code populates the `frs` objectwith the data stored in the `COFFEE_HOUSE` table:

```
frs.execute();
```

The method `execute` does all kinds of things in the background by calling on the `RowSetReader` object for `frs`, which creates a connection, executes the command for `frs`, populates `frs` with the data from the `ResultSet` object that is produced, and closes the connection. Note that if the table `COFFEE_HOUSES` had more rows than the `frs` object could hold in memory at one time, the `CachedRowSet` paging methods would have been used.

In the scenario, the Coffee Break owner would have done the preceding tasks in the office and then imported or downloaded the information stored in the `frs` object to the coffee house comparison application. From now on, the `frs` object will operate independently without the benefit of a connection to the data source.

Creating and Setting Predicate Objects

Now that the `FilteredRowSet` object `frs` contains the list of Coffee Break establishments, you can set selection criteria for narrowing down the number of rows in the `frs` object that are visible.
The following line of code uses the `StateFilter` class defined previously to create the object `myStateFilter`, which checks the column `STORE_ID` to determine which stores are in California (a store is in California if its ID number is between 10000 and 10999, inclusive):

```
StateFilter myStateFilter = new StateFilter(10000, 10999, 1);
```

The following line sets `myStateFilter` as the filter for `frs`.

```
frs.setFilter(myStateFilter);
```

To do the actual filtering, you call the method `next`, which in the reference implementation calls the appropriate version of the `Predicate.evaluate` method that you have implemented previously.
If the return value is `true`, the row will be visible; if the return value is `false`, the row will not be visible.

Setting FilteredRowSet Objects with New Predicate Objects to Filter Data Further

You set multiple filters serially. The first time you call the method `setFilter` and pass it a `Predicate` object, you have applied the filtering criteria in that filter. After calling the method `next` on each row, which makes visible only those rows that satisfy the filter, you can call `setFilter` again, passing it a different `Predicate` object. Even though only one filter is set at a time, the effect is that both filters apply cumulatively.

For example, the owner has retrieved a list of the Coffee Break stores in California by setting `stateFilter` as the `Predicate` object for `frs`. Now the owner wants to compare the stores in two California cities, San Francisco (SF in the table `COFFEE_HOUSES`) and Los Angeles (LA in the table). The first thing to do is to write a `Predicate` implementation that filters for stores in either SF or LA:

```
public class CityFilter implements Predicate {

    private String[] cities;
    private String colName = null;
    private int colNumber = -1;

    public CityFilter(String[] citiesArg, String colNameArg) {
        this.cities = citiesArg;
        this.colNumber = -1;
        this.colName = colNameArg;
    }

    public CityFilter(String[] citiesArg, int colNumberArg) {
        this.cities = citiesArg;
        this.colNumber = colNumberArg;
        this.colName = null;
    }

    public boolean evaluate Object valueArg, String colNameArg) {

        if (colNameArg.equalsIgnoreCase(this.colName)) {
            for (int i = 0; i < this.cities.length; i++) {
                if (this.cities[i].equalsIgnoreCase((String)valueArg)) {
                    return true;
                }
            }
        }
        return false;
    }

    public boolean evaluate(Object valueArg, int colNumberArg) {

        if (colNumberArg == this.colNumber) {
            for (int i = 0; i < this.cities.length; i++) {
                if (this.cities[i].equalsIgnoreCase((String)valueArg)) {
                    return true;
                }
            }
        }
        return false;
    }


    public boolean evaluate(RowSet rs) {

        if (rs == null) return false;

        try {
            for (int i = 0; i < this.cities.length; i++) {

                String cityName = null;

                if (this.colNumber > 0) {
                    cityName = (String)rs.getObject(this.colNumber);
                } else if (this.colName != null) {
                    cityName = (String)rs.getObject(this.colName);
                } else {
                    return false;
                }

                if (cityName.equalsIgnoreCase(cities[i])) {
                    return true;
                }
            }
        } catch (SQLException e) {
            return false;
        }
        return false;
    }
}
```

The following code fragment from `FilteredRowSetSample` sets the new filter and iterates through the rows in `frs`, printing out the rows where the `CITY` column contains either SF or LA. Note that `frs` currently contains only rows where the store is in California, so the criteria of the `Predicate` object `state` are still in effect when the filter is changed to another `Predicate` object. The code that follows sets the filter to the `CityFilter` object `city`. The `CityFilter` implementation uses arrays as parameters to the constructors to illustrate how that can be done:

```
public void testFilteredRowSet() {
```

```
    FilteredRowSet frs = null;
    StateFilter myStateFilter = new StateFilter(10000, 10999, 1);
    String[] cityArray = { "SF", "LA" };

    CityFilter myCityFilter = new CityFilter(cityArray, 2);

    try {
        frs = new FilteredRowSetImpl();

        frs.setCommand("SELECT * FROM COFFEE_HOUSES");
        frs.setUsername(settings.userName);
        frs.setPassword(settings.password);
        frs.setUrl(settings.urlString);
        frs.execute();

        System.out.println("\nBefore filter:");
        FilteredRowSetSample.viewTable(this.con);

        System.out.println("\nSetting state filter:");
        frs.beforeFirst();
        frs.setFilter(myStateFilter);
        this.viewFilteredRowSet(frs);

        System.out.println("\nSetting city filter:");
        frs.beforeFirst();
        frs.setFilter(myCityFilter);
        this.viewFilteredRowSet(frs);
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    }
}
```

The output should contain a row for each store that is in San Francisco, California or Los Angeles, California. If there were a row in which the `CITY` column contained LA and the `STORE_ID` column contained 40003, it would not be included in the list because it had already been filtered out when the filter was set to `state`. (40003 is not in the range of 10000 to 10999.)

Updating FilteredRowSet Objects

You can make a change to a `FilteredRowSet` object but only if that change does not violate any of the filtering criteria currently in effect. For example, you can insert a new row or change one or more values in an existing row if the new value or values are within the filtering criteria.

Inserting or Updating Rows

Assume that two new Coffee Break coffee houses have just opened and the owner wants to add them to the list of all coffee houses. If a row to be inserted does not meet the cumulative filtering criteria in effect, it will be blocked from being added.

The current state of the `frs` object is that the `StateFilter` object was set and then the `CityFilter` object was set. As a result, `frs` currently makes visible only those rows that satisfy the criteria for both filters. And, equally important, you cannot add a row to the `frs` object unless it satisfies the criteria for both filters. The following code fragment attempts to insert two new rows into the `frs` object, one row in which the values in the `STORE_ID` and `CITY` columns both meet the criteria, and one row in which the value in `STORE_ID` does not pass the filter but the value in the `CITY` column does:

```
frs.moveToInsertRow();
frs.updateInt("STORE_ID", 10101);
frs.updateString("CITY", "SF");
frs.updateLong("COF_SALES", 0);
frs.updateLong("MERCH_SALES", 0);
frs.updateLong("TOTAL_SALES", 0);
frs.insertRow();

frs.updateInt("STORE_ID", 33101);
frs.updateString("CITY", "SF");
frs.updateLong("COF_SALES", 0);
frs.updateLong("MERCH_SALES", 0);
frs.updateLong("TOTAL_SALES", 0);
frs.insertRow();
frs.moveToCurrentRow();
```

If you were to iterate through the `frs` object using the method `next`, you would find a row for the new coffee house in San Francisco, California, but not for the store in San Francisco, Washington.

Removing All Filters so All Rows Are Visible

The owner can add the store in Washington by nullifying the filter. With no filter set, all rows in the `frs` object are once more visible, and a store in any location can be added to the list of stores. The following line of code unsets the current filter, effectively nullifying both of the `Predicate` implementations previously set on the `frs` object.

```
frs.setFilter(null);
```

Deleting Rows

If the owner decides to close down or sell one of the Coffee Break coffee houses, the owner will want to delete it from the `COFFEE_HOUSES` table. The owner can delete the row for the underperforming coffee house as long as the row is visible.

For example, given that the method `setFilter` has just been called with the argument null, there is no filter set on the `frs` object. This means that all rows are visible and can therefore be deleted. However, after the `StateFilter` object `myStateFilter` was set, which filtered out any state other than California, only stores located in California could be deleted. When the `CityFilter` object `myCityFilter` was set for the `frs` object, only coffee houses in San Francisco, California or Los Angeles, California could be deleted because they were in the only rows visible.

Using WebRowSet Objects

A `WebRowSet` object is very special because in addition to offering all of the capabilities of a `CachedRowSet` object, it can write itself as an XML document and can also read that XML document to convert itself back to a `WebRowSet` object. Because XML is the language through which disparate enterprises can communicate with each other, it has become the standard for Web Services communication. As a consequence, a `WebRowSet` object fills a real need by enabling Web Services to send and receive data from a database in the form of an XML document.
The following topics are covered:

The Coffee Break company has expanded to selling coffee online. Users order coffee by the pound from the Coffee Break Web site. The price list is regularly updated by getting the latest information from the company's database. This section demonstrates how to send the price data as an XML document with a `WebRowSet` object and a single method call.

Creating and Populating WebRowSet Objects

You create a new `WebRowSet` object with the default constructor defined in the reference implementation, `WebRowSetImpl`, as shown in the following line of code:

```
WebRowSet priceList = new WebRowSetImpl();
```

Although the `priceList` object has no data yet, it has the default properties of a `BaseRowSet` object. Its `SyncProvider` object is at first set to the `RIOptimisticProvider` implementation, which is the default for all disconnected `RowSet` objects. However, the `WebRowSet` implementation resets the `SyncProvider` object to be the `RIXMLProvider` implementation.

You can use an instance of `RowSetFactory`, which is created from the `RowSetProvider` class, to create a `WebRowSet` object. See Using the RowSetFactory Interface in Using JdbcRowSet Objects for more information.

The Coffee Break headquarters regularly sends price list updates to its web site. This information on `WebRowSet` objects will show one way you can send the latest price list in an XML document.

The price list consists of the data in the columns `COF_NAME` and `PRICE` from the table `COFFEES`. The following code fragment sets the properties needed and populates the `priceList` object with the price list data:

```
public void getPriceList(String username, String password) {
    priceList.setCommand("SELECT COF_NAME, PRICE FROM COFFEES");
    priceList.setURL("jdbc:mySubprotocol:myDatabase");
    priceList.setUsername(username);
    priceList.setPassword(password);
    priceList.execute();
    // ...
}
```

At this point, in addition to the default properties, the `priceList` object contains the data in the `COF_NAME` and `PRICE` columns from the `COFFEES` table and also the metadata about these two columns.

Writing and Reading WebRowSet Object to XML

To write a `WebRowSet` object as an XML document, call the method `writeXml`. To read that XML document's contents into a `WebRowSet` object, call the method `readXml`. Both of these methods do their work in the background, meaning that everything, except the results, is invisible to you.

Using the writeXml Method

The method `writeXml` writes the `WebRowSet` object that invoked it as an XML document that represents its current state. It writes this XML document to the stream that you pass to it. The stream can be an `OutputStream` object, such as a `FileOutputStream` object, or a `Writer` object, such as a `FileWriter` object. If you pass the method `writeXml` an `OutputStream` object, you will write in bytes, which can handle all types of data; if you pass it a `Writer` object, you will write in characters. The following code demonstrates writing the `WebRowSet` object `priceList` as an XML document to the `FileOutputStream` object `oStream`:

```
java.io.FileOutputStream oStream =
    new java.io.FileOutputStream("priceList.xml");
priceList.writeXml(oStream);
```

The following code writes the XML document representing `priceList` to the `FileWriter` object `writer` instead of to an `OutputStream` object. The `FileWriter` class is a convenience class for writing characters to a file.

```
java.io.FileWriter writer =
    new java.io.FileWriter("priceList.xml");
priceList.writeXml(writer);
```

The other two versions of the method `writeXml` let you populate a `WebRowSet` object with the contents of a `ResultSet` object before writing it to a stream. In the following line of code, the method `writeXml` reads the contents of the `ResultSet` object `rs` into the `priceList` object and then writes `priceList` to the `FileOutputStream` object `oStream` as an XML document.

```
priceList.writeXml(rs, oStream);
```

In the next line of code, the `writeXml` methodpopulates `priceList` with the contents of `rs`, but it writes the XML document to a `FileWriter` object instead of to an `OutputStream` object:

```
priceList.writeXml(rs, writer);
```

Using the readXml Method

The method `readXml` parses an XML document in order to construct the `WebRowSet` object the XML document describes. Similar to the method `writeXml`, you can pass `readXml` an `InputStream` object or a `Reader` object from which to read the XML document.

```
java.io.FileInputStream iStream =
    new java.io.FileInputStream("priceList.xml");
priceList.readXml(iStream);

java.io.FileReader reader = new
    java.io.FileReader("priceList.xml");
priceList.readXml(reader);
```

Note that you can read the XML description into a new `WebRowSet` object or into the same `WebRowSet` object that called the `writeXml` method. In the scenario, where the price list information is being sent from headquarters to the Web site, you would use a new `WebRowSet` object, as shown in the following lines of code:

```
WebRowSet recipient = new WebRowSetImpl();
java.io.FileReader reader =
    new java.io.FileReader("priceList.xml");
recipient.readXml(reader);
```

What Is in XML Documents

`RowSet` objects are more than just the data they contain. They have properties and metadata about their columns as well. Therefore, an XML document representing a `WebRowSet` object includes this other information in addition to its data. Further, the data in an XML document includes both current values and original values. (Recall that original values are the values that existed immediately before the most recent changes to data were made. These values are necessary for checking if the corresponding value in the database has been changed, thus creating a conflict over which value should be persistent: the new value you put in the `RowSet` object or the new value someone else put in the database.)
The WebRowSet XML Schema, itself an XML document, defines what an XML document representing a `WebRowSet` object will contain and also the format in which it must be presented. Both the sender and the recipient use this schema because it tells the sender how to write the XML document (which represents the `WebRowSet` object) and the recipient how to parse the XML document. Because the actual writing and reading is done internally by the implementations of the methods `writeXml` and `readXml`, you, as a user, do not need to understand what is in the WebRowSet XML Schema document.
XML documents contain elements and subelements in a hierarchical structure. The following are the three main elements in an XML document describing a `WebRowSet` object:

- Properties
- Metadata
- Data

Element tags signal the beginning and end of an element. For example, the `<properties>` tag signals the beginning of the properties element, and the `</properties>` tag signals its end. The `<map/>` tag is a shorthand way of saying that the map subelement (one of the subelements in the properties element) has not been assigned a value. The following sample XML documents uses spacing and indentation to make it easier to read, but those are not used in an actual XML document, where spacing does not mean anything.
The next three sections show you what the three main elements contain for the `WebRowSet priceList` object, created in the sample `WebRowSetSample.java`.

Properties

Calling the method `writeXml` on the `priceList` object would produce an XML document describing `priceList`. The properties section of this XML document would look like the following:

```
<properties>
  <command>
    select COF_NAME, PRICE from COFFEES
  </command>
  <concurrency>1008</concurrency>
  <datasource><null/></datasource>
  <escape-processing>true</escape-processing>
  <fetch-direction>1000</fetch-direction>
  <fetch-size>0</fetch-size>
  <isolation-level>2</isolation-level>
  <key-columns>
    <column>1</column>
  </key-columns>
  <map>
  </map>
  <max-field-size>0</max-field-size>
  <max-rows>0</max-rows>
  <query-timeout>0</query-timeout>
  <read-only>true</read-only>
  <rowset-type>
    ResultSet.TYPE_SCROLL_INSENSITIVE
  </rowset-type>
  <show-deleted>false</show-deleted>
  <table-name>COFFEES</table-name>
  <url>jdbc:mysql://localhost:3306/testdb</url>
  <sync-provider>
    <sync-provider-name>
      com.sun.rowset.providers.RIOptimisticProvider
```

```
    </sync-provider-name>
    <sync-provider-vendor>
      Sun Microsystems Inc.
    </sync-provider-vendor>
    <sync-provider-version>
      1.0
    </sync-provider-version>
    <sync-provider-grade>
      2
    </sync-provider-grade>
    <data-source-lock>1</data-source-lock>
  </sync-provider>
</properties>
```

Notice that some properties have no value. For example, the `datasource` property is indicated with the `<datasource/>` tag, which is a shorthand way of saying `<datasource></datasource>`. No value is given because the `url` property is set. Any connections that are established will be done using this JDBC URL, so no `DataSource` object needs to be set. Also, the `username` and `password` properties are not listed because they must remain secret.

Metadata

The metadata section of the XML document describing a `WebRowSet` object contains information about the columns in that `WebRowSet` object. The following shows what this section looks like for the `WebRowSet` object `priceList`. Because the `priceList` object has two columns, the XML document describing it has two `<column-definition>` elements. Each `<column-definition>` element has subelements giving information about the column being described.

```
<metadata>
  <column-count>2</column-count>
  <column-definition>
    <column-index>1</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>false</case-sensitive>
    <currency>false</currency>
    <nullable>0</nullable>
    <signed>false</signed>
    <searchable>true</searchable>
    <column-display-size>
      32
    </column-display-size>
    <column-label>COF_NAME</column-label>
    <column-name>COF_NAME</column-name>
    <schema-name></schema-name>
    <column-precision>32</column-precision>
    <column-scale>0</column-scale>
    <table-name>coffees</table-name>
    <catalog-name>testdb</catalog-name>
    <column-type>12</column-type>
    <column-type-name>
      VARCHAR
    </column-type-name>
  </column-definition>
  <column-definition>
    <column-index>2</column-index>
    <auto-increment>false</auto-increment>
    <case-sensitive>true</case-sensitive>
    <currency>false</currency>
    <nullable>0</nullable>
    <signed>true</signed>
    <searchable>true</searchable>
    <column-display-size>
      12
    </column-display-size>
    <column-label>PRICE</column-label>
    <column-name>PRICE</column-name>
    <schema-name></schema-name>
    <column-precision>10</column-precision>
    <column-scale>2</column-scale>
    <table-name>coffees</table-name>
    <catalog-name>testdb</catalog-name>
    <column-type>3</column-type>
    <column-type-name>
      DECIMAL
    </column-type-name>
  </column-definition>
</metadata>
```

From this metadata section, you can see that there are two columns in each row. The first column is `COF_NAME`, which holds values of type `VARCHAR`. The second column is `PRICE`, which holds values of type `REAL`, and so on. Note that the column types are the data types used in the data source, not types in the Java programming language. To get or update values in the `COF_NAME` column, you use the methods `getString` or `updateString`, and the driver makes the conversion to the `VARCHAR` type, as it usually does.

Data

The data section gives the values for each column in each row of a `WebRowSet` object. If you have

populated the `priceList` object and not made any changes to it, the data element of the XML document will look like the following. In the next section you will see how the XML document changes when you modify the data in the `priceList` object.

For each row there is a `<currentRow>` element, and because `priceList` has two columns, each `<currentRow>` element contains two `<columnValue>` elements.

```
<data>
  <currentRow>
    <columnValue>Colombian</columnValue>
    <columnValue>7.99</columnValue>
  </currentRow>
  <currentRow>
    <columnValue>
      Colombian_Decaf
    </columnValue>
    <columnValue>8.99</columnValue>
  </currentRow>
  <currentRow>
    <columnValue>Espresso</columnValue>
    <columnValue>9.99</columnValue>
  </currentRow>
  <currentRow>
    <columnValue>French_Roast</columnValue>
    <columnValue>8.99</columnValue>
  </currentRow>
  <currentRow>
    <columnValue>French_Roast_Decaf</columnValue>
    <columnValue>9.99</columnValue>
  </currentRow>
</data>
```

Making Changes to WebRowSet Objects

You make changes to a `WebRowSet` object the same way you do to a `CachedRowSet` object. Unlike a `CachedRowSet` object, however, a `WebRowSet` object keeps track of updates, insertions, and deletions so that the `writeXml` method can write both the current values and the original values. The three sections that follow demonstrate making changes to the data and show what the XML document describing the `WebRowSet` object looks like after each change. You do not have to do anything at all regarding the XML document; any change to it is made automatically, just as with writing and reading the XML document.

Inserting Rows

If the owner of the Coffee Break chain wants to add a new coffee to the price list, the code might look like this:

```
priceList.absolute(3);
priceList.moveToInsertRow();
priceList.updateString(COF_NAME, "Kona");
priceList.updateFloat(PRICE, 8.99f);
priceList.insertRow();
priceList.moveToCurrentRow();
```

In the reference implementation, an insertion is made immediately following the current row. In the preceding code fragment, the current row is the third row, so the new row would be added after the third row and become the new fourth row. To reflect this insertion, the XML document would have the following `<insertRow>` element added to it after the third `<currentRow>` element in the `<data>` element.

The `<insertRow>` element will look similar to the following.

```
<insertRow>
  <columnValue>Kona</columnValue>
  <columnValue>8.99</columnValue>
</insertRow>
```

Deleting Rows

The owner decides that Espresso is not selling enough and should be removed from the coffees sold at The Coffee Break shops. The owner therefore wants to delete Espresso from the price list. Espresso is in the third row of the `priceList` object, so the following lines of code delete it:

```
priceList.absolute(3); priceList.deleteRow();
```

The following `<deleteRow>` element will appear after the second row in the data section of the XML document, indicating that the third row has been deleted.

```
<deleteRow>
  <columnValue>Espresso</columnValue>
  <columnValue>9.99</columnValue>
</deleteRow>
```

Modifying Rows

The owner further decides that the price of Colombian coffee is too expensive and wants to lower it to .99 a pound. The following code sets the new price for Colombian coffee, which is in the first row, to .99 a pound:

```
priceList.first();
priceList.updateFloat(PRICE, 6.99);
```

The XML document will reflect this change in an `<updateRow>` element that gives the new value. The value for the first column did not change, so there is an `<updateValue>` element only for the second column:

```
<currentRow>
  <columnValue>Colombian</columnValue>
  <columnValue>7.99</columnValue>
  <updateRow>6.99</updateRow>
</currentRow>
```

At this point, with the insertion of a row, the deletion of a row, and the modification of a row, the XML document for the `priceList` object would look like the following:

```
<data>
  <insertRow>
    <columnValue>Kona</columnValue>
    <columnValue>8.99</columnValue>
  </insertRow>
  <currentRow>
    <columnValue>Colombian</columnValue>
    <columnValue>7.99</columnValue>
    <updateRow>6.99</updateRow>
  </currentRow>
  <currentRow>
    <columnValue>
      Colombian_Decaf
    </columnValue>
    <columnValue>8.99</columnValue>
  </currentRow>
  <deleteRow>
    <columnValue>Espresso</columnValue>
    <columnValue>9.99</columnValue>
  </deleteRow>
  <currentRow>
    <columnValue>French_Roast</columnValue>
    <columnValue>8.99</columnValue>
  </currentRow>
  <currentRow>
    <columnValue>
      French_Roast_Decaf
    </columnValue>
    <columnValue>9.99</columnValue>
  </currentRow>
</data>
```

WebRowSet Code Example

The sample `WebRowSetSample.java` demonstrates all the features described on this page.

Using Advanced Data Types

The advanced data types introduced in this section give a relational database more flexibility in what can be used as a value for a table column. For example, a column can be used to store `BLOB` (binary large object) values, which can store very large amounts of data as raw bytes. A column can also be of type `CLOB` (character large object), which is capable of storing very large amounts of data in character format.
The latest version of the ANSI/ISO SQL standard is commonly referred to as SQL:2003. This standard specifies the following data types:

- SQL92 built-in types, which consist of the familiar SQL column types such as `CHAR`, `FLOAT`, and `DATE`
- SQL99 built-in types, which consist of types added by SQL99:
  - `BOOLEAN`: Boolean (true or false) value
  - `BLOB`: Binary large Bobject
  - `CLOB`: Character large object
- New built-in types added by SQL:2003:
  - `XML`: XML object
- User defined types:
  - Structured type: User-defined type; for example:

    ```
    CREATE TYPE PLANE_POINT
    AS (X FLOAT, Y FLOAT) NOT FINAL
    ```
  - `DISTINCT` type: 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`*`)`: Pointer that persistently denotes an instance of a structured type that resides in the database
  - *`base-type`* `ARRAY[`*`n`*`]`: Array of *n* base-type elements
- Locators: Entities that are logical pointers to data that resides on the database server. A *locator* exists in the client computer 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. (*Materialized views* are query results that have been stored or "materialized" in advance as schema objects.) There are operators defined at the SQL level to retrieve randomly accessed pieces of the data denoted by the locator:
  - `LOCATOR(`*`structured-type`*`)`: Locator to a structured instance in the server
  - `LOCATOR(`*`array`*`)`: Locator to an array in the server
  - `LOCATOR(`*`blob`*`)`: Locator to a binary large object in the server
  - `LOCATOR(`*`clob`*`)`: Locator to a character large object in the server
- `Datalink`: Type for managing data external to the data source. `Datalink` values are part of SQL MED (Management of External Data), a part of the SQL ANSI/ISO standard specification.

Mapping Advanced Data Types

The JDBC API provides default mappings for advanced data types specified by the SQL:2003 standard. The following list gives the data types and the interfaces or classes to which they are mapped:

- `BLOB`: `Blob` interface
- `CLOB`: `Clob` interface
- `NCLOB`: `NClob` interface
- `ARRAY`: `Array` interface
- `XML`: `SQLXML` interface
- Structured types: `Struct` interface
- `REF(structured type)`: `Ref` interface
- `ROWID`: `RowId` interface
- `DISTINCT`: Type to which the base type is mapped. For example, a `DISTINCT` value based on a SQL `NUMERIC` type maps to a `java.math.BigDecimal` type because `NUMERIC` maps to `BigDecimal` in the Java programming language.
- `DATALINK`: `java.net.URL` object

Using Advanced Data Types

You retrieve, store, and update advanced data types the same way you handle other data types. You use either `ResultSet.get`*`DataType`* or `CallableStatement.get`*`DataType`* methods to retrieve them, `PreparedStatement.set`*`DataType`* methods to store them, and `ResultSet.update`*`DataType`* methods to update them. (The variable *`DataType`* is the name of a Java interface or class mapped to an advanced data type.) Probably 90 percent of the operations performed on advanced data types involve using the `get`*`DataType`*, `set`*`DataType`*, and `update`*`DataType`* methods. The following table shows which methods to use:

Advanced Data Type	`get`*`DataType`* Method	`set`*`DataType`* method	`update`*`DataType`* Method
`BLOB`	`getBlob`	`setBlob`	`updateBlob`
`CLOB`	`getClob`	`setClob`	`updateClob`
`NCLOB`	`getNClob`	`setNClob`	`updateNClob`
`ARRAY`	`getArray`	`setArray`	`updateArray`
`XML`	`getSQLXML`	`setSQLXML`	`updateSQLXML`
`Structured type`	`getObject`	`setObject`	`updateObject`
`REF(structured type)`	`getRef`	`setRef`	`updateRef`
`ROWID`	`getRowId`	`setRowId`	`updateRowId`
`DISTINCT`	`getBigDecimal`	`setBigDecimal`	`updateBigDecimal`
`DATALINK`	`getURL`	`setURL`	`updateURL`

Note: The `DISTINCT` data type behaves differently from other advanced SQL data types. Being a user-defined type that is based on an already existing built-in types, it has no interface as its mapping in the Java programming language. Consequently, you use the method that corresponds to the Java type on which the `DISTINCT` data type is based. See Using DISTINCT Data Type for more information.

For example, the following code fragment retrieves a SQL `ARRAY` value. For this example, suppose that the column `SCORES` in the table `STUDENTS` contains values of type `ARRAY`. The variable *`stmt`* is a `Statement` object.

```
ResultSet rs = stmt.executeQuery(
    "SELECT SCORES FROM STUDENTS " +
    "WHERE ID = 002238");
rs.next();
Array scores = rs.getArray("SCORES");
```

The variable *`scores`* is a logical pointer to the SQL `ARRAY` object stored in the table `STUDENTS` in the row for student `002238`.

If you want to store a value in the database, you use the appropriate `set` method. For example, the following code fragment, in which *`rs`* is a `ResultSet` object, stores a `Clob` object:

```
Clob notes = rs.getClob("NOTES");
PreparedStatement pstmt =
    con.prepareStatement(
        "UPDATE MARKETS SET COMMENTS = ? " +
        "WHERE SALES < 1000000");
pstmt.setClob(1, notes);
pstmt.executeUpdate();
```

This code sets *`notes`* as the first parameter in the update statement being sent to the database. The `Clob` value designated by *`notes`* will be stored in the table `MARKETS` in column `COMMENTS` in every row where the value in the column `SALES` is less than one million.

Using Large Objects

An important feature of `Blob`, `Clob`, and `NClob` Java objects is that you can manipulate them without having to bring all of their data from the database server to your client computer. Some implementations represent an instance of these types with a locator (logical pointer) to the object in the database that the instance represents. Because a `BLOB`, `CLOB`, or `NCLOB` SQL object may be very large, the use of locators can make performance significantly faster. However, other implementations fully materialize large objects on the client computer.

If you want to bring the data of a `BLOB`, `CLOB`, or `NCLOB` SQL value to the client computer, use methods in the `Blob`, `Clob`, and `NClob` Java interfaces that are provided for this purpose. These large object type objects materialize the data of the objects they represent as a stream.

The following topics are covered:

Adding Large Object Type Object to Database

The following excerpt from `ClobSample.addRowToCoffeeDescriptions` adds a `CLOB` SQL value to the table `COFFEE_DESCRIPTIONS`. The `Clob` Java object `myClob` contains the contents of the file specified by `fileName`.

```
public void addRowToCoffeeDescriptions(
    String coffeeName, String fileName)
    throws SQLException {

    PreparedStatement pstmt = null;
    try {
        Clob myClob = this.con.createClob();
        Writer clobWriter = myClob.setCharacterStream(1);
        String str = this.readFile(fileName, clobWriter);
        System.out.println("Wrote the following: " +
            clobWriter.toString());

        if (this.settings.dbms.equals("mysql")) {
            System.out.println(
                "MySQL, setting String in Clob " +
                "object with setString method");
            myClob.setString(1, str);
        }
        System.out.println("Length of Clob: " + myClob.length());

        String sql = "INSERT INTO COFFEE_DESCRIPTIONS " +
                     "VALUES(?,?)";

        pstmt = this.con.prepareStatement(sql);
        pstmt.setString(1, coffeeName);
        pstmt.setClob(2, myClob);
        pstmt.executeUpdate();
    } catch (SQLException sqlex) {
        JDBCTutorialUtilities.printSQLException(sqlex);
    } catch (Exception ex) {
      System.out.println("Unexpected exception: " + ex.toString());
    } finally {
        if (pstmt != null)pstmt.close();
    }
}
```

The following line creates a `Clob` Java object:

```
Clob myClob = this.con.createClob();
```

The following line retrieves a stream (in this case a `Writer` object named `clobWriter`) that is used to write a stream of characters to the `Clob` Java object `myClob`. The method `ClobSample.readFile` writes this stream of characters; the stream is from the file specified by the `String fileName`. The method argument `1` indicates that the `Writer` object will start writing the stream of characters at the beginning of the `Clob` value:

```
Writer clobWriter = myClob.setCharacterStream(1);
```

The `ClobSample.readFile` method reads the file line-by-line specified by the file `fileName` and writes it to the `Writer` object specified by `writerArg`:

```
private String readFile(String fileName, Writer writerArg)
        throws FileNotFoundException, IOException {

    BufferedReader br = new BufferedReader(new FileReader(fileName));
    String nextLine = "";
    StringBuffer sb = new StringBuffer();
    while ((nextLine = br.readLine()) != null) {
        System.out.println("Writing: " + nextLine);
        writerArg.write(nextLine);
        sb.append(nextLine);
    }
    // Convert the content into to a string
    String clobData = sb.toString();

    // Return the data.
    return clobData;
}
```

The following excerpt creates a `PreparedStatement` object `pstmt` that inserts the `Clob` Java object `myClob` into `COFFEE_DESCRIPTIONS`:

```
PreparedStatement pstmt = null;
// ...
String sql = "INSERT INTO COFFEE_DESCRIPTIONS VALUES(?,?)";
pstmt = this.con.prepareStatement(sql);
pstmt.setString(1, coffeeName);
pstmt.setClob(2, myClob);
pstmt.executeUpdate();
```

Retrieving CLOB Values

The method `ClobSample.retrieveExcerpt` retrieves the `CLOB` SQL value stored in the `COF_DESC` column of `COFFEE_DESCRIPTIONS` from the row whose column value `COF_NAME` is equal to the `String` value specified by the `coffeeName` parameter:

```
public String retrieveExcerpt(String coffeeName, int numChar)
    throws SQLException {

    String description = null;
    Clob myClob = null;
    PreparedStatement pstmt = null;

    try {
        String sql =
            "select COF_DESC " +
            "from COFFEE_DESCRIPTIONS " +
            "where COF_NAME = ?";

        pstmt = this.con.prepareStatement(sql);
        pstmt.setString(1, coffeeName);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            myClob = rs.getClob(1);
            System.out.println("Length of retrieved Clob: " +
                myClob.length());
        }
        description = myClob.getSubString(1, numChar);
    } catch (SQLException sqlex) {
        JDBCTutorialUtilities.printSQLException(sqlex);
    } catch (Exception ex) {
        System.out.println("Unexpected exception: " + ex.toString());
    } finally {
        if (pstmt != null) pstmt.close();
    }
    return description;
}
```

The following line retrieves the `Clob` Java value from the `ResultSet` object `rs`:

```
myClob = rs.getClob(1);
```

The following line retrieves a substring from the `myClob` object. The substring begins at the first character of the value of `myClob` and has up to the number of consecutive characters specified in `numChar`, where `numChar` is an integer.

```
description = myClob.getSubString(1, numChar);
```

Adding and Retrieving BLOB Objects

Adding and retrieving `BLOB` SQL objects is similar to adding and retrieving `CLOB` SQL objects. Use the `Blob.setBinaryStream` method to retrieve an `OutputStream` object to write the `BLOB` SQL value that the `Blob` Java object (which called the method) represents.

Releasing Resources Held by Large Objects

`Blob`, `Clob`, and `NClob` Java 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 the following excerpt, the method `Clob.free` is called to release the resources held for a previously created `Clob` object:

```
Clob aClob = con.createClob();
int numWritten = aClob.setString(1, val);
aClob.free();
```

Using 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`, `setBinaryStream`, `setCharacterStream` or `setResult` method on the `SQLXML` interface.
The following topics are covered:

Creating SQLXML Objects

In the following excerpt, the method `Connection.createSQLXML` is used to create an empty `SQLXML` object. The `SQLXML.setString` method is used to write data to the `SQLXML` object that was created.

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

Retrieving SQLXML Values in 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, the following excerpt retrieves a `SQLXML` value from the first column of the `ResultSet` *rs*:

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

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

Accessing SQLXML Object Data

The `SQLXML` interface provides the `getString`, `getBinaryStream`, `getCharacterStream`, and `getSource` methods to access its internal content. The following excerpt retrieves the contents of an `SQLXML` object using the `getString` method:

```
SQLXML xmlVal= rs.getSQLXML(1);
String val = xmlVal.getString();
```

The `getBinaryStream` or `getCharacterStream` methods can be used to obtain an `InputStream` or a `Reader` object that can be passed directly to an XML parser. The following excerpt obtains an `InputStream` object from an `SQLXML` Object and then processes the stream using a DOM (Document Object Model) parser:

```
SQLXML sqlxml = rs.getSQLXML(column);
InputStream binaryStream = sqlxml.getBinaryStream();
DocumentBuilder parser =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document result = parser.parse(binaryStream);
```

The `getSource` method returns a `javax.xml.transform.Source` object. Sources are used as input to XML parsers and XSLT transformers.

The following excerpt retrieves and parses the data from a `SQLXML` object using the `SAXSource` object returned by invoking the `getSource` method:

```
SQLXML xmlVal= rs.getSQLXML(1);
SAXSource saxSource = sqlxml.getSource(SAXSource.class);
XMLReader xmlReader = saxSource.getXMLReader();
xmlReader.setContentHandler(myHandler);
xmlReader.parse(saxSource.getInputSource());
```

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 the following excerpt, `authorData` is an instance of the `java.sql.SQLXML` interface whose data was initialized previously.

```
PreparedStatement pstmt = conn.prepareStatement("INSERT INTO bio " +
                          "(xmlData, authId) VALUES (?, ?)");
pstmt.setSQLXML(1, authorData);
pstmt.setInt(2, authorId);
```

The `updateSQLXML` method can be used to update a column value in an updatable result set.
If the `java.xml.transform.Result`, `Writer`, or `OutputStream` object for the `SQLXML` object has not been closed prior to calling `setSQLXML` or `updateSQLXML`, a `SQLException` will be thrown.

Initializing SQLXML Objects

The `SQLXML` interface provides the methods `setString`, `setBinaryStream`, `setCharacterStream`, or `setResult` to initialize the content for a `SQLXML` object that has been created by calling the `Connection.createSQLXML` method.

The following excerpt uses the method `setResult` to return a `SAXResult` object to populate a newly created `SQLXML` object:

```
SQLXML sqlxml = con.createSQLXML();
SAXResult saxResult = sqlxml.setResult(SAXResult.class);
ContentHandler contentHandler = saxResult.getXMLReader().getContentHandler();
contentHandler.startDocument();

// set the XML elements and
// attributes into the result
contentHandler.endDocument();
```

The following excerpt uses the `setCharacterStream` method to obtain a `java.io.Writer` object in order to initialize a `SQLXML` object:

```
SQLXML sqlxml = con.createSQLXML();
Writer out= sqlxml.setCharacterStream();
BufferedReader in = new BufferedReader(new FileReader("xml/foo.xml"));
String line = null;
while((line = in.readLine() != null) {
    out.write(line);
}
```

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

If an attempt is made to call the `setString`, `setBinaryStream`, `setCharacterStream`, and `setResult` methods on a `SQLXML` object that has previously been initialized, a `SQLException` will be thrown. If more than one call to the methods `setBinaryStream`, `setCharacterStream`, and `setResult` occurs for the same `SQLXML` object, a `SQLException` is thrown and the previously returned `javax.xml.transform.Result`, `Writer`, or `OutputStream` object is not affected.

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 the following excerpt, the method SQLXML.free is called to release the resources held for a previously created SQLXML object.

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

Sample Code

MySQL and Java DB and their respective JDBC drivers do not fully support the `SQLXML` JDBC data type as described on in this section. However, the sample `RSSFeedsTable` demonstrates how to handle XML data with MySQL and Java DB.
The owner of The Coffee Break follows several RSS feeds from various web sites that cover restaurant and beverage industry news. An RSS (Really Simple Syndication or Rich Site Summary) feed is an XML document that contains a series of articles and associated metadata, such as the date of publication and author for each article. The owner would like to store these RSS feeds into a database table, including the RSS feed from The Coffee Break's blog.
The file `rss-the-coffee-break-blog.xml` is an example RSS feed from The Coffee Break's blog.

Working with XML Data in MySQL

The sample `RSSFeedsTable` stores RSS feeds in the table `RSS_FEEDS`, which is created with the following command:

```
create table RSS_FEEDS
    (RSS_NAME varchar(32) NOT NULL,
    RSS_FEED_XML longtext NOT NULL,
    PRIMARY KEY (RSS_NAME));
```

MySQL does not support the XML data type. Instead, this sample stores XML data in a column of type `LONGTEXT`, which is a `CLOB` SQL data type. MySQL has four `CLOB` data types; the `LONGTEXT` data type holds the greatest amount of characters among the four.
The method `RSSFeedsTable.addRSSFeed` adds an RSS feed to the `RSS_FEEDS` table. The first statements of this method converts the RSS feed (which is represented by an XML file in this sample) into an object of type `org.w3c.dom.Document`, which represents a DOM (Document Object Model) document. This class, along with classes and interfaces contained in the package `javax.xml`, contain methods that enable you to manipulate XML data content. For example, the following statement uses an XPath expression to retrieve the title of the RSS feed from the `Document` object:

```
Node titleElement =
    (Node)xPath.evaluate("/rss/channel/title[1]",
        doc, XPathConstants.NODE);
```

The XPath expression `/rss/channel/title[1]` retrieves the contents of the first `<title>` element. For the file `rss-the-coffee-break-blog.xml`, this is the string `The Coffee Break Blog`.
The following statements add the RSS feed to the table `RSS_FEEDS`:

```
// For databases that support the SQLXML
// data type, this creates a
// SQLXML object from
// org.w3c.dom.Document.

System.out.println("Adding XML file " + fileName);
String insertRowQuery =
    "insert into RSS_FEEDS " +
    "(RSS_NAME, RSS_FEED_XML) values " +
    "(?, ?)";
insertRow = con.prepareStatement(insertRowQuery);
insertRow.setString(1, titleString);

System.out.println("Creating SQLXML object with MySQL");
rssData = con.createSQLXML();
System.out.println("Creating DOMResult object");
DOMResult dom = (DOMResult)rssData.setResult(DOMResult.class);
dom.setNode(doc);

insertRow.setSQLXML(2, rssData);
System.out.println("Running executeUpdate()");
insertRow.executeUpdate();
```

The `RSSFeedsTable.viewTable` method retrieves the contents of `RSS_FEEDS`. For each row, the method creates an object of type `org.w3c.dom.Document` named `doc` in which to store the XML content in the column `RSS_FEED_XML`. The method retrieves the XML content and stores it in an object of type `SQLXML` named `rssFeedXML`. The contents of `rssFeedXML` are parsed and stored in the `doc` object.

Working with XML Data in Java DB

Note: See the section "XML data types and operators" in *Java DB Developer's Guide* for more information about working with XML data in Java DB.

The sample `RSSFeedsTable` stores RSS feeds in the table `RSS_FEEDS`, which is created with the following command:

```
create table RSS_FEEDS
    (RSS_NAME varchar(32) NOT NULL,
    RSS_FEED_XML xml NOT NULL,
    PRIMARY KEY (RSS_NAME));
```

Java DB supports the XML data type, but it does not support the `SQLXML` JDBC data type. Consequently, you must convert any XML data to a character format, and then use the Java DB operator `XMLPARSE` to convert it to the XML data type.

The `RSSFeedsTable.addRSSFeed` method adds an RSS feed to the `RSS_FEEDS` table. The first statements of this method convert the RSS feed (which is represented by an XML file in this sample) into an object of type `org.w3c.dom.Document`. This is described in the section Working with XML Data in MySQL.

The `RSSFeedsTable.addRSSFeed` method converts the RSS feed to a `String` object with the method `JDBCTutorialUtilities.convertDocumentToString`.

Java DB has an operator named `XMLPARSE` that parses a character string representation into a Java DB XML value, which is demonstrated by the following excerpt:

```
String insertRowQuery =
    "insert into RSS_FEEDS " +
    "(RSS_NAME, RSS_FEED_XML) values " +
    "(?, xmlparse(document cast " +
    "(? as clob) preserve whitespace))";
```

The `XMLPARSE` operator requires that you convert the character representation of the XML document into a string data type that Java DB recognizes. In this example, it converts it into a `CLOB` data type. See Getting Started and the Java DB documentation for more information about Apache Xalan and Java DB requirements.

The method `RSSFeedsTable.viewTable` retrieves the contents of `RSS_FEEDS`. Because Java DB does not support the JDBC data type `SQLXML` you must retrieve the XML content as a string. Java DB has an operator named `XMLSERIALIZE` that converts an XML type to a character type:

```
String query =
    "select RSS_NAME, " +
    "xmlserialize " +
    "(RSS_FEED_XML as clob) " +
    "from RSS_FEEDS";
```

As with the `XMLPARSE` operator, the `XMLSERIALIZE` operator requires that Apache Xalan be listed in your Java class path.

Using Array Objects

Note: MySQL and Java DB currently do not support the `ARRAY` SQL data type. Consequently, no JDBC tutorial example is available to demonstrate the `Array` JDBC data type.
The following topics are covered:

Creating Array Objects

Use the method `Connection.createArrayOf` to create `Array` objects.
For example, suppose your database contains a table named `REGIONS`, which has been created and populated with the following SQL statements; note that the syntax of these statements will vary depending on your database:

```
create table REGIONS
    (REGION_NAME varchar(32) NOT NULL,
    ZIPS varchar32 ARRAY[10] NOT NULL,
    PRIMARY KEY (REGION_NAME));

insert into REGIONS values(
    'Northwest',
    '{"93101", "97201", "99210"}');
insert into REGIONS values(
    'Southwest',
    '{"94105", "90049", "92027"}');

Connection con = DriverManager.getConnection(url, props);
String [] northEastRegion = { "10022", "02110", "07399" };
Array aArray = con.createArrayOf("VARCHAR", northEastRegionnewYork);
```

The Oracle Database JDBC driver implements the `java.sql.Array` interface with the `oracle.sql.ARRAY` class.

Retrieving and Accessing Array Values in ResultSet

As with the JDBC 4.0 large object interfaces (`Blob`, `Clob`, `NClob`), you can manipulate `Array` objects without having to bring all of their data from the database server to your client computer. An `Array` object materializes the SQL `ARRAY` it represents as either a result set or a Java array.

The following excerpt retrieves the SQL `ARRAY` value in the column `ZIPS` and assigns it to the `java.sql.Array` object `z` object. The excerpt retrieves the contents of `z` and stores it in `zips`, a Java array that contains objects of type `String`. The excerpt iterates through the `zips` array and checks that each postal (zip) code is valid. This code assumes that the class `ZipCode` has been defined previously with the method `isValid` returning `true` if the given zip code matches one of the zip codes in a master list of valid zip codes:

```
ResultSet rs = stmt.executeQuery(
    "SELECT region_name, zips FROM REGIONS");

while (rs.next()) {
    Array z = rs.getArray("ZIPS");
    String[] zips = (String[])z.getArray();
    for (int i = 0; i < zips.length; i++) {
        if (!ZipCode.isValid(zips[i])) {
            // ...
            // Code to display warning
        }
    }
}
```

In the following statement, the `ResultSet` method `getArray` returns the value stored in the column `ZIPS` of the current row as the `java.sql.Array` object `z`:

```
Array z = rs.getArray("ZIPS");
```

The variable *`z`* contains a locator, which is a logical pointer to the SQL `ARRAY` on the server; it does not contain the elements of the `ARRAY` itself. Being a logical pointer, *`z`* can be used to manipulate the array on the server.

In the following line, `getArray` is the `Array.getArray` method, not the `ResultSet.getArray` method used in the previous line. Because the `Array.getArray` method returns an `Object` in the Java programming language and because each zip code is a `String` object, the result is cast to an array of `String` objects before being assigned to the variable `zips`.

```
String[] zips = (String[])z.getArray();
```

The `Array.getArray` method materializes the SQL `ARRAY` elements on the client as an array of `String` objects. Because, in effect, the variable *`zips`* contains the elements of the array, it is possible to iterate through `zips` in a `for` loop, looking for zip codes that are not valid.

Storing and Updating Array Objects

Use the methods `PreparedStatement.setArray` and `PreparedStatement.setObject` to pass an `Array` value as an input parameter to a `PreparedStatement` object.
The following example sets the `Array` object `northEastRegion` (created in a previous example) as the second parameter to the PreparedStatement `pstmt`:

```
PreparedStatement pstmt = con.prepareStatement(
    "insert into REGIONS (region_name, zips) " + "VALUES (?, ?)");
pstmt.setString(1, "NorthEast");
pstmt.setArray(2, northEastRegion);
pstmt.executeUpdate();
```

Similarly, use the methods `PreparedStatement.updateArray` and `PreparedStatement.updateObject` to update a column in a table with an `Array` value.

Releasing Array Resources

`Array` 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 `Array` resources by invoking their `free` method.

In the following excerpt, the method `Array.free` is called to release the resources held for a previously created `Array` object.

```
Array aArray = con.createArrayOf("VARCHAR", northEastRegionnewYork);
// ...
aArray.free();
```

Using DISTINCT Data Type

Note: MySQL and Java DB currently do not support the `DISTINCT` SQL data type. Consequently, no JDBC tutorial example is available to demonstrate the features described in this section.

The `DISTINCT` data type behaves differently from the other advanced SQL data types. Being a user-defined type that is based on one of the already existing built-in types, it has no interface as its mapping in the Java programming language. Instead, the standard mapping for a `DISTINCT` data type is the Java type to which its underlying SQL data type maps.

To illustrate, create a `DISTINCT` data type and then see how to retrieve, set, or update it. Suppose you always use a two-letter abbreviation for a state and want to create a `DISTINCT` data type to be used for these abbreviations. You could define your new `DISTINCT` data type with the following SQL statement:

```
CREATE TYPE STATE AS CHAR(2);
```

Some databases use an alternate syntax for creating a `DISTINCT` data type, which is shown in the following line of code:

```
CREATE DISTINCT TYPE STATE AS CHAR(2);
```

If one syntax does not work, you can try the other. Alternatively, you can check the documentation for your driver to see the exact syntax it expects.

These statements create a new data type, `STATE`, which can be used as a column value or as the value for an attribute of a SQL structured type. Because a value of type `STATE` is in reality a value that is two `CHAR` types, you use the same method to retrieve it that you would use to retrieve a `CHAR` value, that is, `getString`. For example, assuming that the fourth column of `ResultSet` *`rs`* stores values of type `STATE`, the following line of code retrieves its value:

```
String state = rs.getString(4);
```

Similarly, you would use the method `setString` to store a `STATE` value in the database and the method `updateString` to modify its value.

Using Structured Objects

Note: MySQL and Java DB currently do not support user-defined types. Consequently, no JDBC tutorial example is available to demonstrate the features described in this section.
The following topics are covered:

Overview of Structured Types

SQL structured types and DISTINCT types are the two data types that a user can define in SQL. They are often referred to as UDTs (user-defined types), and you create them with a SQL CREATE TYPE statement.
Getting back to the example of The Coffee Break, suppose that the owner has been successful beyond all expectations and has been expanding with new branches. The owner has decided to add a STORES table to the database containing information about each establishment. STORES will have four columns:

- STORE_NO for each store's identification number
- LOCATION for its address
- COF_TYPES for the coffees it sells
- MGR for the name of the store manager

The owner makes the column LOCATION be a SQL structured type, the column COF_TYPES a SQL ARRAY, and the column MGR a REF(MANAGER), with MANAGER being a SQL structured type.
The first thing the owner must define the new structured types for the address and the manager. A SQL structured type is similar to structured types in the Java programming language in that it has members, called *attributes*, that may be any data type. The owner writes the following SQL statement to create the new data type ADDRESS:

```
CREATE TYPE ADDRESS
(
    NUM INTEGER,
    STREET VARCHAR(40),
    CITY VARCHAR(40),
    STATE CHAR(2),
    ZIP CHAR(5)
);
```

In this statement, the new type ADDRESS has five attributes, which are analogous to fields in a Java class. The attribute NUM is an INTEGER, the attribute STREET is a VARCHAR(40), the attribute CITY is a VARCHAR(40), the attribute STATE is a CHAR(2), and the attribute ZIP is a CHAR(5).
The following excerpt, in which con is a valid Connection object, sends the definition of ADDRESS to the database:

```
String createAddress =
    "CREATE TYPE ADDRESS " +
    "(NUM INTEGER, STREET VARCHAR(40), " +
    "CITY VARCHAR(40), STATE CHAR(2), ZIP CHAR(5))";
Statement stmt = con.createStatement();
stmt.executeUpdate(createAddress);
```

Now the ADDRESS structured type is registered with the database as a data type, and the owner can use it as the data type for a table column or an attribute of a structured type.

Using DISTINCT Type in Structured Type

One of the attributes the owner of The Coffee Break plans to include in the new structured type `MANAGER` is the manager's telephone number. Because the owner will always list the telephone number as a 10-digit number (to be sure it includes the area code) and will never manipulate it as a number, the owner decides to define a new type called `PHONE_NO` that consists of 10 characters. The SQL definition of this data type, which can be thought of as a structured type with only one attribute, looks like this:

```
CREATE TYPE PHONE_NO AS CHAR(10);
```

Or, as noted earlier, for some drivers the definition might look like this:

```
CREATE DISTINCT TYPE PHONE_NO AS CHAR(10);
```

A `DISTINCT` type is always based on another data type, which must be a predefined type. In other words, a `DISTINCT` type cannot be based on a user-defined type (UDT). To retrieve or set a value that is a `DISTINCT` type, use the appropriate method for the underlying type (the type on which it is based). For example, to retrieve an instance of `PHONE_NO`, which is based on a `CHAR` type, you would use the method `getString` because that is the method for retrieving a `CHAR`.

Assuming that a value of type `PHONE_NO` is in the fourth column of the current row of the `ResultSet` object *`rs`*, the following line of code retrieves it:

```
String phoneNumber = rs.getString(4);
```

Similarly, the following line of code sets an input parameter that has type `PHONE_NO` for a prepared statement being sent to the database:

```
pstmt.setString(1, phoneNumber);
```

Adding on to the previous code fragment, the definition of `PHONE_NO` will be sent to the database with the following line of code:

```
stmt.executeUpdate(
    "CREATE TYPE PHONE_NO AS CHAR(10)");
```

After registering the type `PHONE_NO` with the database, the owner can use it as a column type in a table or as the data type for an attribute in a structured type. The definition of `MANAGER` in the following SQL statement uses `PHONE_NO` as the data type for the attribute `PHONE`:

```
CREATE TYPE MANAGER
(
    MGR_ID INTEGER,
    LAST_NAME VARCHAR(40),
    FIRST_NAME VARCHAR(40),
    PHONE PHONE_NO
);
```

Reusing *`stmt`*, defined previously, the following code fragment sends the definition of the structured type `MANAGER` to the database:

```
  String createManager =
    "CREATE TYPE MANAGER " +
    "(MGR_ID INTEGER, LAST_NAME " +
    "VARCHAR(40), " +
    "FIRST_NAME VARCHAR(40), " +
    "PHONE PHONE_NO)";
  stmt.executeUpdate(createManager);
```

Using References to Structured Types

The owner of The Coffee Break has created three new data types used as column types or attribute types in the database: The structured types LOCATION and MANAGER, and the DISTINCT type PHONE_NO. The entrepreneur has used PHONE_NO as the type for the attribute PHONE in the new type MANAGER, and ADDRESS as the data type for the column LOCATION in the table STORES. The MANAGER type could be used as the type for the column MGR, but instead the entrepreneur prefers to use the type REF(MANAGER) because the entrepreneur often has one person manage two or three stores. Using REF(MANAGER) as a column type avoids repeating all the data for MANAGER when one person manages more than one store.

With the structured type MANAGER already created, the owner can now create a table containing instances of MANAGER that can be referenced. A reference to an instance of MANAGER will have the type REF(MANAGER). A SQL REF is nothing more than a logical pointer to a structured type, so an instance of REF(MANAGER) serves as a logical pointer to an instance of MANAGER.

Because a SQL REF value needs to be permanently associated with the instance of the structured type that it references, it is stored in a special table together with its associated instance. A programmer does not create REF types directly but rather creates the table that will store instances of a particular structured type that can be referenced. Every structured type that is to be referenced will have its own table. When you insert an instance of the structured type into the table, the database automatically creates a REF instance. For example, to contain instances of MANAGER that can be referenced, the owner created the following special table using SQL:

```
CREATE TABLE MANAGERS OF MANAGER
(OID REF(MANAGER)
VALUES ARE SYSTEM GENERATED);
```

This statement creates a table with the special column OID, which stores values of type REF(MANAGER). Each time an instance of MANAGER is inserted into the table, the database will generate an instance of REF(MANAGER) and store it in the column OID. Implicitly, an additional column stores each attribute of MANAGER that has been inserted into the table, as well. For example, the following code fragment shows how the entrepreneur created three instances of the MANAGER structured type to represent three managers:

```
INSERT INTO MANAGERS (
  MGR_ID, LAST_NAME,
  FIRST_NAME, PHONE) VALUES
(
  000001,
  'MONTOYA',
  'ALFREDO',
  '8317225600'
);

INSERT INTO MANAGERS (
  MGR_ID, LAST_NAME,
  FIRST_NAME, PHONE) VALUES
(
  000002,
  'HASKINS',
  'MARGARET',
  '4084355600'
);

INSERT INTO MANAGERS (
  MGR_ID, LAST_NAME,
  FIRST_NAME, PHONE) VALUES
(
  000003,
  'CHEN',
  'HELEN',
  '4153785600'
 );
```

The table MANAGERS will now have three rows, one row for each manager inserted so far. The column OID will contain three unique object identifiers of type REF(MANAGER), one for each instance of MANAGER. These object identifiers were generated automatically by the database and will be permanently stored in the table MANAGERS. Implicitly, an additional column stores each attribute of MANAGER. For example, in the table MANAGERS, one row contains a REF(MANAGER) that references Alfredo Montoya, another row contains a REF(MANAGER) that references Margaret Haskins, and a third row contains a REF(MANAGER) that references Helen Chen.

To access a `REF(MANAGER)` instance, you select it from its table. For example, the owner retrieved the reference to Alfredo Montoya, whose ID number is 000001, with the following code fragment:

```
String selectMgr =
  "SELECT OID FROM MANAGERS " +
  "WHERE MGR_ID = 000001";
ResultSet rs = stmt.executeQuery(selectMgr);
rs.next();
Ref manager = rs.getRef("OID");
```

Now the variable *`manager`* can be used as a column value that references Alfredo Montoya.

Sample Code for Creating SQL REF Object

The following code example creates the table `MANAGERS`, a table of instances of the structured type `MANAGER` that can be referenced, and inserts three instances of `MANAGER` into the table. The column `OID` in this table will store instances of `REF(MANAGER)`. After this code is executed, the `MANAGERS` table will have a row for each of the three `MANAGER` objects inserted, and the value in the `OID` column will be the `REF(MANAGER)` type that identifies the instance of `MANAGER` stored in that row.

```
package com.oracle.tutorial.jdbc;

import java.sql.*;

public class CreateRef {

    public static void main(String args[]) {

        JDBCTutorialUtilities myJDBCTutorialUtilities;
        Connection myConnection = null;

        if (args[0] == null) {
            System.err.println("Properties file not specified " +
                               "at command line");
            return;
        } else {
            try {
                myJDBCTutorialUtilities = new JDBCTutorialUtilities(args[0]);
            } catch (Exception e) {
                System.err.println("Problem reading properties " +
                                   "file " + args[0]);
                e.printStackTrace();
                return;
            }
        }

        Connection con = null;
        Statement stmt = null;

        try {
            String createManagers =
                "CREATE TABLE " +
                "MANAGERS OF MANAGER " +
                "(OID REF(MANAGER) " +
                "VALUES ARE SYSTEM " +
                "GENERATED)";

            String insertManager1 =
                "INSERT INTO MANAGERS " +
                "(MGR_ID, LAST_NAME, " +
                "FIRST_NAME, PHONE) " +
                "VALUES " +
                "(000001, 'MONTOYA', " +
                "'ALFREDO', " +
                "'8317225600')";

            String insertManager2 =
                "INSERT INTO MANAGERS " +
                "(MGR_ID, LAST_NAME, " +
                "FIRST_NAME, PHONE) " +
                "VALUES " +
                "(000002, 'HASKINS', " +
                "'MARGARET', " +
                "'4084355600')";

            String insertManager3 =
                "INSERT INTO MANAGERS " +
                "(MGR_ID, LAST_NAME, " +
                "FIRST_NAME, PHONE) " +
                "VALUES " +
                "(000003, 'CHEN', 'HELEN', " +
                "'4153785600')";

            con = myJDBCTutorialUtilities.getConnection();
            con.setAutoCommit(false);

            stmt = con.createStatement();
            stmt.executeUpdate(createManagers);

            stmt.addBatch(insertManager1);
            stmt.addBatch(insertManager2);
            stmt.addBatch(insertManager3);
            int [] updateCounts = stmt.executeBatch();

            con.commit();

            System.out.println("Update count for:  ");
            for (int i = 0; i < updateCounts.length; i++) {
                System.out.print("    command " + (i + 1) + " = ");
                System.out.println(updateCounts[i]);
            }
        } catch(BatchUpdateException b) {
            System.err.println("-----BatchUpdateException-----");
            System.err.println("Message:  " + b.getMessage());
            System.err.println("SQLState:  " + b.getSQLState());
            System.err.println("Vendor:  " + b.getErrorCode());
            System.err.print("Update counts for " + "successful commands:  ");
            int [] rowsUpdated = b.getUpdateCounts();
            for (int i = 0; i < rowsUpdated.length; i++) {
                System.err.print(rowsUpdated[i] + "   ");
            }
            System.err.println("");
        } catch(SQLException ex) {
            System.err.println("------SQLException------");
            System.err.println("Error message:  " + ex.getMessage());
            System.err.println("SQLState:  " + ex.getSQLState());
            System.err.println("Vendor:  " + ex.getErrorCode());
        } finally {
            if (stmt != null) { stmt.close(); }
              JDBCTutorialUtilities.closeConnection(con);
        }
```

```
    }
}
```

Using User-Defined Types as Column Values

Our entrepreneur now has the UDTs required to create the table `STORES`. The structured type `ADDRESS` is the type for the column `LOCATION`, and the type `REF(MANAGER)` is the type for the column `MGR`.
The UDT `COF_TYPES` is based on the SQL data type `ARRAY` and is the type for the column `COF_TYPES`. The following line of code creates the type `COF_ARRAY` as an `ARRAY` value with 10 elements. The base type of `COF_ARRAY` is `VARCHAR(40)`.

```
CREATE TYPE COF_ARRAY AS ARRAY(10) OF VARCHAR(40);
```

With the new data types defined, the following SQL statement creates the table `STORES`:

```
CREATE TABLE STORES
(
  STORE_NO INTEGER,
  LOCATION ADDRESS,
  COF_TYPES COF_ARRAY,
  MGR REF(MANAGER)
);
```

Inserting User-Defined Types into Tables

The following code fragment inserts one row into the STORES table, supplying values for the columns STORE_NO, LOCATION, COF_TYPES, and MGR, in that order:

```
INSERT INTO STORES VALUES
(
  100001,
  ADDRESS(888, 'Main_Street',
    'Rancho_Alegre',
    'CA', '94049'),
  COF_ARRAY('Colombian', 'French_Roast',
    'Espresso', 'Colombian_Decaf',
    'French_Roast_Decaf'),
  SELECT OID FROM MANAGERS
    WHERE MGR_ID = 000001
);
```

The following goes through each column and the value inserted into it.

```
STORE_NO: 100001
```

This column is type INTEGER, and the number 100001 is an INTEGER type, similar to entries made before in the tables COFFEES and SUPPLIERS.

```
LOCATION: ADDRESS(888, 'Main_Street',
  'Rancho_Alegre', 'CA', '94049')
```

The type for this column is the structured type ADDRESS, and this value is the constructor for an instance of ADDRESS. When we sent the definition of ADDRESS was sent to the database, one of the things it did was to create a constructor for the new type. The comma-separated values in parentheses are the initialization values for the attributes of the ADDRESS type, and they must appear in the same order in which the attributes were listed in the definition of the ADDRESS type. 888 is the value for the attribute NUM, which is an INTEGER value. "Main_Street" is the value for STREET, and "Rancho_Alegre" is the value for CITY, with both attributes being of type VARCHAR(40). The value for the attribute STATE is "CA", which is of type CHAR(2), and the value for the attribute ZIP is "94049", which is of type CHAR(5).

```
COF_TYPES: COF_ARRAY(
  'Colombian',
  'French_Roast',
  'Espresso',
  'Colombian_Decaf',
  'French_Roast_Decaf'),
```

The column COF_TYPES is of type COF_ARRAY with a base type of VARCHAR(40), and the comma-separated values between parentheses are the String objects that are the array elements. The owner defined the type COF_ARRAY as having a maximum of 10 elements. This array has 5 elements because the entrepreneur supplied only 5 String objects for it.

```
MGR: SELECT OID FROM MANAGERS
  WHERE MGR_ID = 000001
```

The column MGR is type REF(MANAGER), which means that a value in this column must be a reference to the structured type MANAGER. All of the instances of MANAGER are stored in the table MANAGERS. All of the instances of REF(MANAGER) are also stored in this table, in the column OID. The manager for the store described in this table row is Alfredo Montoya, and his information is stored in the instance of MANAGER that has 100001 for the attribute MGR_ID. To get the REF(MANAGER) instance associated with the MANAGER object for Alfredo Montoya, select the column OID that is in the row where MGR_ID is 100001 in the table MANAGERS. The value that will be stored in the MGR column of the STORES table (the REF(MANAGER) value) is the value the DBMS generated to uniquely identify this instance of the MANAGER structured type.

Send the preceding SQL statement to the database with the following code fragment:

```
String insertMgr =
  "INSERT INTO STORES VALUES " +
  "(100001, " +
  "ADDRESS(888, 'Main_Street', " +
    "'Rancho_Alegre', 'CA', " +
    "'94049'), " +
  "COF_ARRAY('Colombian', " +
    "'French_Roast', 'Espresso', " +
    "'Colombian_Decaf', " +
    "'French_Roast_Decaf'}, " +
  "SELECT OID FROM MANAGERS " +
  "WHERE MGR_ID = 000001)";

stmt.executeUpdate(insertMgr);
```

However, because you are going to send several `INSERT INTO` statements, it will be more efficient to send them all together as a batch update, as in the following code example:

```
package com.oracle.tutorial.jdbc;

import java.sql.*;

public class InsertStores {
    public static void main(String args[]) {

        JDBCTutorialUtilities myJDBCTutorialUtilities;
        Connection myConnection = null;

        if (args[0] == null) {
            System.err.println(
                "Properties file " +
                "not specified " +
                "at command line");
            return;
        } else {
            try {
                myJDBCTutorialUtilities = new
                    JDBCTutorialUtilities(args[0]);
            } catch (Exception e) {
                System.err.println(
                    "Problem reading " +
                    "properties file " +
                    args[0]);
                e.printStackTrace();
                return;
            }
        }

        Connection con = null;
        Statement stmt = null;

        try {
            con = myJDBCTutorialUtilities.getConnection();
            con.setAutoCommit(false);

            stmt = con.createStatement();

            String insertStore1 =
                "INSERT INTO STORES VALUES (" +
                "100001, " +
                "ADDRESS(888, 'Main_Street', " +
                    "'Rancho_Alegre', 'CA', " +
                    "'94049'), " +
                "COF_ARRAY('Colombian', " +
                    "'French_Roast', " +
                    "'Espresso', " +
                    "'Colombian_Decaf', " +
                    "'French_Roast_Decaf'), " +
                "(SELECT OID FROM MANAGERS " +
                "WHERE MGR_ID = 000001))";

            stmt.addBatch(insertStore1);

            String insertStore2 =
                "INSERT INTO STORES VALUES (" +
                "100002, " +
                "ADDRESS(1560, 'Alder', " +
                    "'Ochos_Pinos', " +
                    "'CA', '94049'), " +
                "COF_ARRAY('Colombian', " +
                    "'French_Roast', " +
                    "'Espresso', " +
                    "'Colombian_Decaf', " +
                    "'French_Roast_Decaf', " +
                    "'Kona', 'Kona_Decaf'), " +
                "(SELECT OID FROM MANAGERS " +
                "WHERE MGR_ID = 000001))";

            stmt.addBatch(insertStore2);

            String insertStore3 =
                "INSERT INTO STORES VALUES (" +
                "100003, " +
                "ADDRESS(4344, " +
                    "'First_Street', " +
                    "'Verona', " +
                    "'CA', '94545'), " +
                "COF_ARRAY('Colombian', " +
                    "'French_Roast', " +
                    "'Espresso', " +
                    "'Colombian_Decaf', " +
                    "'French_Roast_Decaf', " +
                    "'Kona', 'Kona_Decaf'), " +
                "(SELECT OID FROM MANAGERS " +
                "WHERE MGR_ID = 000002))";

            stmt.addBatch(insertStore3);

            String insertStore4 =
                "INSERT INTO STORES VALUES (" +
                "100004, " +
                "ADDRESS(321, 'Sandy_Way', " +
                    "'La_Playa', " +
                    "'CA', '94544'), " +
                "COF_ARRAY('Colombian', " +
                    "'French_Roast', " +
                    "'Espresso', " +
                    "'Colombian_Decaf', " +
                    "'French_Roast_Decaf', " +
                    "'Kona', 'Kona_Decaf'), " +
                "(SELECT OID FROM MANAGERS " +
                "WHERE MGR_ID = 000002))";

            stmt.addBatch(insertStore4);

            String insertStore5 =
                "INSERT INTO STORES VALUES (" +
                "100005, " +
                "ADDRESS(1000, 'Clover_Road', " +
                    "'Happyville', " +
                    "'CA', '90566'), " +
                "COF_ARRAY('Colombian', " +
```

```
                "'French_Roast', " +
                "'Espresso', " +
                "'Colombian_Decaf', " +
                "'French_Roast_Decaf'), " +
            "(SELECT OID FROM MANAGERS " +
            "WHERE MGR_ID = 000003))";

        stmt.addBatch(insertStore5);

        int [] updateCounts = stmt.executeBatch();

        ResultSet rs = stmt.executeQuery(
            "SELECT * FROM STORES");
        System.out.println("Table STORES after insertion:");
        System.out.println("STORE_NO   " + "LOCATION   " +
            "COF_TYPE   " + "MGR");

        while (rs.next()) {
            int storeNo = rs.getInt("STORE_NO");
            Struct location = (Struct)rs.getObject("LOCATION");
            Object[] locAttrs = location.getAttributes();
            Array coffeeTypes = rs.getArray("COF_TYPE");
            String[] cofTypes = (String[])coffeeTypes.getArray();

            Ref managerRef = rs.getRef("MGR");
            PreparedStatement pstmt = con.prepareStatement(
                "SELECT MANAGER " +
                "FROM MANAGERS " +
                "WHERE OID = ?");

            pstmt.setRef(1, managerRef);
            ResultSet rs2 = pstmt.executeQuery();
            rs2.next();
            Struct manager = (Struct)rs2.getObject("MANAGER");
            Object[] manAttrs = manager.getAttributes();

            System.out.print(storeNo + "   ");
            System.out.print(
                locAttrs[0] + " " +
                locAttrs[1] + " " +
                locAttrs[2] + ", " +
                locAttrs[3] + " " +
                locAttrs[4] + " ");

            for (int i = 0; i < cofTypes.length; i++)
                System.out.print( cofTypes[i] + " ");

            System.out.println(
                manAttrs[1] + ", " +
                manAttrs[2]);

            rs2.close();
            pstmt.close();
        }

        rs.close();

    } catch(BatchUpdateException b) {
        System.err.println("-----BatchUpdateException-----");
        System.err.println("SQLState:  " + b.getSQLState());
        System.err.println("Message:  " + b.getMessage());
        System.err.println("Vendor:  " + b.getErrorCode());
        System.err.print("Update counts:  ");
        int [] updateCounts = b.getUpdateCounts();

        for (int i = 0; i < updateCounts.length; i++) {
            System.err.print(updateCounts[i] + "   ");
        }
        System.err.println("");

    } catch(SQLException ex) {
        System.err.println("SQLException: " + ex.getMessage());
        System.err.println("SQLState:  " + ex.getSQLState());
        System.err.println("Message:  " + ex.getMessage());
        System.err.println("Vendor:  " + ex.getErrorCode());
    } finally {
        if (stmt != null) { stmt.close(); }
            JDBCTutorialUtilities.closeConnection(con);
        }
    }
  }
}
```

Using Customized Type Mappings

Note: MySQL currently does not support user-defined types. MySQL and Java DB currently do not support structured types, or the `DISTINCT` SQL data type. No JDBC tutorial example is available to demonstrate the features described in this section.

With business booming, the owner of The Coffee Break is regularly adding new stores and making changes to the database. The owner has decided to use a custom mapping for the structured type `ADDRESS`. This enables the owner to make changes to the Java class that maps the `ADDRESS` type. The Java class will have a field for each attribute of `ADDRESS`. The name of the class and the names of its fields can be any valid Java identifier.

The following topics are covered:

Implementing SQLData

The first thing required for a custom mapping is to create a class that implements the interface `SQLData`.
The SQL definition of the structured type `ADDRESS` looks like this:

```
  CREATE TYPE ADDRESS
  (
    NUM INTEGER,
    STREET VARCHAR(40),
    CITY VARCHAR(40),
    STATE CHAR(2),
    ZIP CHAR(5)
  );
```

A class that implements the `SQLData` interface for the custom mapping of the `ADDRESS` type might look like this:

```
public class Address implements SQLData {
    public int num;
    public String street;
    public String city;
    public String state;
    public String zip;
    private String sql_type;

    public String getSQLTypeName() {
        return sql_type;
    }

    public void readSQL(SQLInput stream, String type)
        throws SQLException {
        sql_type = type;
        num = stream.readInt();
        street = stream.readString();
        city = stream.readString();
        state = stream.readString();
        zip = stream.readString();
    }

    public void writeSQL(SQLOutput stream)
        throws SQLException {
        stream.writeInt(num);
        stream.writeString(street);
        stream.writeString(city);
        stream.writeString(state);
        stream.writeString(zip);
    }
}
```

Using a Connection's Type Map

After writing a class that implements the interface `SQLData`, the only other thing you have to do to set up a custom mapping is to make an entry in a type map. For the example, this means entering the fully qualified SQL name for the `ADDRESS` type and the `Class` object for the class `Address`. A type map, an instance of the `java.util.Map` interface, is associated with every new connection when it is created, so you use that one. Assuming that `con` is the active connection, the following code fragment adds an entry for the UDT `ADDRESS` to the type map associated with `con`.

```
java.util.Map map = con.getTypeMap();
map.put("SchemaName.ADDRESS", Class.forName("Address"));
con.setTypeMap(map);
```

Whenever you call the `getObject` method to retrieve an instance of the `ADDRESS` type, the driver will check the type map associated with the connection and see that it has an entry for `ADDRESS`. The driver will note the `Class` object for the `Address` class, create an instance of it, and do many other things in the background to map `ADDRESS` to `Address`. You do not have to do anything more than generate the class for the mapping and then make an entry in a type map to let the driver know that there is a custom mapping. The driver will do all the rest.

The situation is similar for storing a structured type that has a custom mapping. When you call the method `setObject`, the driver will check to see if the value to be set is an instance of a class that implements the interface `SQLData`. If it is (meaning that there is a custom mapping), the driver will use the custom mapping to convert the value to its SQL counterpart before returning it to the database. Again, the driver does the custom mapping behind the scenes; all you need to do is supply the method `setObject` with a parameter that has a custom mapping. You will see an example of this later in this section.

Look at the difference between working with the standard mapping, a `Struct` object, and the custom mapping, a class in the Java programming language. The following code fragment shows the standard mapping to a `Struct` object, which is the mapping the driver uses when there is no entry in the connection's type map.

```
ResultSet rs = stmt.executeQuery(
    "SELECT LOCATION " +
    "WHERE STORE_NO = 100003");
rs.next();
Struct address = (Struct)rs.getObject("LOCATION");
```

The variable `address` contains the following attribute values: `4344`, `"First_Street"`, `"Verona"`, `"CA"`, `"94545"`.

The following code fragment shows what happens when there is an entry for the structured type `ADDRESS` in the connection's type map. Remember that the column `LOCATION` stores values of type `ADDRESS`.

```
ResultSet rs = stmt.executeQuery(
    "SELECT LOCATION " +
    "WHERE STORE_NO = 100003");
rs.next();
Address store_3 = (Address)rs.getObject("LOCATION");
```

The variable `store_3` is now an instance of the class `Address`, with each attribute value being the current value of one of the fields of `Address`. Note that you must remember to convert the object retrieved by the `getObject` method to an `Address` object before assigning it to `store_3`. Note also that `store_3` must be an `Address` object.

Compare working with the `Struct` object to working with the instance of the `Address` class. Suppose the store moved to a better location in the neighboring town and therefore you must update the database. With the custom mapping, reset the fields of `store_3`, as in the following code fragment:

```
ResultSet rs = stmt.executeQuery(
    "SELECT LOCATION " +
    "WHERE STORE_NO = 100003");
rs.next();
Address store_3 = (Address)rs.getObject("LOCATION");
store_3.num = 1800;
store_3.street = "Artsy_Alley";
store_3.city = "Arden";
```

```
store_3.state = "CA";
store_3.zip = "94546";
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE STORES " +
    "SET LOCATION = ? " +
    "WHERE STORE_NO = 100003");
pstmt.setObject(1, store_3);
pstmt.executeUpdate();
```

Values in the column `LOCATION` are instances of the `ADDRESS` type. The driver checks the connection's type map and sees that there is an entry linking `ADDRESS` with the class `Address` and consequently uses the custom mapping indicated in `Address`. When the code calls the method `setObject` with the variable *`store_3`* as the second parameter, the driver checks and sees that *`store_3`* represents an instance of the class `Address`, which implements the interface `SQLData` for the structured type `ADDRESS`, and again automatically uses the custom mapping.

Without a custom mapping for `ADDRESS`, the update would look more like this:

```
PreparedStatement pstmt = con.prepareStatement(
    "UPDATE STORES " +
    "SET LOCATION.NUM = 1800, " +
    "LOCATION.STREET = 'Artsy_Alley', " +
    "LOCATION.CITY = 'Arden', " +
    "LOCATION.STATE = 'CA', " +
    "LOCATION.ZIP = '94546' " +
    "WHERE STORE_NO = 100003");
pstmt.executeUpdate;
```

Using Your Own Type Map

Up to this point, you have used only the type map associated with a connection for custom mapping. Ordinarily, that is the only type map most programmers will use. However, it is also possible to create a type map and pass it to certain methods so that the driver will use that type map instead of the one associated with the connection. This allows two different mappings for the same user-defined type (UDT). In fact, it is possible to have multiple custom mappings for the same UDT, just as long as each mapping is set up with a class implementing the `SQLData` interface and an entry in a type map. If you do not pass a type map to a method that can accept one, the driver will by default use the type map associated with the connection.

There are very few situations that call for using a type map other than the one associated with a connection. It could be necessary to supply a method with a type map if, for instance, several programmers working on a JDBC application brought their components together and were using the same connection. If two or more programmers had created their own custom mappings for the same SQL UDT, each would need to supply his or her own type map, thus overriding the connection's type map.

Using Datalink Objects

A `DATALINK` value references a resource outside the underlying data source through a URL. A URL, uniform resource locator, is a pointer to a resource on the World Wide Web. A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a query to a database or to a search engine.
The following topics are covered:

Storing References to External Data

Use the method `PreparedStatement.setURL` to specify 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, store the URL with the `setString` method.

For example, suppose the owner of The Coffee Break would like to store a list of important URLs in a database table. The following example, `DatalinkSample.addURLRow` adds one row of data to the table `DATA_REPOSITORY`. The row consists of a string identifying the URL, `DOCUMENT_NAME` and the URL itself, `URL`:

```
public void addURLRow(String description, String url)
    throws SQLException {

    PreparedStatement pstmt = null;

    try {
        pstmt = this.con.prepareStatement(
            "INSERT INTO data_repository" +
            "(document_name,url) VALUES (?,?)");

        pstmt.setString(1, description);
        pstmt.setURL(2,new URL(url));
        pstmt.execute();
    } catch (SQLException sqlex) {
        JDBCTutorialUtilities.printSQLException(sqlex);
    } catch (Exception ex) {
        System.out.println("Unexpected exception");
        ex.printStackTrace();
    } finally {
        if (pstmt != null) {
            pstmt.close();
        }
    }
}
```

Retrieving References to External Data

Use the method `ResultSet.getURL` to retrieve a reference to external data as a `java.net.URL` object. In cases where the type of URL returned by the methods `getObject` or `getURL` is not supported by the Java platform, retrieve the URL as a `String` object by calling the method `getString`.
The following example, `DatalinkSample.viewTable`, displays the contents of all the URLs stored in the table `DATA_REPOSITORY`:

```
public static void viewTable(Connection con, Proxy proxy)
    throws SQLException, IOException {

    Statement stmt = null;
    String query =
      "SELECT document_name, url " +
      "FROM data_repository";

    try {
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(query);

        if ( rs.next() )  {
            String documentName = null;
            java.net.URL url = null;

            documentName = rs.getString(1);

            // Retrieve the value as a URL object.
            url = rs.getURL(2);

            if (url != null) {

                // Retrieve the contents
                // from the URL

                URLConnection myURLConnection =
                    url.openConnection(proxy);
                BufferedReader bReader =
                    new BufferedReader(
                        new InputStreamReader(
                            myURLConnection.
                                getInputStream()));

                System.out.println("Document name: " + documentName);

                String pageContent = null;

                while ((pageContent = bReader.readLine()) != null ) {
                    // Print the URL contents
                    System.out.println(pageContent);
                }
            } else {
                System.out.println("URL is null");
            }
        }
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } catch(IOException ioEx) {
        System.out.println("IOException caught: " + ioEx.toString());
    } catch (Exception ex) {
        System.out.println("Unexpected exception");
        ex.printStackTrace();
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

The sample `DatalinkSample` stores the Oracle URL, http://www.oracle.com in the table `DATA_REPOSITORY`. Afterward, it displays the contents of all documents referred to by the URLs stored in `DATA_REPOSITORY`, which includes the Oracle home page, http://www.oracle.com.
The sample retrieves the URL from the result set as a `java.net.URL` object with the following statement:

```
url = rs.getURL(2);
```

The sample accesses the data referred to by the `URL` object with the following statements:

```
URLConnection myURLConnection = url.openConnection(proxy);
BufferedReader bReader = new BufferedReader(
    new InputStreamReader(
        myURLConnection.getInputStream()));

System.out.println("Document name: " + documentName);

String pageContent = null;

while ((pageContent = bReader.readLine()) != null ) {
    // Print the URL contents
    System.out.println(pageContent);
}
```

The method `URLConnection.openConnection` can take no arguments, which means that the `URLConnection` represents a direct connection to the Internet. If you require a proxy server to connect to the Internet, the `openConnection` method accepts a `java.net.Proxy` object as an argument. The following statements demonstrate how to create an HTTP proxy with the server name `www-proxy.example.com` and port number `80`:

```
Proxy myProxy;
InetSocketAddress myProxyServer;
myProxyServer = new InetSocketAddress("www-proxy.example.com", 80);
myProxy = new Proxy(Proxy.Type.HTTP, myProxyServer);
```

Using RowId Objects

Note: MySQL and Java DB currently do not support the `RowId` JDBC interface. Consequently, no JDBC tutorial example is available to demonstrate the features described in this section.
A `RowId` object represents an address to a row in a database table. Note, however, that the `ROWID` type is not a standard SQL type. `ROWID` values can be useful because they are typically the fastest way to access a single row and are unique identifies for rows in a table. However, you should not use a `ROWID` value as the primary key of a table. For example, if you delete a particular row from a table, a database might reassign its `ROWID` value to a row inserted later.
The following topics are covered:

Retrieving RowId Objects

Retrieve a `java.sql.RowId` object by calling the getter methods defined in the interfaces `ResultSet` and `CallableStatement`. The `RowId` object that is returned is an immutable object that you can use for subsequent referrals as a unique identifier to a row. The following is an example of calling the `ResultSet.getRorId` method:

```
java.sql.RowId rowId_1 = rs.getRowId(1);
```

Using RowId Objects

You can set a `RowId` object as a parameter in a parameterized `PreparedStatement` object:

```
Connection conn = ds.getConnection(username, password);
PreparedStatement ps = conn.prepareStatement(
    "INSERT INTO BOOKLIST" +
    "(ID, AUTHOR, TITLE, ISBN) " +
    "VALUES (?, ?, ?, ?)");
ps.setRowId(1, rowId_1);
```

You can also update a column with a specific `RowId` object in an updatable `ResultSet` object:

```
ResultSet rs = ...
rs.next();
rs.updateRowId(1, rowId_1);
```

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` object from a `ResultSet` object with a connection to one data source and then attempt to use the same `RowId` object in a unrelated `ResultSet` object with a connection to a different data source.

Lifetime of RowId Validity

A `RowId` object is valid as long as the identified row is not deleted and the lifetime of the `RowId` object is within the bounds of the lifetime specified by that the data source for the `RowId`.

To determine the lifetime of `RowId` objects of your database or data source, call the method `DatabaseMetaData.getRowIdLifetime`. It returns a value of a `RowIdLifetime` enumerated data type. The following method `JDBCTutorialUtilities.rowIdLifeTime` returns the lifetime of `RowId` objects:

```
public static void rowIdLifetime(Connection conn)
    throws SQLException {

    DatabaseMetaData dbMetaData = conn.getMetaData();
    RowIdLifetime lifetime = dbMetaData.getRowIdLifetime();

    switch (lifetime) {
        case ROWID_UNSUPPORTED:
            System.out.println("ROWID type not supported");
            break;

        case ROWID_VALID_FOREVER:
            System.out.println("ROWID has unlimited lifetime");
            break;

        case ROWID_VALID_OTHER:
            System.out.println("ROWID has indeterminate lifetime");
            break;

        case ROWID_VALID_SESSION:
            System.out.println(
                "ROWID type has lifetime that " +
                "is valid for at least the " +
                "containing session");
            break;

        case ROWID_VALID_TRANSACTION:
            System.out.println(
                "ROWID type has lifetime that " +
                "is valid for at least the " +
                "containing transaction");
            break;
    }
}
```

Using Stored Procedures

A stored procedure is a group of SQL statements that form a logical unit and perform a particular task, and they are used to encapsulate a set of operations or queries to execute on a database server. For example, operations on an employee database (hire, fire, promote, lookup) could be coded as stored procedures executed by application code. Stored procedures can be compiled and executed with different parameters and results, and they can have any combination of input, output, and input/output parameters.

Note that stored procedures are supported by most DBMSs, but there is a fair amount of variation in their syntax and capabilities. Consequently, the tutorial contains two classes, `StoredProcedureJavaDBSample` and `StoredProcedureMySQLSample` to demonstrate how to create stored procedures in Java DB and MySQL, respectively.

This page covers the following topics:

Overview of Stored Procedures Examples

The examples `StoredProcedureJavaDBSample.java` and `StoredProcedureMySQLSample.java` create and call the following stored procedures:

- `SHOW_SUPPLIERS`: Prints a result set that contains the names of coffee suppliers and the coffees they supply to The Coffee Break. This stored procedure does not require any parameters. When the example calls this stored procedure, the example produces output similar to the following:

```
Acme, Inc.: Colombian_Decaf
Acme, Inc.: Colombian
Superior Coffee: French_Roast_Decaf
Superior Coffee: French_Roast
The High Ground: Espresso
```

- `GET_SUPPLIER_OF_COFFEE`: Prints the name of the supplier `supplierName` for the coffee `coffeeName`. It requires the following parameters:
  - `IN coffeeName varchar(32)`: The name of the coffee
  - `OUT supplierName varchar(40)`: The name of the coffee supplier

  When the example calls this stored procedure with `Colombian` as the value for `coffeeName`, the example produces output similar to the following:

```
Supplier of the coffee Colombian: Acme, Inc.
```

- `RAISE_PRICE`: Raises the price of the coffee `coffeeName` to the price `newPrice`. If the price increase is greater than the percentage `maximumPercentage`, then the price is raised by that percentage. This procedure will not change the price if the price `newPrice` is lower than the original price of the coffee. It requires the following parameters:
  - `IN coffeeName varchar(32)`: The name of the coffee
  - `IN maximumPercentage float`: The maximum percentage to raise the coffee's price
  - `INOUT newPrice numeric(10,2)`: The new price of the coffee. After the `RAISE_PRICE` stored procedure has been called, this parameter will contain the current price of the coffee `coffeeName`.

  When the example calls this stored procedure with `Colombian` as the value for `coffeeName`, `0.10` as the value for `maximumPercentage`, and `19.99` as the value for `newPrice`, the example produces output similar to the following:

```
Contents of COFFEES table before calling RAISE_PRICE:
Colombian, 101, 7.99, 0, 0
Colombian_Decaf, 101, 8.99, 0, 0
Espresso, 150, 9.99, 0, 0
French_Roast, 49, 8.99, 0, 0
French_Roast_Decaf, 49, 9.99, 0, 0

Calling the procedure RAISE_PRICE

Value of newPrice after calling RAISE_PRICE: 8.79

Contents of COFFEES table after calling RAISE_PRICE:
Colombian, 101, 8.79, 0, 0
Colombian_Decaf, 101, 8.99, 0, 0
Espresso, 150, 9.99, 0, 0
French_Roast, 49, 8.99, 0, 0
French_Roast_Decaf, 49, 9.99, 0, 0
```

Parameter Modes

The parameter attributes `IN` (the default), `OUT`, and `INOUT` are parameter modes. They define the action of formal parameters. The following table summarizes the information about parameter modes.

Creating Stored Procedures in Java DB

Note: See the section "CREATE PROCEDURE statement" in *Java DB Reference Manual* for more information about creating stored procedures in Java DB.
Creating and using a stored procedure in Java DB involves the following steps:

1. Create a public static Java method in a Java class: This method performs the required task of the stored procedure.
2. Create the stored procedure: This stored procedure calls the Java method you created.
3. Package the Java class (that contains the public static Java method you created earlier) in a JAR file.
4. Call the stored procedure with the `CALL` SQL statement. See the section Calling Stored Procedures in Java DB and MySQL.

Creating Public Static Java Method

The following method, `StoredProcedureJavaDBSample.showSuppliers`, contains the SQL statements that the stored procedure `SHOW_SUPPLIERS` calls:

```
public static void showSuppliers(ResultSet[] rs)
    throws SQLException {

    Connection con = DriverManager.getConnection("jdbc:default:connection");
    Statement stmt = null;

    String query =
        "select SUPPLIERS.SUP_NAME, " +
        "COFFEES.COF_NAME " +
        "from SUPPLIERS, COFFEES " +
        "where SUPPLIERS.SUP_ID = " +
        "COFFEES.SUP_ID " +
        "order by SUP_NAME";

    stmt = con.createStatement();
    rs[0] = stmt.executeQuery(query);
}
```

The `SHOW_SUPPLIERS` stored procedure takes no arguments. You can specify arguments in a stored procedure by defining them in the method signature of your public static Java method. Note that the method `showSuppliers` contains a parameter of type `ResultSet[]`. If your stored procedure returns any number of `ResultSet` objects, specify one parameter of type `ResultSet[]` in your Java method. In addition, ensure that this Java method is public and static.
Retrieve the `Connection` object from the URL `jdbc:default:connection`. This is a convention in Java DB to indicate that the stored procedure will use the currently existing `Connection` object.
Note that the `Statement` object is not closed in this method. Do not close any `Statement` objects in the Java method of your stored procedure; if you do so, the `ResultSet` object will not exist when you issue the `CALL` statement when you call your stored procedure.
In order for the stored procedure to return a generated result set, you must assign the result set to an array component of the `ResultSet[]` parameter. In this example, the generated result set is assigned to the array component `rs[0]`.
The following method is `StoredProcedureJavaDBSample.showSuppliers`:

```
public static void getSupplierOfCoffee(String coffeeName, String[] supplierName)
    throws SQLException {

    Connection con = DriverManager.getConnection("jdbc:default:connection");
    PreparedStatement pstmt = null;
    ResultSet rs = null;

    String query =
        "select SUPPLIERS.SUP_NAME " +
        "from SUPPLIERS, COFFEES " +
        "where " +
        "SUPPLIERS.SUP_ID = COFFEES.SUP_ID " +
        "and ? = COFFEES.COF_NAME";

    pstmt = con.prepareStatement(query);
    pstmt.setString(1, coffeeName);
    rs = pstmt.executeQuery();

    if (rs.next()) {
        supplierName[0] = rs.getString(1);
    } else {
        supplierName[0] = null;
    }
```

```
}
```

The formal parameter `coffeeName` has the parameter mode `IN`. This formal parameter is used like any other parameter in a Java method. Because the formal parameter `supplierName` has the parameter mode `OUT`, it must use a one dimensional array data type. Because this method does not produce a result set, the method definition does not contain a parameter of type `ResultSet[]`. In order to retrieve a value from an `OUT` formal parameter, you must assign the value to be retrieved to an array component of the `OUT` formal parameter. In this example, the retrieved name of the coffee supplier is assigned to the array component `supplierName[0]`.

The following is the method signature of the `StoredProcedureJavaDBSample.raisePrice` method:

```
public static void raisePrice(
   String coffeeName, double maximumPercentage,
   BigDecimal[] newPrice) throws SQLException
```

Because the formal parameter `newPrice` has the parameter mode `INOUT`, it must use a one dimensional array data type. Java DB maps the `FLOAT` and `NUMERIC` SQL data types to the `double` and `java.math.BigDecimal` Java data types, respectively.

Creating Stored Procedures in Java DB with SQL Scripts or JDBC API

Java DB uses the Java programming language for its stored procedures. Consequently, when you define a stored procedure, you specify which Java class to call and where Java DB can find it.

The following excerpt from `StoredProcedureJavaDBSample.createProcedures` creates a stored procedure named `SHOW_SUPPLIERS`:

```
public void createProcedures(Connection con)
    throws SQLException {

    Statement stmtCreateShowSuppliers = null;

    // ...

    String queryShowSuppliers =
        "CREATE PROCEDURE SHOW_SUPPLIERS() " +
        "PARAMETER STYLE JAVA " +
        "LANGUAGE JAVA " +
        "DYNAMIC RESULT SETS 1 " +
        "EXTERNAL NAME " +
        "'com.oracle.tutorial.jdbc." +
        "StoredProcedureJavaDBSample." +
        "showSuppliers'";

    // ...

    try {
        System.out.println("Calling CREATE PROCEDURE");
        stmtCreateShowSuppliers = con.createStatement();

        // ...

    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmtCreateShowSuppliers != null) {
            stmtCreateShowSuppliers.close();
        }
        // ...
    }
}
```

The following list describes the procedure elements you can specify in the `CREATE PROCEDURE` statement:

- `PARAMETER STYLE`: Identifies the convention used to pass parameters to the stored procedure. The following options are valid:
  - `JAVA`: Specifies that the stored procedure uses a parameter-passing convention that conforms to the Java language and the SQL routines specification.
  - `DERBY`: Specifies that the stored procedure supports a vararg as the final argument in the parameter list.
- `LANGUAGE JAVA`: Specifies the programming language of the stored procedure (currently, `JAVA` is the only option).
- `DYNAMIC RESULT SETS 1`: Specifies the maximum number of result sets retrieved; in this case, it is `1`.
- `EXTERNAL NAME`

`'com.oracle.tutorial.jdbc.StoredProcedureJavaDBSample.showSuppliers'` specifies the fully qualified Java method that this stored procedure calls. **Note**: Java DB must be able to find the method specified here in your class path or in a JAR file directly added to the database. See the following step, Package Java Class in JAR File.

The following statement (which is found in `StoredProcedureJavaDBSample.createProcedures`) creates a stored procedure named `GET_SUPPLIERS_OF_COFFEE` (line breaks have been added for clarity):

```
CREATE PROCEDURE GET_SUPPLIER_OF_COFFEE(
    IN coffeeName varchar(32),
    OUT supplierName
    varchar(40))
    PARAMETER STYLE JAVA
    LANGUAGE JAVA
    DYNAMIC RESULT SETS 0
    EXTERNAL NAME 'com.oracle.tutorial.jdbc.
        StoredProcedureJavaDBSample.
        getSupplierOfCoffee'
```

This stored procedure has two formal parameters, `coffeeName` and `supplierName`. The parameter specifiers `IN` and `OUT` are called parameter modes. They define the action of formal parameters. See Parameter Modes for more information. This stored procedure does not retrieve a result set, so the procedure element `DYNAMIC RESULT SETS` is `0`.

The following statement creates a stored procedure named `RAISE_PRICE` (line breaks have been added for clarity):

```
CREATE PROCEDURE RAISE_PRICE(
    IN coffeeName varchar(32),
    IN maximumPercentage float,
    INOUT newPrice float)
    PARAMETER STYLE JAVA
    LANGUAGE JAVA
    DYNAMIC RESULT SETS 0
    EXTERNAL NAME 'com.oracle.tutorial.jdbc.
        StoredProcedureJavaDBSample.raisePrice'
```

You can use SQL scripts to create stored procedures in Java DB. See the script `javadb/create-procedures.sql` and the Ant target `javadb-create-procedure` in the `build.xml` Ant build script.

Package Java Class in JAR File

The Ant build script `build.xml` contains targets to compile and package the tutorial in a JAR file. At a command prompt, change the current directory to *<JDBC tutorial directory>*. From this directory, run the following command to compile and package the tutorial in a JAR file:

```
ant jar
```

The name of the JAR file is *<JDBC tutorial directory>*`/lib/JDBCTutorial.jar`.

The Ant build script adds the file `JDBCTutorial.jar` to the class path. You can also specify the location of the JAR file in your `CLASSPATH` environment variable. This enables Java DB to find the Java method that the stored procedure calls.

Adding JAR File Directly to Database

Java DB looks first in your class path for any required classes, and then in the database. This section shows you how to add JAR files directly to the database.

Use the following system procedures to add the `JDBCTutorial.jar` JAR file to the database (line breaks have been added for clarity):

```
CALL sqlj.install_jar(
  '<JDBC tutorial directory>/
  lib/JDBCTutorial.jar',
  'APP.JDBCTutorial', 0)
CALL sqlj.replace_jar(
  '<JDBC tutorial directory>/
  lib/JDBCTutorial.jar',
  'APP.JDBCTutorial')";
CALL syscs_util.syscs_set_database_property(
  'derby.database.classpath',
  'APP.JDBCTutorial')";
```

Note: The method `StoredProcedureJavaDBSample.registerJarFile` demonstrates how to call these system procedures. If you call this method, ensure that you have modified `javadb-sample-properties.xml` so that the value of the property `jar_file` is set to the full path name of `JDBCTutorial.jar`.

The `install_jar` procedure in the `SQL` schema adds a JAR file to the database. The first argument of this procedure is the full path name of the JAR file on the computer from which this procedure is run. The second argument is an identifier that Java DB uses to refer to the JAR file. (The identifier `APP` is the Java DB default schema.) The `replace_jar` procedure replaces a JAR file already in the database.

The system procedure `SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` sets or deletes the value of a property of the database on the current connection. This method sets the property `derby.database.classpath` to the identifier specified in the `install_jar` file. Java DB first looks in your Java class path for a class, then it looks in `derby.database.classpath`.

Creating Stored Procedure in MySQL

Creating and using a stored procedure in Java DB involves the following steps:

1. Create the stored procedure with an SQL script or JDBC API
2. Call the stored procedure with the `CALL` SQL statement. See the section Calling Stored Procedures in Java DB and MySQL

Creating Stored Procedure in MySQL with SQL Scripts or JDBC API

MySQL uses a SQL-based syntax for its stored procedures. The following excerpt from the SQL script `mysql/create-procedures.sql` creates a stored procedure named `SHOW_SUPPLIERS`:

```
SELECT 'Dropping procedure SHOW_SUPPLIERS' AS ' '|
drop procedure if exists SHOW_SUPPLIERS|

# ...

SELECT 'Creating procedure SHOW_SUPPLIERS' AS ' '|
create procedure SHOW_SUPPLIERS()
    begin
        select SUPPLIERS.SUP_NAME,
        COFFEES.COF_NAME
        from SUPPLIERS, COFFEES
        where SUPPLIERS.SUP_ID = COFFEES.SUP_ID
        order by SUP_NAME;
    end|
```

The `DROP PROCEDURE` statement deletes that procedure `SHOW_SUPPLIERS` if it exists. In MySQL, statements in a stored procedure are separated by semicolons. However, a different delimiter is required to end the `create procedure` statement. This example uses the pipe (`|`) character; you can use another character (or more than one character). This character that separates statements is defined in the `delimiter` attribute in the Ant target that calls this script. This excerpt is from the Ant build file `build.xml` (line breaks have been inserted for clarity):

```
<target name="mysql-create-procedure">

  <sql driver="${DB.DRIVER}"
       url="${DB.URL}" userid="${DB.USER}"
       password="${DB.PASSWORD}"
       classpathref="CLASSPATH"
       print="true"
       delimiter="|"
       autocommit="false"
       onerror="abort">
       <transaction
         src="./sql/${DB.VENDOR}/
           create-procedures.sql">
       </transaction>
  </sql>

</target>
```

Alternatively, you can use the `DELIMITER` SQL statement to specify a different delimiter character.

The `CREATE PROCEDURE` statement consists of the name of the procedure, a comma-separated list of parameters in parentheses, and SQL statements within the `BEGIN` and `END` keywords.

You can use the JDBC API to create a stored procedure. The following method, `StoredProcedureMySQLSample.createProcedureShowSuppliers`, performs the same tasks as the previous script:

```
public void
    createProcedureShowSuppliers()
    throws SQLException {
    String createProcedure = null;

    String queryDrop =
        "DROP PROCEDURE IF EXISTS SHOW_SUPPLIERS";

    createProcedure =
        "create procedure SHOW_SUPPLIERS() " +
        "begin " +
            "select SUPPLIERS.SUP_NAME, " +
            "COFFEES.COF_NAME " +
            "from SUPPLIERS, COFFEES " +
            "where SUPPLIERS.SUP_ID = " +
            "COFFEES.SUP_ID " +
            "order by SUP_NAME; " +
        "end";
    Statement stmt = null;
    Statement stmtDrop = null;

    try {
        System.out.println("Calling DROP PROCEDURE");
        stmtDrop = con.createStatement();
        stmtDrop.execute(queryDrop);
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
```

```
        if (stmtDrop != null)
        {
            stmtDrop.close();
        }
    }

    try {
        stmt = con.createStatement();
        stmt.executeUpdate(createProcedure);
    } catch (SQLException e) {
        JDBCTutorialUtilities.printSQLException(e);
    } finally {
        if (stmt != null) { stmt.close(); }
    }
}
```

Note that the delimiter has not been changed in this method.

The stored procedure `SHOW_SUPPLIERS` generates a result set, even though the return type of the method `createProcedureShowSuppliers` is `void` and the method does not contain any parameters. A result set is returned when the stored procedure `SHOW_SUPPLIERS` is called with the method `CallableStatement.executeQuery`:

```
CallableStatement cs = null;
cs = this.con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

The following excerpt from the method `StoredProcedureMySQLSample.createProcedureGetSupplierOfCoffee` contains the SQL query that creates a stored procedure named `GET_SUPPLIER_OF_COFFEE`:

```
public void createProcedureGetSupplierOfCoffee()
    throws SQLException {

    String createProcedure = null;

    // ...

    createProcedure =
        "create procedure GET_SUPPLIER_OF_COFFEE(" +
        "IN coffeeName varchar(32), " +
        "OUT supplierName varchar(40)) " +
        "begin " +
            "select SUPPLIERS.SUP_NAME into " +
            "supplierName " +
            "from SUPPLIERS, COFFEES " +
            "where SUPPLIERS.SUP_ID = " +
            "COFFEES.SUP_ID " +
            "and coffeeName = COFFEES.COF_NAME; " +
            "select supplierName; " +
        "end";
    // ...
}
```

This stored procedure has two formal parameters, `coffeeName` and `supplierName`. The parameter specifiers `IN` and `OUT` are called parameter modes. They define the action of formal parameters. See Parameter Modes for more information. The formal parameters are defined in the SQL query, not in the method `createProcedureGetSupplierOfCoffee`. To assign a value to the `OUT` parameter `supplierName`, this stored procedure uses a `SELECT` statement.

The following excerpt from the method `StoredProcedureMySQLSample.createProcedureRaisePrice` contains the SQL query that creates a stored procedure named `RAISE_PRICE`:

```
public void createProcedureRaisePrice()
    throws SQLException {

    String createProcedure = null;

    // ...

    createProcedure =
        "create procedure RAISE_PRICE(" +
        "IN coffeeName varchar(32), " +
        "IN maximumPercentage float, " +
        "INOUT newPrice numeric(10,2)) " +
        "begin " +
        "main: BEGIN " +
            "declare maximumNewPrice " +
                "numeric(10,2); " +
            "declare oldPrice numeric(10,2); " +
            "select COFFEES.PRICE into oldPrice " +
                "from COFFEES " +
                "where COFFEES.COF_NAME " +
                "= coffeeName; " +
            "set maximumNewPrice = " +
                "oldPrice * (1 + " +
                "maximumPercentage); " +
            "if (newPrice > maximumNewPrice) " +
                "then set newPrice = " +
                "maximumNewPrice; " +
            "end if; " +
            "if (newPrice <= oldPrice) " +
                "then set newPrice = oldPrice; " +
                "leave main; " +
            "end if; " +
            "update COFFEES " +
                "set COFFEES.PRICE = newPrice " +
                "where COFFEES.COF_NAME " +
                "= coffeeName; " +
            "select newPrice; " +
```

```
        "END main; " +
        "end";

    // ...
}
```

The stored procedure assigns a value to the `INOUT` parameter `newPrice` with the `SET` and `SELECT` statements. To exit the stored procedure, the stored procedure first encloses the statements in a `BEGIN ... END` block labeled `main`. To exit the procedure, the method uses the statement `leave main`.

Calling Stored Procedures in Java DB and MySQL

The following excerpt from method `runStoredProcedures`, calls the stored procedure `SHOW_SUPPLIERS` and prints the generated result set:

```
cs = this.con.prepareCall("{call SHOW_SUPPLIERS()}");
ResultSet rs = cs.executeQuery();

while (rs.next()) {
    String supplier = rs.getString("SUP_NAME");
    String coffee = rs.getString("COF_NAME");
    System.out.println(supplier + ": " + coffee);
}
```

Note: As with `Statement` objects, to call the stored procedure, you can call `execute`, `executeQuery`, or `executeUpdate` depending on how many `ResultSet` objects the procedure returns. However, if you are not sure how many `ResultSet` objects the procedure returns, call `execute`.

Calling the stored procedure `SHOW_SUPPLIERS` is demonstrated in the section Creating Stored Procedure with JDBC API in MySQL.

The following excerpt from method `runStoredProcedures`, calls the stored procedure `GET_SUPPLIER_OF_COFFEE`:

```
cs = this.con.prepareCall("{call GET_SUPPLIER_OF_COFFEE(?, ?)}");
cs.setString(1, coffeeNameArg);
cs.registerOutParameter(2, Types.VARCHAR);
cs.executeQuery();

String supplierName = cs.getString(2);
```

The interface `CallableStatement` extends `PreparedStatement`. It is used to call stored procedures. Specify values for `IN` parameters (such as `coffeeName` in this example) just like you would with a `PreparedStatement` object by calling the appropriate setter method. However, if a stored procedure contains an `OUT` parameter, you must register it with the `registerOutParameter` method.

The following excerpt from the method `runStoredProcedures`, calls the stored procedure `RAISE_PRICE`:

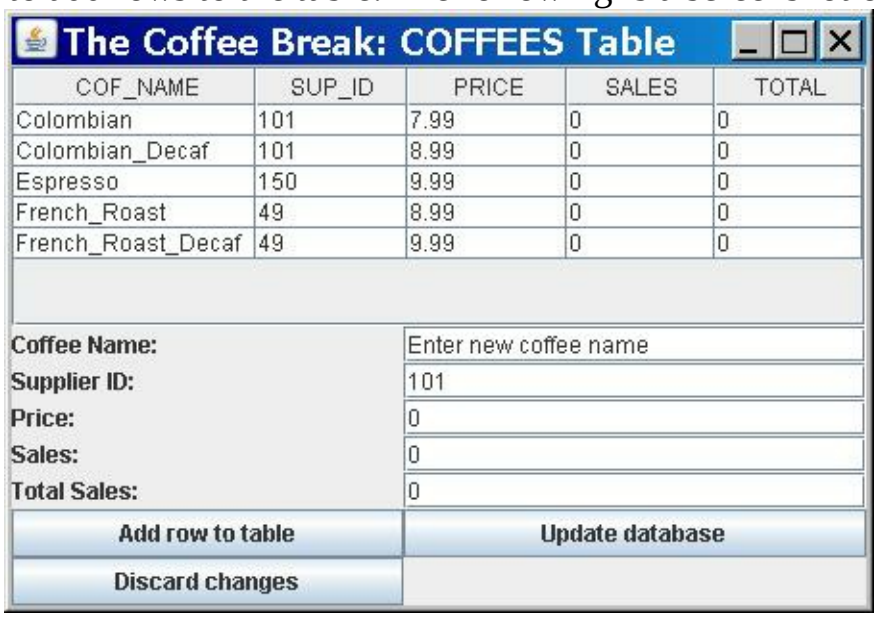
```
cs = this.con.prepareCall("{call RAISE_PRICE(?,?,?)}");
cs.setString(1, coffeeNameArg);
cs.setFloat(2, maximumPercentageArg);
cs.registerOutParameter(3, Types.NUMERIC);
cs.setFloat(3, newPriceArg);

cs.execute();
```

Because the parameter `newPrice` (the third parameter in the procedure `RAISE_PRICE`) has the parameter mode `INOUT`, you must both specify its value by calling the appropriate setter method and register it with the `registerOutParameter` method.

Using JDBC with GUI API

The sample `CoffeesFrame.java` demonstrates how to integrate JDBC with a GUI API, in particular, the Swing API. It displays the contents of the `COFFEES` database table in a table and contains fields and buttons that enable you to add rows to the table. The following is a screenshot of this sample:



The sample contains five text fields that correspond to each of the columns in the `COFFEES` table. It also contains three buttons:

- **Add row to table**: Adds a row to the sample's table based on the data entered in the text fields.
- **Update database**: Updates the table `COFFEES` based on the data in the sample's table.
- **Discard changes**: Retrieves the contents of the `COFFEES` table, replacing the existing data in the sample's table.

This sample (which requires `CoffeesTableModel`) demonstrates the following general steps to integrate JDBC with the Swing API:

1. Implementing the `TableModel` interface
2. Implementing the `RowSetListener` interface
3. Laying out the Swing components
4. Adding listeners for the buttons in the sample

Implementing javax.swing.event.TableModel

The `TableModel` interface enables a Java Swing application to manage data in a `JTable` object. The sample, `CoffeesTableModel.java`, implements this interface. It specifies how a `JTable` object should retrieve data from a `RowSet` object and display it in a table.

Note: Although this sample displays the contents of the `COFFEES` table in a Swing application, the class `CoffeesTableModel` should work for any SQL table provided that its data can be represented with `String` objects. (However, the fields that enable users to add rows to `COFFEES`, which are specified in the class `CoffeesFrame`, would have to be modified for other SQL tables.)

Before implementing the methods of the interface `TableModel`, the constructor of the class `CoffeeTableModel` initializes various member variables required for these implemented methods as follows:

```
public CoffeesTableModel(CachedRowSet rowSetArg)
    throws SQLException {

    this.coffeesRowSet = rowSetArg;
    this.metadata = this.coffeesRowSet.getMetaData();
    numcols = metadata.getColumnCount();

    // Retrieve the number of rows.
    this.coffeesRowSet.beforeFirst();
    this.numrows = 0;
    while (this.coffeesRowSet.next()) {
        this.numrows++;
    }
    this.coffeesRowSet.beforeFirst();
}
```

The following describes the member variables initialized in this constructor:

- `CachedRowSet coffeesRowSet`: Stores the contents of the table `COFFEES`.
  This sample uses a `RowSet` object, in particular, a `CachedRowSet` object, rather than a `ResultSet` object for two reasons. A `CachedRowSet` object enables the user of the application to make changes to the data contained in it without being connected to the database. In addition, because a `CachedRowSet` object is a JavaBeans component, it can notify other components when certain things happen to it. In this sample, when a new row is added to the `CachedRowSet` object, it notifies the Swing component that is rendering its data in a table to refresh itself and display the new row.
- `ResultSetMetaData metadata`: Retrieves the number of columns in the table `COFFEES` as well as the names of each of them.
- `int numcols, numrows`: Stores the number of columns and rows, respectively, in the table `COFFEES`.

The `CoffeesTableModel.java` sample implements the following methods from `TableModel` interface:

- `Class<?> getColumnClass(int columnIndex)`: Returns the most specific superclass for all the cell values in the column.
- `int getColumnCount()`: Returns the number of columns in the model.
- `String getColumnName(int columnIndex)`: Returns the name of the column specified by the parameter `columnIndex`.
- `int getRowCount()`: Returns the number of rows in the model.
- `Object getValueAt(int rowIndex, int columnIndex)`: Returns the value for the cell at intersection of the column `columnIndex` and the row `rowIndex`.
- `boolean isCellEditable(int rowIndex, int columnIndex)`: Returns true if the cell at the intersection of the column `rowIndex` and the row `columnIndex` can be edited.

The following methods have not been implemented because this sample does not allow users to directly edit the contents of the table:

- `void addTableModelListener(TableModelListener l)`: Adds a listener to the list that is notified each time a change to the data model occurs.
- `void removeTableModelListener(TableModelListener l)`: Removes a listener from the list that is notified each time a change to the data model occurs.
- `void setValueAt(Object aValue, int rowIndex, int columnIndex)`: Sets the value in the cell at the intersection of the column `columnIndex` and the row `rowIndex` to the object `aValue`.

Implementing getColumnCount and getRowCount

The methods `getColumnCount` and `getRowCount` return the value of the member variables `numcols` and `numrows`, respectively:

```
public int getColumnCount() {
    return numcols;
}

public int getRowCount() {
    return numrows;
}
```

Implementing getColumnClass

The `getColumnClass` method returns the data type of the specified column. To keep things simple, this method returns the `String` class, thereby converting all data in the table into `String` objects. The `JTable` class uses this method to determine how to render data in the GUI application.

```
public Class getColumnClass(int column) {
    return String.class;
}
```

Implementing getColumnName

The `getColumnName` method returns the name of the specified column. The `JTable` class uses this method to label each of its columns.

```
public String getColumnName(int column) {
    try {
        return this.metadata.getColumnLabel(column + 1);
    } catch (SQLException e) {
        return e.toString();
    }
}
```

Implementing getColumnAt

The `getColumnAt` method retrieves the value at the specified row and column in the row set `coffeesRowSet`. The `JTable` class uses this method to populate its table. Note that SQL starts numbering its rows and columns at 1, but the `TableModel` interface starts at 0; this is the reason why the `rowIndex` and `columnIndex` values are incremented by 1.

```
public Object getValueAt(int rowIndex, int columnIndex) {

    try {
        this.coffeesRowSet.absolute(rowIndex + 1);
        Object o = this.coffeesRowSet.getObject(columnIndex + 1);
        if (o == null)
            return null;
        else
            return o.toString();
    } catch (SQLException e) {
        return e.toString();
    }
}
```

Implementing isCellEditable

Because this sample does not allow users to directly edit the contents of the table (rows are added by another window control), this method returns `false` regardless of the values of `rowIndex` and `columnIndex`:

```
public boolean isCellEditable(int rowIndex, int columnIndex) {
    return false;
}
```

Implementing javax.sql.RowSetListener

The class `CoffeesFrame` implements only one method from the interface `RowSetListener`, `rowChanged`. This method is called when a user adds a row to the table.

```
public void rowChanged(RowSetEvent event) {

    CachedRowSet currentRowSet =
        this.myCoffeesTableModel.coffeesRowSet;

    try {
        currentRowSet.moveToCurrentRow();
        myCoffeesTableModel = new CoffeesTableModel(
            myCoffeesTableModel.getCoffeesRowSet());
        table.setModel(myCoffeesTableModel);

    } catch (SQLException ex) {

        JDBCTutorialUtilities.printSQLException(ex);

        // Display the error in a dialog box.

        JOptionPane.showMessageDialog(
            CoffeesFrame.this,
            new String[] {
                // Display a 2-line message
                ex.getClass().getName() + ": ",
                ex.getMessage()
            }
        );
    }
}
```

This method updates the table in the GUI application.

Laying Out Swing Components

The constructor of the class `CoffeesFrame` initializes and lays out the Swing components. The following statement retrieves the contents of the `COFFEES` table, stores the contents in the `CachedRowSet` object `myCachedRowSet`, and initializes the `JTable` Swing component:

```
CachedRowSet myCachedRowSet = getContentsOfCoffeesTable();
myCoffeesTableModel = new CoffeesTableModel(myCachedRowSet);
myCoffeesTableModel.addEventHandlersToRowSet(this);

// Displays the table
table = new JTable();
table.setModel(myCoffeesTableModel);
```

As mentioned previously, instead of a `ResultSet` object to represent the contents of the `COFFEES` table, this sample uses a `RowSet` object, notably a `CachedRowSet` object.

The method `CoffeesFrame.getContentsOfCoffeesTable` retrieves the contents of the table `COFFEES`.

The method `CoffeesTableModel.addEventHandlersToRowSet` adds the event handler defined in the `CoffeesFrame` class, which is the method `rowChanged`, to the row set member variable `CoffeesTableModel.coffeesRowSet`. This enables the class `CoffeesFrame` to notify the row set `coffeesRowSet` of any events, in particular, when a user clicks the button **Add row to table**, **Update database**, or **Discard changes**. When the row set `coffeesRowSet` is notified of one of these changes, the method `CoffeesFrame.rowChanged` is called.

The statement `table.setModel(myCoffeesTableModel)` specifies that it use the `CoffeesTableModel` object `myCoffeesTableModel` to populate the `JTable` Swing component `table`.

The following statements specify that the `CoffeesFrame` class use the layout `GridBagLayout` to lay out its Swing components:

```
Container contentPane = getContentPane();
contentPane.setComponentOrientation(
    ComponentOrientation.LEFT_TO_RIGHT);
contentPane.setLayout(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
```

See How to Use GridBagLayout in the Creating a GUI With JFC/Swing for more information about using the layout `GridBagLayout`.

See the source code for `CoffeesFrame.java` to see how the Swing components of this sample are added to the layout `GridBagLayout`.

Adding Listeners for Buttons

The following statement adds a listener to the button **Add row to table**:

```
button_ADD_ROW.addActionListener(
    new ActionListener() {

    public void actionPerformed(ActionEvent e) {

        JOptionPane.showMessageDialog(
            CoffeesFrame.this, new String[] {
                "Adding the following row:",
                "Coffee name: [" +
                textField_COF_NAME.getText() +
                "]",
                "Supplier ID: [" +
                textField_SUP_ID.getText() + "]",
                "Price: [" +
                textField_PRICE.getText() + "]",
                "Sales: [" +
                textField_SALES.getText() + "]",
                "Total: [" +
                textField_TOTAL.getText() + "]"
            }
        );

        try {
            myCoffeesTableModel.insertRow(
                textField_COF_NAME.getText(),
                Integer.parseInt(textField_SUP_ID.getText().trim()),
                Float.parseFloat(textField_PRICE.getText().trim()),
                Integer.parseInt(textField_SALES.getText().trim()),
                Integer.parseInt(textField_TOTAL.getText().trim())
            );
        } catch (SQLException sqle) {
            displaySQLExceptionDialog(sqle);
        }
    }
});
```

When a user clicks this button, it performs the following:

- Creates a message dialog box that displays the row to be added to the table.
- Calls the method `CoffeesTableModel.insertRow`, which adds the row to the member variable `CoffeesTableModel.coffeesRowSet`.

If an `SQLException` is thrown, then the method `CoffeesFrame.displaySQLExceptionDialog` creates a message dialog box that displays the content of the `SQLException`.
The following statement adds a listener to the button **Update database**:

```
button_UPDATE_DATABASE.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                myCoffeesTableModel.coffeesRowSet.acceptChanges();
                msgline.setText("Updated database");
            } catch (SQLException sqle) {
                displaySQLExceptionDialog(sqle);
                // Now revert back changes
                try {
                    createNewTableModel();
                    msgline.setText("Discarded changes");
                } catch (SQLException sqle2) {
                    displaySQLExceptionDialog(sqle2);
                }
            }
        }
    }
);
```

When a user clicks this button, the table `COFFEES` is updated with the contents of the row set `myCoffeesTableModel.coffeesRowSet`.
The following statement adds a listener to the button **Discard changes**:

```
button_DISCARD_CHANGES.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        try {
            createNewTableModel();
        } catch (SQLException sqle) {
            displaySQLExceptionDialog(sqle);
        }
    }
});
```

When a user clicks this button, the method `CoffeesFrame.createNewTableModel` is called, which repopulates the `JTable` component with the contents of the `COFFEES` table.

JDBC(TM) Database Access: End of Trail

You have reached the end of the "JDBC(TM) Database Access" trail.
If you have comments or suggestions about this trail, use our feedback page to tell us about it.

 Creating a GUI With JFC/Swing: Using the JDBC to connect and interact with a database is generally considered the back-end of an application. You can use Swing to put a user interface on the front-end of your database application.