Unit 3- Database Connectivity

A **database** is an organized collection of data. There are many different strategies for organizing data to facilitate easy access and manipulation. A **database management system(DBMS)** provides mechanisms for **storing, organizing, retrieving and modifying data** form any users. **Database management systems allow for the access and storage of data without concern for the internal representation of data.**

Today's most popular database systems are **relational databases**.A language called **SQL**—pronounced "sequel," or as its individual letters—is the international standard language used almost universally with relational databases to perform **queries** (i.e., to request information that satisfies given criteria) and to manipulate data.

Some popular **relational database management systems (RDBMSs)** are **Microsoft SQL Server, Oracle, Sybase, IBM DB2, Informix, PostgreSQL and MySQL.** The JDK now comes with a pure-Java RDBMS called **Java DB**—Oracles's version of Apache Derby.

Java programs communicate with databases and manipulate their data using the **Java Database Connectivity (JDBC™) API**. A **JDBC driver** enables Java applications to connect to a database in a particular DBMS and allows you to manipulate that database using the JDBC API.

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 to write Java applications that manage these three programming activities:

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

JDBC includes four components:

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.

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.

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.

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.

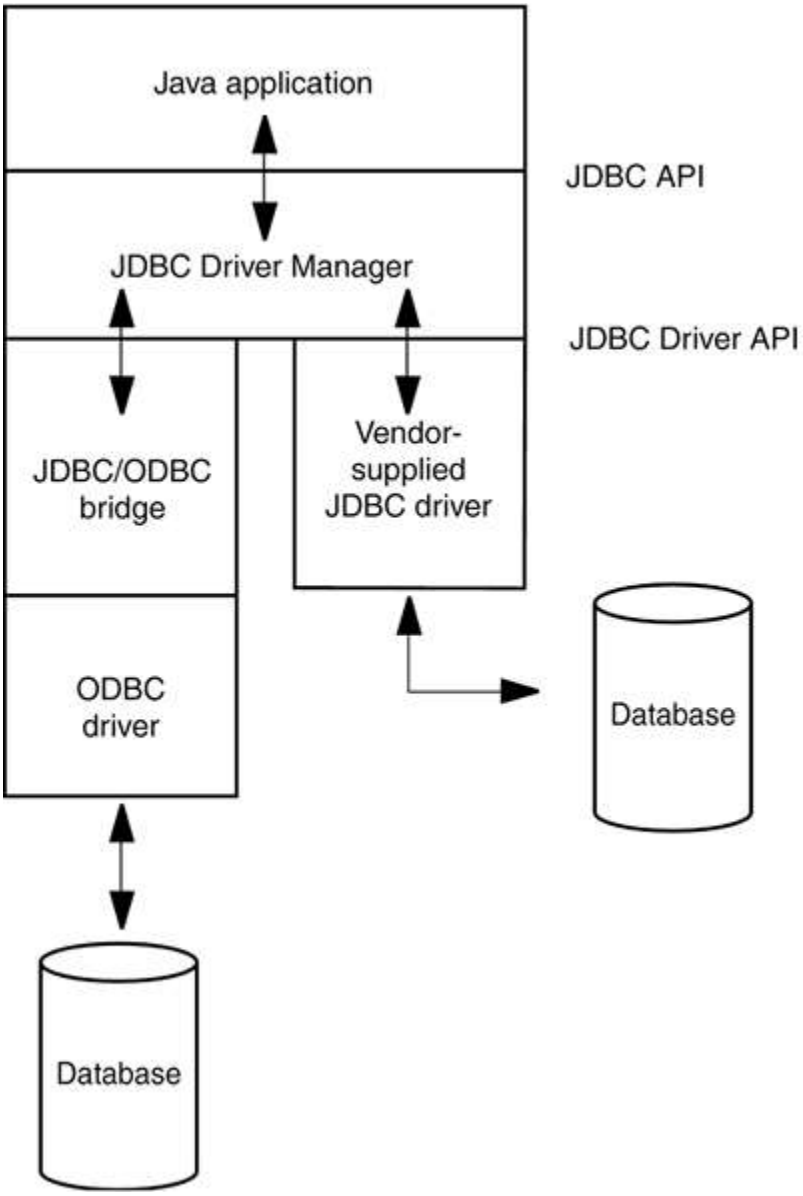

fig.JDBC-to-database communication path

JDBC Driver Types

JDBC drivers are classified into the following types:

- A **type 1 driver** translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Sun includes one such driver, the JDBC/ODBC bridge, with the JDK. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, plenty of better drivers are available, and is advised against using the JDBC/ODBC bridge.

- A **type 2 driver** is written partly in Java and partly in native code; it communicates with the client API of a database. When you use such a driver, you must install some platform-specific code in addition to a Java library.

- A **type 3 driver** is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This can simplify deployment since the database-dependent code is located only on the server.

- A **type 4 driver** is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers that are provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:
Programmers can write applications in the Java programming language to access any database, using standard SQL statementsor even specialized extensions of SQLwhile still following Java language conventions.
Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

Typical Uses of JDBC

The traditional **client/server** model has a rich **GUI on the client** and **a database on the server (**figure below). In this model, a JDBC driver is deployed on the client.

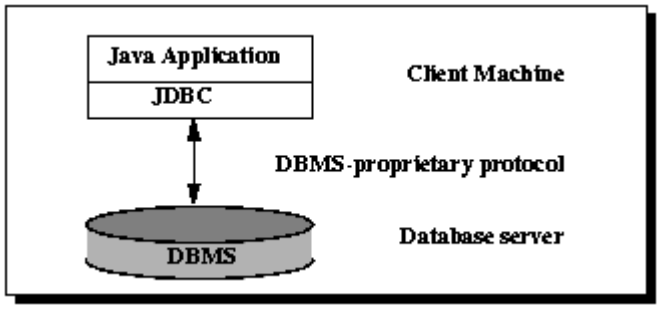

fig.Two-tier Architecture for Data Access.

However, the world is moving away from client/server and toward a **"three-tier model"** or even more advanced "n-tier models." In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates **visual presentation (on the client)** from the **business logic (in the middle tier)** and the **raw data (in the database).** Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when you use a web browser as the client), RMI (when you use an application), or another mechanism. JDBC manages the communication between the middle tier and the back-end database. Figure below shows the basic three tier architecture.

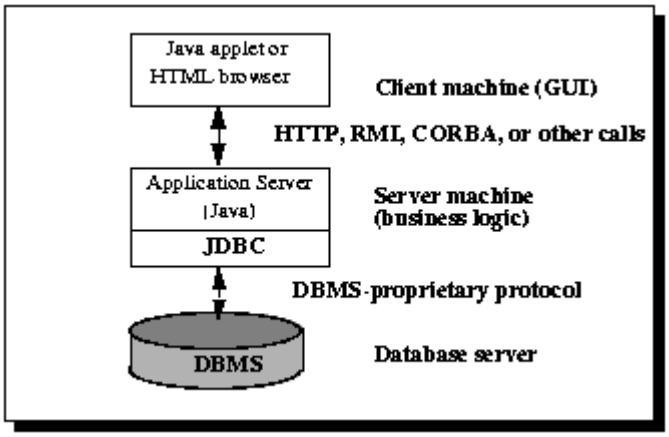

fig.Three-tier Architecture for Data Access

Database connections in Java using JDBC and Netbeans IDE

In order to make a JDBC connection to MySQL database one needs to downlad the **MySQL Connector/J**. It is also expected that you have Netbeans 7+ installed on your machine.

Extract the zip file to a folder, you'll see file '**mysql-connector-java-5.1.20-bin.jar**' which is the library file that we want. Just copy the file to the library folder, for example to **"C:\Program Files\Java\jdk1.7\lib" also to the "C:\Program Files\Java\jdk1.7\jre\lib directory.**

Next, create a **new Java project** on NetBeans named 'DatabaseConnectivity'.
Right click the project and select '**properties**' then under the '**categories'** click on **'libraries'** and click on the '**Add JAR/Folder'** and then browse to **"C:\Program Files\Java\jdk1.7\lib\mysql-connector-java-5.1.20-bin.jar" ,** click on **'open** ' and **'ok'.**

In the services tab of the netbeans , right click the **database** and click on '**new connection'**. Under the **'new connection wizard'**, click on the **'Driver Combobox'** and select the **'mySQL Connector/J driver'** and click on **'next'** untill the wizard completes.

Instructions for Setting Up a MySQL User Account

For the MySQL examples to execute correctly, you need to set up a user account that allows
users to create, delete and modify a database. After MySQL is installed, follow the (these steps assume MySQL is installed in its default installation directory):

1. Open a Command Prompt and start the database server by executing the command **mysqld.exe**. This command has no output—it simply starts the MySQL server. Do not close this window—doing so terminates the server.
2. Next, you'll start theMySQL monitor so you can set up a user account, open another Command Prompt and execute the command
mysql -h localhost -u root
The -h option indicates the host (i.e., computer) on which the MySQL server is running—in this case your local computer (localhost). The -u option indicates the user account that will be used to log in to the server—root is the default user account that is created during installation to allow you to configure the server. Once you've logged in, you'll see a mysql> prompt at which you can type commands to interact with the MySQL server.
3.At the mysql> prompt, type
USE mysql;
and press Enter to select the built-in database named mysql, which stores server information, such as user accounts and their privileges for interacting with the server. Each command must end with a semicolon. To confirm the command, MySQL issues the message "Database changed."
4. Next, you'll add the user account to the mysql built-in database. The mysql database contains a table called user with columns that represent the user's name, password and various privileges. To create the user account '**abc**'with the password **'pqr'** , execute the following commands from the mysql> prompt:
create user 'abc'@'localhost' identified by 'pqr';
grant select, insert, update, delete, create, drop, references,
execute on *.* to 'abc'@'localhost';
5.Type the command
exit;
to terminate the MySQL monitor.

Connecting to and Querying a Database

The example below performs a simple query on the **books database** that retrieves the entire Authors table and displays the data. The program illustrates **connecting to the database, querying the database and processing the result**.The discussion that follows presents the key JDBC aspects of the program.

```
// DisplayAuthors.java
// Displaying the contents of the Authors table.
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;

public class DisplayAuthors
 {
// database URL
static final String DATABASE_URL = "jdbc:mysql://localhost/books";
// launch the application
public static void main( String args[] )
{
Connection connection = null; // manages connection
```

```
Statement statement = null; // query statement
ResultSet resultSet = null; // manages results

 // connect to database books and query database
 try
 {
// establish connection to database
connection = DriverManager.getConnection(
DATABASE_URL, "root", "" );
// create Statement for querying database
statement = connection.createStatement();
// query database
resultSet = statement.executeQuery(
"SELECT AuthorID, FirstName, LastName FROM Authors" );
// process query results
ResultSetMetaData metaData = resultSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();
System.out.println( "Authors Table of Books Database:\n" );

for ( int i = 1; i <= numberOfColumns; i++ )
System.out.printf( "%-8s\t",metaData.getColumnName( i ));
System.out.println();

while(resultSet.next())
{
for ( int i = 1; i <= numberOfColumns; i++ )
System.out.printf( "%-8s\t",resultSet.getObject( i ));
System.out.println();
} // end while
} // end try
catch(SQLException sqlException)
{
sqlException.printStackTrace();
} // end catch
finally // ensure resultSet, statement and connection are closed
{
try
{
resultSet.close();
statement.close();
connection.close();
} // end try
catch ( Exception exception )
{
exception.printStackTrace();
} // end catch
} // end finally
} // end main
```

} // end class DisplayAuthors

output

```
Authors Table of Books Database:

AuthorID        FirstName       LastName
1               Harvey          Deitel
2               Paul            Deitel
3               Andrew          Goldberg
4               David           Choffnes
```

Connecting to the Database

An object that implements **interface Connection** manages the connection between the Java program and the database. Connection objects enable programs to create SQL statements that manipulate databases. The program initializes connection with the result of a call to **static method getConnection of class DriverManager (package java.sql),** which attempts to connect to the database specified by its URL.**Method get-Connection takes three arguments—a String that specifies the database URL**, **a String that specifies the username and a String that specifies the password**. **The URL locates the database** (possibly on a network or in the local file system of the computer). The URL jdbc:mysql://localhost/books specifies the **protocol for communication (jdbc), the subprotocol for communication (mysql)** and **the location of the database (//localhost/books, where localhost is the host running the MySQL server and books is the database name).** The subprotocol mysql indicates that the program uses a MySQL-specific subprotocol to connect to the MySQL database. If the DriverManager cannot connect to the database, method getConnection throws a **SQLException (package java.sql**).

Figure below lists the JDBC driver names and database URL formats of several popular RDBMSs.

RDBMS	Database URL format
MySQL	`jdbc:mysql://`*hostname*`:`*portNumber*`/`*databaseName*
ORACLE	`jdbc:oracle:thin:@`*hostname*`:`*portNumber*`:`*databaseName*
DB2	`jdbc:db2:`*hostname*`:`*portNumber*`/`*databaseName*
PostgreSQL	`jdbc:postgresql://`*hostname*`:`*portNumber*`/`*databaseName*
Java DB/Apache Derby	`jdbc:derby:`*dataBaseName* (embedded) `jdbc:derby://`*hostname*`:`*portNumber*`/`*databaseName* (network)
Microsoft SQL Server	`jdbc:sqlserver://`*hostname*`:`*portNumber*`;databaseName=`*dataBaseName*
Sybase	`jdbc:sybase:Tds:`*hostname*`:`*portNumber*`/`*databaseName*

Creating a Statement for Executing Queries

Connection method **createStatement** is invoked to obtain an object that implements **interface Statement** (package java.sql). The program uses the Statement object to submit SQL statements to the database.

Executing a Query

The **Statement** object's **executeQuery** method is used to submit a query that selects all the author information from table Authors. This method returns an object that implements interface **ResultSet** and contains the query results. The ResultSet methods enable the program to manipulate the query result.

Processing a Query's ResultSet

The **metadata** describes the ResultSet's contents. Programs can use metadata programmatically to obtain information about the **ResultSet's column names and types**. **ResultSetMetaData** method

getColumnCount is used to retrieve the number of columns in the ResultSet.

Retrieving and Modifying Values from Result Sets

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. 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.

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:

(a)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.

(b)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.

(c)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.

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.

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.

```
try
{

  // create Statement for querying database
  statement = connection.createStatement();

  // query database
  resultSet = statement.executeQuery(
    "SELECT AuthorID, FirstName, LastName FROM authors" );

  // process query results
  ResultSetMetaData metaData = resultSet.getMetaData();
  int numberOfColumns = metaData.getColumnCount();
  System.out.println( "Authors Table of Books Database:\n" );

  for ( int i = 1; i <= numberOfColumns; i++ )
    System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
  System.out.println();

  while ( resultSet.next() )
```

```
    {
        int id = rs.getInt("AuthorID");
        String firstName = rs.getString("FirstName");
        String lastName = rs.getString("LastName");
        System.out.println(id+ "\t" + firstName+
                "\t" + lastName );
    } // end while
  } // end try
  catch ( SQLException sqlException )
  {
    sqlException.printStackTrace();
  } // end catch
```

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.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.

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.

```
try {
    // establish connection to database
     connection = DriverManager.getConnection(
      DATABASE_URL, "root", "" );
    statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
          ResultSet.CONCUR_UPDATABLE);
    ResultSet uprs= statement.executeQuery(
```

```
        "SELECT * FROM authors");

      while (uprs.next()) {
        uprs.updateString( "LastName","Sharma");
        uprs.updateRow();
      }
      }
    catch ( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch
```

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.updateString** updates the specified column (in this example, LastName 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**.

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.

```
try {
      statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE);

      ResultSet uprs = statement.executeQuery(
        "SELECT * FROM authors");

      uprs.moveToInsertRow();
      uprs.updateInt("AuthorID",9);
      uprs.updateString("FirstName","Subash");
      uprs.updateString("LastName","Pakhrin");
      uprs.insertRow();
      uprs.beforeFirst();
   }
      catch ( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch
```

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 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.

```
try {
   connection = DriverManager.getConnection(
     DATABASE_URL, "root", "" );
 connection.setAutoCommit(false);
 statement = connection.createStatement();

 statement.addBatch(
    "INSERT INTO authors " +
    "VALUES('15','Hari','Shrestha')");

 statement.addBatch(
    "INSERT INTO authors " +
    "VALUES('16','Ram','Acharya')");

 statement.addBatch(
    "INSERT INTO authors  " +
```

```
        "VALUES('17','Shyam','Gautam')");

    statement.addBatch(
        "INSERT INTO authors  " +
        "VALUES('18','Govinda','Paudel')");

     int [] updateCounts = statement.executeBatch();
    connection.commit();

  } catch(BatchUpdateException b) {
     b.printStackTrace();
  } catch(SQLException ex) {
     ex.printStackTrace();
  }
```

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.
connection.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 statement. In this example, these commands are all INSERT INTO statements, each one adding a row consisting of three column values.

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 = statement.executeBatch();

Note that **statement** 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 "Hari" , then add the row for "Ram", then "Shyam" , and finally "Govinda". 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 "authors" 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.

connection.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.

PreparedStatements

A **PreparedStatement** enables you to create **compiled SQL statements** that execute more efficiently than Statements. PreparedStatements can also specify parameters, making them more flexible than Statements—you can execute the same query repeatedly with different
parameter values.

The **PreparedStatement** is derived from the more general class, **Statement**. If you want to execute a Statement object many times, it usually reduces execution time to use a PreparedStatement object instead.

Performing Parameterized Batch Update using PreparedStatement

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

```
try {
        connection = DriverManager.getConnection(
          DATABASE_URL, "root", "" );
     connection.setAutoCommit(false);
PreparedStatement pstmt = connection.prepareStatement(
                    "INSERT INTO authors VALUES(?, ?, ?)");
pstmt.setInt(1,19);
pstmt.setString(2, "Navin");
pstmt.setString(3,"Sharma");
pstmt.addBatch();

pstmt.setInt(1,20);
pstmt.setString(2, "Rajesh");
pstmt.setString(3,"Paudel");
pstmt.addBatch();

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

int [] updateCounts = pstmt.executeBatch();
connection.commit();
     }
    catch ( SQLException sqlException )
     {
       sqlException.printStackTrace();
     } // end catch
```

The three question marks (?) in the the preceding SQL statement's last line are placeholders for values that will be passed as part of the query to the database. Before executing a PreparedStatement, the program must specify the parameter values by using the **Prepared- Statement interface's set methods.**

For the preceding query, parameters are int and strings that can be set with Prepared- Statement method **setInt and setString.**
Method setInt's and setString's **first argument represents the parameter number being set, and the second argument is that parameter's value**. Parameter numbers are counted from 1, starting with the first question mark (?).
Interface PreparedStatement provides set methods for each supported SQL type. **It's important to use the set method that is appropriate for the parameter's SQL type in the database—SQLExceptions occur when a program attempts to convert a parameter value to an incorrect type.**

Transaction Processing
Many database applications require guarantees that a series of database insertions, updates and deletions executes properly before the application continues processing the next database operation. For example, when you transfer money electronically between bank accounts, several factors determine if the transaction is successful. You begin by specifying the source account and the amount you wish to transfer from that account to a destination account. Next, you specify the destination account. The bank checks the source account to determine whether its funds are sufficient to complete the transfer. If so, the bank withdraws the specified amount and, if all goes well, deposits it into the destination account to complete the transfer. What happens if the transfer fails after the bank withdraws the money from the source account? In a proper banking system, the bank redeposits the money in the source account.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.**

The way to allow two or more statements to be **grouped into a transaction** is to **disable the auto-commit mode.**
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.
The way to allow two or more statements to be **grouped into a transaction** is to **disable the auto-commit mode.**
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**.
con.commit();
Rollback
If you group update statements to a transaction, then the transaction either succeeds in its entirety and it can be committed, or it fails somewhere in the middle. In that case, you can carry out a **rollback** and **the database automatically undoes the effect of all updates that occurred since the last committed transaction**.

You turn off autocommit mode with the command
conn.setAutoCommit(false);
Now you create a statement object in the normal way:
Statement stat = conn.createStatement();
Call executeUpdate any number of times:
stat.executeUpdate(command1);

stat.executeUpdate(command2);
stat.executeUpdate(command3);
. . .
When all commands have been executed, call the commit method:
conn.commit();
However, if an error occurred, call
conn.rollback();
Then, all commands until the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a SQLException.

Save Points
You can gain finer-grained control over the rollback process by using save points. Creating a save point marks a point to which you can later return without having to return to the start of the transaction. For example,

```
Statement stat = conn.createStatement();     // start transaction; rollback() goes here
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint();        // set savepoint; rollback(svpt) goes here
stat.executeUpdate(command2);
if (. . .) conn.rollback(svpt);              // undo effect of command2
. . .
conn.commit();
```

Here, we used an anonymous save point. You can also give the save point a name, such as
Savepoint svpt = conn.setSavepoint("stage1");
When you are done with a save point, **you should release it:**
stat.releaseSavepoint(svpt);

```
import java.sql.Connection;
import java.sql.Statement;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.BatchUpdateException;
import java.sql.PreparedStatement;


public class DisplayAuthors
{
  // database URL
  static final String DATABASE_URL = "jdbc:mysql://localhost/books";

  // launch the application
  public static void main( String args[] )
  {
    Connection connection = null; // manages connection
    Statement statement = null; // query statement
```

```
  ResultSet resultSet = null; // manages results

  try {
   // establish connection to database
     connection = DriverManager.getConnection(
       DATABASE_URL, "root", "" );
   statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
   ResultSet uprs= statement.executeQuery(
       "SELECT * FROM authors");

   while (uprs.next()) {
            uprs.updateString( "LastName","Sharma");
      uprs.updateRow();
   }
   }
 catch ( SQLException sqlException )
  {
    sqlException.printStackTrace();
  } // end catch

  try {
   statement = connection.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
      ResultSet.CONCUR_UPDATABLE);

   ResultSet uprs = statement.executeQuery(
      "SELECT * FROM authors");

   uprs.moveToInsertRow();
   uprs.updateInt("AuthorID",9);
   uprs.updateString("FirstName","Subash");
   uprs.updateString("LastName","Pakhrin");
   uprs.insertRow();
   uprs.beforeFirst();
}
   catch ( SQLException sqlException )
  {
    sqlException.printStackTrace();
  } // end catch
  try {
           connection.setAutoCommit(false);
   statement = connection.createStatement();

  statement.addBatch(
      "INSERT INTO authors " +
      "VALUES('15','Hari','Shrestha')");

   statement.addBatch(
```

```
            "INSERT INTO authors " +
            "VALUES('16','Ram','Acharya')");

          statement.addBatch(
            "INSERT INTO authors  " +
            "VALUES('17','Shyam','Gautam')");

         statement.addBatch(
            "INSERT INTO authors  " +
            "VALUES('18','Govinda','Paudel')");

          int [] updateCounts = statement.executeBatch();
         connection.commit();

       } catch(BatchUpdateException b) {
          b.printStackTrace();
       } catch(SQLException ex) {
          ex.printStackTrace();
       }
        try {
           connection.setAutoCommit(false);
PreparedStatement pstmt = connection.prepareStatement(
                        "INSERT INTO authors VALUES(?, ?, ?)");
pstmt.setInt(1,19);
pstmt.setString(2, "Navin");
pstmt.setString(3,"Sharma");
pstmt.addBatch();

pstmt.setInt(1,20);
pstmt.setString(2, "Rajesh");
pstmt.setString(3,"Paudel");
pstmt.addBatch();

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

int [] updateCounts = pstmt.executeBatch();
connection.commit();
        }
       catch ( SQLException sqlException )
        {
          sqlException.printStackTrace();
        } // end catch

        // connect to database books and query database
        try
        {
```

```
      // create Statement for querying database
      statement = connection.createStatement();

      // query database
      resultSet = statement.executeQuery(
        "SELECT AuthorID, FirstName, LastName FROM authors" );

      // process query results
      ResultSetMetaData metaData = resultSet.getMetaData();
      int numberOfColumns = metaData.getColumnCount();
      System.out.println( "Authors Table of Books Database:\n" );

      for ( int i = 1; i <= numberOfColumns; i++ )
        System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
      System.out.println();

      while ( resultSet.next() )
      {
        for ( int i = 1; i <= numberOfColumns; i++ )
          System.out.printf( "%-8s\t", resultSet.getObject( i ) );
        System.out.println();
      } // end while
    }  // end try
    catch ( SQLException sqlException )
    {
      sqlException.printStackTrace();
    } // end catch

    finally // ensure resultSet, statement and connection are closed
    {
      try
      {
        resultSet.close();
        statement.close();
        connection.setAutoCommit(true);
        connection.close();

      } // end try
      catch ( Exception exception )
      {
        exception.printStackTrace();
      } // end catch
    } // end finally
  } // end main
} // end class DisplayAuthors
```

RowSet Interface

A JDBC RowSet object holds tabular data in a way that makes it more flexible and easier to use than a result set.The RowSet interface configures the database connection and prepares query statements automatically.It provides several **set methods** that allow you to specify the properties needed to establish a connection (such as the database URL, user name and password of the database) and create a Statement (such as a query). RowSet also provides several **get methods** that return these properties.

Connected and Disconnected RowSets

There are **two types of RowSet objects—connected and disconnected**. A **connected RowSet** object connects to the database once and remains connected while the object is in use. A **disconnected RowSet** object connects to the database, executes a query to retrieve the data from the database and then closes the connection. A program may change the data in a disconnected RowSet while it's disconnected. Modified data can be updated in the database
after a disconnected RowSet reestablishes the connection with the database.

Package javax.sql.rowset contains two subinterfaces of RowSet—**JdbcRowSet** and **CachedRowSet**. JdbcRowSet, a connected RowSet, acts as a wrapper around a ResultSet object and allows you to scroll through and update the rows in the ResultSet. By default, a ResultSet object is nonscrollable and read only—you must explicitly set the result set type constant to TYPE_SCROLL_INSENSITIVE and set the result set concurrency constant to CONCUR_UPDATABLE to make a ResultSet object scrollable and updatable.
A JdbcRowSet object is scrollable and updatable by default. CachedRowSet, a disconnected
RowSet, caches the data of a ResultSet in memory and disconnects from the database.Like JdbcRowSet, a CachedRowSet object is scrollable and updatable by default. A Cached-RowSet object is also serializable, so it can be passed between Java applications through a network, such as the Internet. However, CachedRowSet has a limitation—the amount of data that can be stored in memory is limited. Package javax.sql.rowset contains three other subinterfaces of RowSet:**WebRowSet,JoinRowSet and FilteredRowSet.**
(For Reference:http://docs.oracle.com/javase/tutorial/jdbc/basics/rowset.html).

JdbcRowSet

Navigating JdbcRowSet Objects

```
JdbcRowSet jdbcRs = new JdbcRowSetImpl();
jdbcRs.absolute(4);
jdbcRs.previous();
```

Updating Column Values

```
jdbcRs.absolute(3);
jdbcRs.updateString("lastName", "Sharma");
jdbcRs.updateRow();
```

Inserting Rows

```
jdbcRs.moveToInsertRow();
jdbcRs.updateInt("Author_ID", 10);
jdbcRs.updateString("FirstName", "Navin");
jdbcRs.updateString("LastName", "Sharma");
jdbcRs.insertRow();
```

Deleting Rows

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

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

// Program demonstrating JdbcRowSet

```
//JdbcRowSetTest.java
//Displaying the contents of the Authors table using JdbcRowSet.
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import javax.sql.rowset.JdbcRowSet;
import com.sun.rowset.JdbcRowSetImpl; // Sun's JdbcRowSet implementation
public class JdbcRowSetTest
{
// JDBC driver name and database URL
static final String DATABASE_URL = "jdbc:mysql://localhost/books";
static final String USERNAME = "root";
static final String PASSWORD = "";

// constructor connects to database, queries database, processes
 // results and displays results in window
public JdbcRowSetTest()
{
 // connect to database books and query database
try
 {
// specify properties of JdbcRowSet
JdbcRowSet rowSet = new JdbcRowSetImpl();
rowSet.setUrl( DATABASE_URL ); // set database URL
rowSet.setUsername( USERNAME ); // set username
rowSet.setPassword( PASSWORD ); // set password
rowSet.setCommand( "SELECT * FROM Authors" ); // set query
rowSet.execute(); // execute query
// process query results
ResultSetMetaData metaData = rowSet.getMetaData();
int numberOfColumns = metaData.getColumnCount();
System.out.println( "Authors Table of Books Database:\n" );

// display rowset header
for ( int i = 1; i <= numberOfColumns; i++ )
System.out.printf( "%-8s\t", metaData.getColumnName( i ) );
System.out.println();

// display each row
while(rowSet.next())
{
for ( int i = 1; i <= numberOfColumns; i++ )
System.out.printf( "%-8s\t",rowSet.getObject( i ));
} // end while
// close the underlying ResultSet, Statement and Connection
```

```
rowSet.close();
} // end try
catch ( SQLException sqlException )
{
sqlException.printStackTrace();
 System.exit( 1 );
} // end catch
} // end DisplayAuthors constructor
// launch the application
public static void main( String args[] )
{
JdbcRowSetTest application = new JdbcRowSetTest();
} // end main
} // end class JdbcRowSetTest
```

CachedRowSet

Creating CachedRowSet Objects:

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

Setting CachedRowSet Properties:

```
crs.setUsername(username);
crs.setPassword(password);
crs.setUrl("jdbc:mySubprotocol:mySubname");
```

Setting up command:

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

Populating CachedRowSet Objects:

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

Updating CachedRowSet Object:

```
 crs.updateInt("Author_ID",       10);
 crs.updateString("FirstName", "Navin");
            ....................
  crs.updateRow();
// Synchronizing the row back to the DB
    crs.acceptChanges(con);
```

Inserting and Deleting Rows:

```
  crs.absolute(3);
  crs.updateString("lastName", "Sharma");
  crs.updateRow();
  crs.insertRow();
  crs.moveToCurrentRow();
  crs.acceptChanges(con);
```