Unit - III

The Concept of JDBC; JDBC Driver Types; JDBC Drivers, JDBC Package, JDBC architecture, MySQL. SQL client environment. Establishing Database Connection; DML operations using JDBC connection, Statement, PreparedStatement, CallableStatement, ResultSet Object; Batch updates, Transaction Processing.

Java JDBC

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

We have discussed the above four drivers in the next chapter.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.

The current version of JDBC is 4.3. It is the stable release since 21st September, 2017. It is based on the X/Open SQL Call Level Interface. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class
- Clob class

- Types class

Why Should We Use JDBC

Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

JDBC Driver Types;

JDBC Driver

1. JDBC-ODBC bridge driver
2. Native-API driver
3. Network Protocol driver
4. Thin driver

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

1) JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

2) Native-API driver

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.

Advantage:

- performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

JDBC Package

The Java Database Connectivity (JDBC) API provides universal data access from the Java programming language. Using the JDBC API, you can access virtually any data source, from relational databases to spreadsheets and flat files. JDBC technology also provides a common base on which tools and alternate interfaces can be built.

The JDBC API is comprised of two packages:

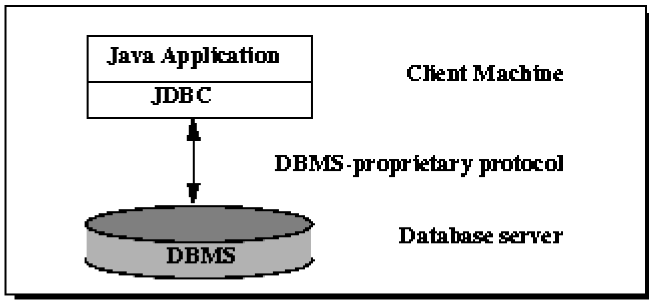
- `java.sql`
- `javax.sql`

To use the JDBC API with a particular database management system, you need a JDBC technology-based driver to mediate between JDBC technology and the database. Depending on various factors, a driver might be written purely in the Java programming language or in a mixture of the Java programming language and Java Native Interface (JNI) native methods. To obtain a JDBC driver for a particular database management system

JDBC Architecture

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

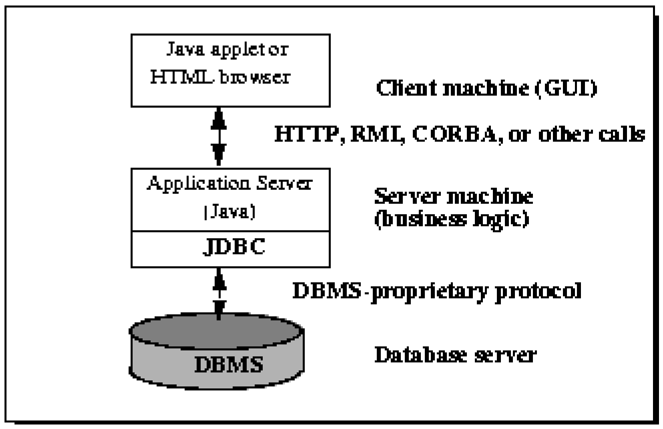
Two-tier Architecture for Data Access.



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.

Three-tier Architecture for Data Access.


Java applet or HTML browser
Client machine (GUI)
HTTP, RMI, CORBA, or other calls
Application Server (Java)
JDBC
Server machine (business logic)
DBMS-proprietary protocol
DBMS
Database server

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.

MySQL

MySQL is the most popular Open Source Relational SQL Database Management System. MySQL is one of the best RDBMS being used for developing various web-based software applications. MySQL is developed, marketed and supported by MySQL AB, which is a Swedish company. This tutorial will give you a quick start to MySQL and make you comfortable with MySQL programming.

Establishing JDBC Connection in Java

Before establishing a connection between front end i.e your Java Program and back end i.e the database we should learn what precisely a JDBC is and why it came to existence.

What is JDBC ?
JDBC is an acronym for Java Database Connectivity. It’s an advancement for ODBC (Open Database Connectivity). JDBC is an standard API specification developed in order to move data from frontend to backend. This API consists of classes and interfaces written in Java. It basically acts as an interface (not the one we use in Java) or channel between your Java program and databases i.e it establishes a link between the two so that a programmer could send data from Java code and store it in the database for future use.

Why JDBC came into existence ?
As previously told JDBC is an advancement for ODBC, ODBC being platform dependent had a lot of drawbacks. ODBC API was written in C,C++, Python, Core Java and as we know above languages (except Java and some part of Python)are platform dependent . Therefore to remove dependence, JDBC was developed by database vendor which consisted of classes and interfaces written in Java.

1. Loading the Driver
To begin with, you first need load the driver or register it before using it in the program . Registration is to be done once in your program. You can register a driver in one of two ways mentioned below :

- **Class.forName() :** Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following example uses Class.forName() to load the Oracle driver –

  ```
  Class.forName("oracle.jdbc.driver.OracleDriver");
  ```

- **DriverManager.registerDriver():** DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time . The following example uses DriverManager.registerDriver()to register the Oracle driver –

  ```
  DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
  ```

2. **Create the connections**
After loading the driver, establish connections using :

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

user – username from which your sql command prompt can be accessed.
password – password from which your sql command prompt can be accessed.

con: is a reference to Connection interface.
url : Uniform Resource Locator. It can be created as follows:

```
String url = " jdbc:oracle:thin:@localhost:1521:xe"
```

Where oracle is the database used, thin is the driver used , @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by programmer before calling the function. Use of this can be referred from final code.

3. Create a statement
Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.
Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Here, con is a reference to Connection interface used in previous step .

4. Execute the query
Now comes the most important part i.e executing the query. Query here is an SQL Query . Now we know we can have multiple types of queries. Some of them are as follows:

- Query for updating / inserting table in a database.
- Query for retrieving data .

The executeQuery() method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.
The executeUpdate(sql query) method ofStatement interface is used to execute queries of updating/inserting .

```
int m = st.executeUpdate(sql);
if (m==1)
    System.out.println("inserted successfully : "+sql);
else
    System.out.println("insertion failed");
```

Here sql is sql query of the type String

5.Close the connections
So finally we have sent the data to the specified location and now we are at the verge of completion of our task .
By closing connection, objects of Statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.
Example :

```
 con.close();
```

DML operations using JDBC connection,

Insert()

Delete()

Select()

Update()

Statement interface

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

Commonly used methods of Statement interface:

The important methods of Statement interface are as follows:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

4) public int[] executeBatch(): is used to execute batch of commands.

JDBC PreparedStatement

A Java JDBC *PreparedStatement* is a special kind of Java JDBC Statement object with some useful additional features. Remember, you need a `Statement` in order to execute either a query or an update. You can use a Java JDBC `PreparedStatement` instead of a `Statement` and benefit from the features of the `PreparedStatement`.

The Java JDBC `PreparedStatement` primary features are:

- Easy to insert parameters into the SQL statement.
- Easy to reuse the `PreparedStatement` with new parameter values.
- May increase performance of executed statements.
- Enables easier batch updates.

I will show you how to insert parameters into SQL statements in this text, and also how to reuse a `PreparedStatement`. The batch updates is explained in a separate text.

Here is a quick example, to give you a sense of how it looks in code:

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);

int rowsAffected = preparedStatement.executeUpdate();
```

Creating a PreparedStatement

Before you can use a `PreparedStatement` you must first create it. You do so using the `Connection.prepareStatement()`, like this:

```
String sql = "select * from people where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);
```

The `PreparedStatement` is now ready to have parameters inserted.

Inserting Parameters into a PreparedStatement

Everywhere you need to insert a parameter into your SQL, you write a question mark (?). For instance:

```
String sql = "select * from people where id=?";
```

Once a `PreparedStatement` is created (prepared) for the above SQL statement, you can insert parameters at the location of the question mark. This is done using the many `setXXX()` methods. Here is an example:

```
preparedStatement.setLong(1, 123);
```

The first number (1) is the index of the parameter to insert the value for. The second number (123) is the value to insert into the SQL statement.

Here is the same example with a bit more details:

```
String sql = "select * from people where id=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setLong(123);
```

You can have more than one parameter in an SQL statement. Just insert more than one question mark. Here is a simple example:

```
String sql = "select * from people where firstname=? and lastname=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setString(1, "John");
preparedStatement.setString(2, "Smith");
```

Executing the PreparedStatement

Executing the `PreparedStatement` looks like executing a regular `Statement`. To execute a query, call the `executeQuery()` or `executeUpdate` method. Here is an `executeQuery()` example:

```
String sql = "select * from people where firstname=? and lastname=?";

PreparedStatement preparedStatement =
        connection.prepareStatement(sql);

preparedStatement.setString(1, "John");
preparedStatement.setString(2, "Smith");
```

```
ResultSet result = preparedStatement.executeQuery();
```

The `executeQuery()` method returns a `ResultSet`. Iterating the `ResultSet` is described in the Query the Database text.

Here is an `executeUpdate()` example:

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement =  connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);

int rowsAffected = preparedStatement.executeUpdate();
```

The `executeUpdate()` method is used when updating the database. It returns an int which tells how many records in the database were affected by the update.

Reusing a PreparedStatement

Once a `PreparedStatement` is prepared, it can be reused after execution. You reuse a `PreparedStatement` by setting new values for the parameters and then execute it again. Here is a simple example:

```
String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement = connection.prepareStatement(sql);

preparedStatement.setString(1, "Gary");
preparedStatement.setString(2, "Larson");
preparedStatement.setLong  (3, 123);
int rowsAffected = preparedStatement.executeUpdate();
preparedStatement.setString(1, "Stan");
preparedStatement.setString(2, "Lee");
preparedStatement.setLong  (3, 456);
int rowsAffected = preparedStatement.executeUpdate();
```

This works for executing queries too, using the `executeQuery()` method, which returns a `ResultSet`.

PreparedStatement Performance

It takes time for a database to parse an SQL string, and create a query plan for it. A query plan is an analysis of how the database can execute the query in the most efficient way.

If you submit a new, full SQL statement for every query or update to the database, the database has to parse the SQL and for queries create a query plan. By reusing an existing

`PreparedStatement` you can reuse both the SQL parsing and query plan for subsequent queries. This speeds up query execution, by decreasing the parsing and query planning overhead of each execution.

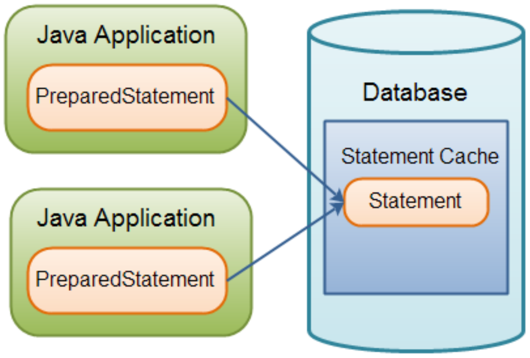
There are two levels of potential reuse for a `PreparedStatement`.

1. Reuse of PreparedStatement by the JDBC driver.
2. Reuse of PreparedStatement by the database.

First of all, the JDBC driver can cache `PreparedStatement` objects internally, and thus reuse the `PreparedStatement` objects. This may save a little of the `PreparedStatement` creation time.

Second, the cached parsing and query plan could potentially be reused across Java applications, for instance application servers in a cluster, using the same database.

Here is a diagram illustrating the caching of statements in the database:



The caching of PreparedStatement's in the database.

JDBC Batch Updates

- Statement Batch Updates
- PreparedStatement Batch Updates
- Adding Batches in a Loop
- Batch Updates and Transactions

A JDBC batch update is a batch of updates grouped together, and sent to the database in one *batch*, rather than sending the updates one by one.

Sending a batch of updates to the database in one go, is faster than sending them one by one, waiting for each one to finish. There is less network traffic involved in sending one batch of updates (only 1 round trip), and the database might be able to execute some of the updates in parallel. The speed up compared to executing the updates one by one, can be quite big.

You can batch both SQL inserts, updates and deletes. It does not make sense to batch select statements.

There are two ways to execute a *JDBC batch update*:

1. Using a Statement
2. Using a PreparedStatement

This JDBC batch update tutorial explains both ways in the following sections.

Statement Batch Updates

You can use a `Statement` object to execute batch updates. You do so using the `addBatch()` and `executeBatch()` methods. Here is an example:

```
Statement statement = null;

try{
    statement = connection.createStatement();

    statement.addBatch("update people set firstname='John' where id=123");
    statement.addBatch("update people set firstname='Eric' where id=456");
    statement.addBatch("update people set firstname='May'  where id=789");

    int[] recordsAffected = statement.executeBatch();
} finally {
    if(statement != null) statement.close();
}
```

First you add the SQL statements to be executed in the batch, using the `addBatch()` method.

Then you execute the SQL statements using the `executeBatch()`. The `int[]` array returned by the `executeBatch()` method is an array of `int` telling how many records were affected by each executed SQL statement in the batch.

PreparedStatement Batch Updates

You can also use a `PreparedStatement` object to execute batch updates. The `PreparedStatement` enables you to reuse the same SQL statement, and just insert new parameters into it, for each update to execute. Here is an example:

```
String sql = "update people set firstname=? , lastname=? where id=?";


PreparedStatement preparedStatement = null;
try{
    preparedStatement =
            connection.prepareStatement(sql);

    preparedStatement.setString(1, "Gary");
    preparedStatement.setString(2, "Larson");
    preparedStatement.setLong  (3, 123);

    preparedStatement.addBatch();

    preparedStatement.setString(1, "Stan");
    preparedStatement.setString(2, "Lee");
    preparedStatement.setLong  (3, 456);

    preparedStatement.addBatch();

    int[] affectedRecords = preparedStatement.executeBatch();

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

First a `PreparedStatement` is created from an SQL statement with question marks in, to show where the parameter values are to be inserted into the SQL.

Second, each set of parameter values are inserted into the preparedStatement, and the `addBatch()` method is called. This adds the parameter values to the batch internally. You can now add another set of values, to be inserted into the SQL statement. Each set of parameters are inserted into the SQL and executed separately, once the full batch is sent to the database.

Third, the `executeBatch()` method is called, which executes all the batch updates. The SQL statement plus the parameter sets are sent to the database in one go. The `int[]` array returned by the `executeBatch()` method is an array of `int` telling how many records were affected by each executed SQL statement in the batch.

Adding Batches in a Loop

Most often you will be adding batches to a `Statement` or `PreparedStatement` from inside a for loop or a while loop. Each iteration in the loop will add one batch. Here is an example of adding batches to a `PreparedStatement` from inside a for-loop:

```
List<Person> persons = ... // get a list of Person objects from somewhere.


String sql = "update people set firstname=? , lastname=? where id=?";

PreparedStatement preparedStatement = null;
try{
    preparedStatement =
            connection.prepareStatement(sql);
    for(Person person : persons) {
        preparedStatement.setString(1, person.getFirstName());
        preparedStatement.setString(2, person.getLastName());
        preparedStatement.setLong  (3, person.getId());

        preparedStatement.addBatch();
    }

    int[] affectedRecords = preparedStatement.executeBatch();

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

In the example above you get a list of `Person` objects from somewhere. This part is left out of the example, as it is not so relevant where this list comes from. What matters is how the list is iterated, and values from each `Person` object is added to the batch. After adding all `Person` objects to the batch, the batch update is executed.

By the way, imagine that the used `Person` class looks like this:

```
public class Person{
    private String firstName = null;
    private String lastName  = null;
    private long   id        = -1;

    public String getFirstName() {
        return this.firstName;
    }

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

    public String getLastName() {
        return this.lastName;
    }

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

    public long getId() {
        return this.id;
    }

    public void setId(long id) {
        this.id = id;
    }
}
```

Batch Updates and Transactions

It is important to keep in mind, that each update added to a `Statement` or `PreparedStatement` is executed separately by the database. That means, that some of them may succeed before one of them fails. All the statements that have succeeded are now applied to the database, but the rest of the updates may not be. This can result in an inconsistent data in the database.

To avoid this, you can execute the batch update inside a JDBC transaction. When executed inside a transaction you can make sure that either all updates are executed, or none are. Any successful updates can be rolled back, in case one of the updates fail.

JDBC: Transactions

A transaction is a set of actions to be carried out as a single, atomic action. Either all of the actions are carried out, or none of them are.

The classic example of when transactions are necessary is the example of bank accounts. You need to transfer $100 from one account to the other. You do so by subtracting $100 from the first account, and adding $100 to the second account. If this process fails after you have subtracted the $100 fromt the first bank account, the $100 are never added to the second bank account. The money is lost in cyber space.

To solve this problem the subtraction and addition of the $100 are grouped into a transaction. If the subtraction succeeds, but the addition fails, you can "rollback" the fist subtraction. That way the database is left in the same state as before the subtraction was executed.

You start a transaction by this invocation:

```
connection.setAutoCommit(false);
```

Now you can continue to perform database queries and updates. All these actions are part of the transaction.

If any action attempted within the transaction fails, you should rollback the transaction. This is done like this:

```
connection.rollback();
```

If all actions succeed, you should commit the transaction. Committing the transaction makes the actions permanent in the database. Once committed, there is no going back. Committing the transaction is done like this:

```
connection.commit();
```

Of course you need a bit of try-catch-finally around these actions. Here is a an example:

```
Connection connection = ...
try{
    connection.setAutoCommit(false);

    // create and execute statements etc.

    connection.commit();
} catch(Exception e) {
    connection.rollback();
} finally {
    if(connection != null) {
        connection.close();
    }
}
```

Here is a full example:

```
Connection connection = ...
try{
    connection.setAutoCommit(false);


    Statement statement1 = null;
    try{
        statement1 = connection.createStatement();
        statement1.executeUpdate(
            "update people set name='John' where id=123");
    } finally {
        if(statement1 != null) {
            statement1.close();
        }
    }


    Statement statement2 = null;
    try{
        statement2 = connection.createStatement();
        statement2.executeUpdate(
            "update people set name='Gary' where id=456");
    } finally {
        if(statement2 != null) {
            statement2.close();
        }
    }

    connection.commit();
} catch(Exception e) {
    connection.rollback();
} finally {
    if(connection != null) {
        connection.close();
    }
}
```

JDBC: CallableStatement

A `java.sql.CallableStatement` is used to call stored procedures in a database.

A stored procedure is like a function or method in a class, except it lives inside the database. Some database heavy operations may benefit performance-wise from being executed inside the same memory space as the database server, as a stored procedure.

Creating a CallableStatement

You create an instance of a `CallableStatement` by calling the `prepareCall()` method on a connection object. Here is an example:

```
CallableStatement callableStatement =
    connection.prepareCall("{call calculateStatistics(?, ?)}");
```

If the stored procedure returns a `ResultSet`, and you need a non-default `ResultSet` (e.g. with different holdability, concurrency etc. characteristics), you will need to specify these characteristics already when creating the `CallableStatement`. Here is an example:

```
CallableStatement callableStatement =
    connection.prepareCall("{call calculateStatistics(?, ?)}",
        ResultSet.TYPE_FORWARD_ONLY,
        ResultSet.CONCUR_READ_ONLY,
        ResultSet.CLOSE_CURSORS_OVER_COMMIT
    );
```

Setting Parameter Values

Once created, a `CallableStatement` is very similar to a `PreparedStatement`. For instance, you can set parameters into the SQL, at the places where you put a ? . Here is an example:

```
CallableStatement callableStatement =
    connection.prepareCall("{call calculateStatistics(?, ?)}");

callableStatement.setString(1, "param1");
callableStatement.setInt   (2, 123);
```

Executing the CallableStatement

Once you have set the parameter values you need to set, you are ready to execute the `CallableStatement`. Here is how that is done:

```
ResultSet result = callableStatement.executeQuery();
```

The `executeQuery()` method is used if the stored procedure returns a `ResultSet`.

If the stored procedure just updates the database, you can call the `executeUpdate()` method instead, like this:

```
callableStatement.executeUpdate();
```

Batch Updates

You can group multiple calls to a stored procedure into a batch update. Here is how that is done:

```
CallableStatement callableStatement =
    connection.prepareCall("{call calculateStatistics(?, ?)}");

callableStatement.setString(1, "param1");
callableStatement.setInt   (2, 123);
callableStatement.addBatch();

callableStatement.setString(1, "param2");
callableStatement.setInt   (2, 456);
callableStatement.addBatch();

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

OUT Parameters

A stored procedure may return `OUT` parameters. That is, values that are returned instead of, or in addition to, a `ResultSet`. After executing the `CallableStatement` you can then access these `OUT` parameters from the `CallableStatement` object. Here is an example:

```
CallableStatement callableStatement =
    connection.prepareCall("{call calculateStatistics(?, ?)}");

callableStatement.setString(1, "param1");
callableStatement.setInt   (2, 123);

callableStatement.registerOutParameter(1, java.sql.Types.VARCHAR);
callableStatement.registerOutParameter(2, java.sql.Types.INTEGER);

ResultSet result = callableStatement.executeQuery();
while(result.next()) { ... }

String out1 = callableStatement.getString(1);
int    out2 = callableStatement.getInt   (2);
```

It is recommended that you first process the `ResultSet` before trying to access any `OUT` parameters.

JDBC: DatabaseMetaData

Through the `java.sql.DatabaseMetaData` interface you can obtain meta data about the database you have connected to. For instance, you can see what tables are defined in the database, and what columns each table has, whether given features are supported etc.

The `DatabaseMetaData` interface contains a lot of methods, should check out the JavaDoc's.

Obtaining a DatabaseMetaData Instance

You obtain the `DatabaseMetaData` object from a `Connection`, like this:

```
DatabaseMetaData databaseMetaData = connection.getMetaData();
```

Once you have obtained this `DatabaseMetaData` instance, you can call methods on it to obtain the meta data about the database.

Database Product Name and Version

You can obtain the database product name and version, like this:

```
int    majorVersion   = databaseMetaData.getDatabaseMajorVersion();
int    minorVersion   = databaseMetaData.getDatabaseMinorVersion();

String productName    = databaseMetaData.getDatabaseProductName();
String productVersion = databaseMetaData.getDatabaseProductVersion();
```

If you already know exactly what database your application is running against, you may not need this. But, if you are developing a product that needs to be able to run against many different database products, this information can be quite handy in determining what database specific features it supports, SQL it supports etc.

Database Driver Version

You can obtain the driver version of the JDBC driver used, like this:

```
int driverMajorVersion = databaseMetaData.getDriverMajorVersion();
int driverMinorVersion = databaseMetaData.getDriverMinorVersion();
```

Again, if your application runs against a very specific database, this may not really be so informative. However, for applications that need to be able to run against many different database products and versions, knowing the exact version of the used driver may be an advantage. For instance, a certain driver version may contain a bug that the application need to work around. Or, the driver may be missing a feature that the application then needs to work around.

Listing Tables

You can obtain a list of the defined tables in your database, via the `DatabaseMetaData`. Here is how that is done:

```
String   catalog          = null;
String   schemaPattern    = null;
String   tableNamePattern = null;
String[] types            = null;

ResultSet result = databaseMetaData.getTables(
    catalog, schemaPattern, tableNamePattern, types );

while(result.next()) {
    String tableName = result.getString(3);
}
```

First you call the `getTables()` method, passing it 4 parameters which are all null. The parameters can help limit the number of tables that are returned in the `ResultSet`. However, since I want all tables returned, I passed null in all of these parameters. See the JavaDoc for more specific details about the parameters.

The `ResultSet` returned from the `getTables()` method contains a list of table names matching the 4 given parameters (which were all null). This `ResultSet` contains 10 columns, which each contain information about the given table. The column with index 3 contains the table name itself. Check the JavaDoc for more details about the rest of the columns.

Listing Columns in a Table

You can obtain the columns of a table via the `DatabaseMetaData` too. Here is how:

```
String   catalog           = null;
String   schemaPattern     = null;
String   tableNamePattern  = "my_table";
String   columnNamePattern = null;


ResultSet result = databaseMetaData.getColumns(
    catalog, schemaPattern,  tableNamePattern, columnNamePattern);
```

```
while(result.next()){
    String columnName = result.getString(4);
    int    columnType = result.getInt(5);
}
```

First you call the `getColumns()` method, passing 4 parameters. Of these, only the `tableNamePattern` is set to a non-null value. Set it to the name of the table you want to obtain the columns of.

The `ResultSet` returned by the `getColumns()` method contains a list of columns for the given table. The column with index 4 contains the column name, and the column with index 5 contains the column type. The column type is an integer matching one of the type constants found in `java.sql.Types`

To get more details about obtaining column information for tables, check out the JavaDoc.

Primary Key for Table

It is also possible to obtain the primary key of a table. You do so, like this:

```
String   catalog   = null;
String   schema    = null;
String   tableName = "my_table";

ResultSet result = databaseMetaData.getPrimaryKeys(
    catalog, schema, tableName);

while(result.next()){
    String columnName = result.getString(4);
}
```

First you call the `getPrimaryKeys()` method, passing 3 parameters to it. Only the `tableName` is non-null in this example.

The `ResultSet` returned by the `getPrimaryKeys()` method contains a list of columns which make up the primary key of the given table. The column with index 4 contains the column name.

A primary key may consist of multiple columns. Such a key is called a compound key. If your tables contains compound keys, the `ResultSet` will contain multiple rows. One row for each column in the compound key.

Supported Features

The `DatabaseMetaData` object also contains information about the features the JDBC driver and the database supports. Many of these features are represented by a method you can call, which will return true or false depending on whether the given feature is supported or not.

```
databaseMetaData.supportsGetGeneratedKeys();
databaseMetaData.supportsGroupBy();

databaseMetaData.supportsOuterJoins();
```