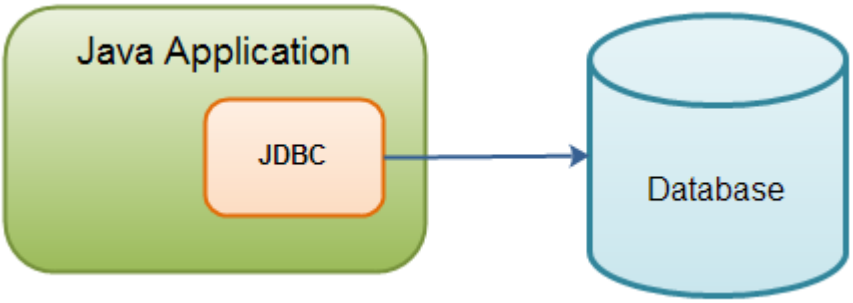
The Java *JDBC* API (Java Database Connectivity) enables Java applications to connect to relational databases like MySQL, PostgreSQL, MS SQL Server, Oracle, H2 Database etc. The JDBC API makes is possible to query and update relational databases, as well as call stored procedures, and obtain meta data about the database. The Java JDBC API is part of the core Java SE SDK, making JDBC available to all Java applications that want to use it. Here is a diagram illustrating a Java application using JDBC to connect to a relational database:



JDBC is Database Independent

The Java JDBC API standardizes how to connect to a database, how to execute queries against it, how to navigate the result of such a query, and how to execute updates in the database, how to call stored procedures, and how to obtain meta data from the database. By "standardizes" I mean that the code looks the same across different database products. Thus, changing to another database will be a lot easier, if your project needs that in the future.

JDBC is Not SQL Independent

JDBC does not standardize the SQL sent to the database. The SQL is written by you, the user of the JDBC API. The SQL dialect used by the various different databases will vary slightly, so to be 100% database independent, you SQL must also be 100% database independent (i.e. use commands understood by all databases).

The core concepts of the Java JDBC API are:

- Driver
- Connection
- Statement
- PreparedStatement
- CallableStatement
- ResultSet
- Transaction
- DatabaseMetaData

```
import java.sql.*;

public class JdbcExample {

    public static void main(String[] args) throws ClassNotFoundException {
        Class.forName("org.h2.Driver");

        String url      = "jdbc:h2:~/test";   //database specific url.
        String user     = "sa";
        String password = "";

        try(Connection connection = DriverManager.getConnection(url, user,
password)) {
            try(Statement statement = connection.createStatement()){
                String sql = "select * from people";
                try(ResultSet result = statement.executeQuery(sql)){
                    while(result.next()) {
                        String name = result.getString("name");
                        long   age  = result.getLong  ("age");
                    }
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }
}
```

Load JDBC Driver

```
Class.forName("DRIVER_NAME");  // name of the JDBC driver class for the given
JDBC driver
```

You will have to find the right class name for the JDBC driver you are using. Typically, each database has its own JDBC driver, so you will have to look up what the JDBC driver class name is.

Open Database Connection

In order to communicate with the database, you must first open a JDBC connection to the database.

```
String url      = "jdbc:h2:~/test";   //database specific url.
String user     = "sa";
String password = "";

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

Once you have an open database connection, you will typically either be updating the database (inserting new records or updating existing ones), or query the database, meaning you read records from it

Create Statement

Whether you need to update or query the database, you will need to create a JDBC `Statement` or PreparedStatement through which the update or query will happen. Here is an example of creating a JDBC `Statement` instance:

```
Statement statement = connection.createStatement();
```

Update the Database

When you have created a JDBC `Statement` or JDBC `PreparedStatement` instance you can update the database. Here is an example of updating the database via a `Statement` instance:

```
Statement statement = connection.createStatement();

String    sql       = "update people set name='John' where id=123";

int rowsAffected    = statement.executeUpdate(sql);
```

Query the Database

You can also query the database via a JDBC `Statement` or `PreparedStatement` object. When you query the database you get a JDBC ResultSet back through which you can access the result of the query. Here is an example of executing a query against a database via JDBC:

```
Statement statement = connection.createStatement();

String sql = "select * from people";

ResultSet result = statement.executeQuery(sql);

while(result.next()) {

    String name = result.getString("name");
    long   age  = result.getLong  ("age");

}
```

Close Database Connection

When you are done with the JDBC database connection, you have to close the connection again. A JDBC connection can take up a big amount of sources both inside your application, but also inside the database server. Therefore it is important to close the database connection again after use. You close a JDBC connection via its `close()` method. Here is an example of closing a JDBC connection:

```
connection.close();
```

The JDBC API consists of the following core parts:

- JDBC Drivers
- Connections
- Statements
- Result Sets

There are four basic JDBC use cases around which most JDBC work evolves:

- Query the database (read data from it).
- Query the database meta data.
- Update the database.
- Perform transactions.

JDBC Drivers

A JDBC driver is a collection of Java classes that enables you to connect to a certain database. For instance, MySQL will have its own JDBC driver.

```
JDBC drivers are typically supplied by the database vendor
```

Connections

Once a JDBC driver is loaded and initialized, you need to connect to the database. You do so by obtaining a `Connection` to the database via the JDBC API and the loaded driver. All communication with the database happens via a connection. An application can have more than one connection open to a database at a time. This is actually very common.

Statements

A `Statement` is what you use to execute queries and updates against the database. There are a few different types of statements you can use. Each statement corresponds to a single query or update.

ResultSets

When you perform a query against the database you get back a `ResultSet`. You can then traverse this `ResultSet` to read the result of the query.

Common JDBC Use Cases

Query the database

One of the most common use cases is to read data from a database. Reading data from a database is called querying the database.

Query the database meta data

Another common use case is to query the database meta data. The database meta data contains information about the database itself. For instance, information about the tables defined, the columns in each table, the data types etc.

Update the database

Another very common JDBC use case is to update the database. Updating the database means writing data to it. In other words, adding new records or modifying (updating) existing records.

Perform transactions

Transactions is anoter common use case. A transaction groups multiple updates and possibly queries into a single action. Either all of the actions are executed, or none of them are.

JDBC Driver Type List

There are four different JDBC driver types. These driver types are:

- Type 1: JDBC-ODBC bridge JDBC driver
- Type 2: Java + Native code JDBC driver
- Type 3: All Java + Middleware translation JDBC driver
- Type 4: All Java JDBC driver.

The JDBC *Connection* class, java.sql.Connection, represents a database connection to a relational database. Before you can read or write data from and to a database via JDBC, you need to open a connection to the database.

Loading the JDBC Driver

The first thing you need to do before you can open a JDBC connection to a database is to load the JDBC driver for the database. Actually, from Java 6 this is no longer necessary, but doing so will not fail. You load the JDBC driver like this:

```
Class.forName("driverClassName");
```

Each JDBC driver has a primary driver class that initializes the driver when it is loaded. For instance, to load the H2Database driver, you write this:

```
Class.forName("org.h2.Driver");
```

You only have to load the driver once. You do not need to load it before every connection opened. Only before the first JDBC connection opened.

Opening the JDBC Connection

You open a JDBC Connection by call the `java.sql.DriverManager` class method `getConnection()`. There are three variants of this method. I will show each variant in the following sections.

Open Connection With URL, User and Password

The second variant of `getConnection()` takes both a database URL, a user name and a password as parameters. Here is an example of calling that variant of `getConnection()` :

```
String url      = "jdbc:h2:~/test";   //database specific url.
String user     = "sa";
String password = "";

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

The `user` and `password` parameters are the user name and password for your database.

JDBC: Query the Database

Querying a database means searching through its data. You do so be sending SQL statements to the database. To do so, you first need an open database connection. Once you have an open connection, you need to create a `Statement` object, like this:

```
Statement statement = connection.createStatement();
```

Once you have created the `Statement` you can use it to execute SQL queries, like this:

```
String sql = "select * from people";

ResultSet result = statement.executeQuery(sql);
```

When you execute an SQL query you get back a `ResultSet`. The `ResultSet` contains the result of your SQL query. The result is returned in rows with columns of data. You iterate the rows of the `ResultSet` like this:

```
while(result.next()) {

    String name = result.getString("name");
    long   age  = result.getLong  ("age");

}
```

You can get column data for the current row by calling some of the `getXXX()` methods, where XXX is a primitive data type. For instance:

```
    result.getString    ("columnName");
    result.getLong      ("columnName");
    result.getInt       ("columnName");
    result.getDouble    ("columnName");
    result.getBigDecimal("columnName");
```

```
    etc.
```

The column name to get the value of is passed as parameter to any of these `getXXX()` method calls.

You can also pass an index of the column instead, like this:

```
    result.getString    (1);
    result.getLong      (2);
    result.getInt       (3);
    result.getDouble    (4);
    result.getBigDecimal(5);
    etc.
```

For that to work you need to know what index a given column has in the `ResultSet`. You can get the index of a given column by calling the `ResultSet.findColumn()` method, like this:

```
    int columnIndex = result.findColumn("columnName");
```

If iterating large amounts of rows, referencing the columns by their index might be faster than by their name.

When you are done iterating the `ResultSet` you need to close both the `ResultSet` and the `Statement` object that created it (if you are done with it, that is). You do so by calling their `close()` methods, like this:

```
result.close();
statement.close();
```

Of course you should call these mehtods inside a `finally` block to make sure that they are called even if an exception occurs during `ResultSet` iteration.

```
Statement statement = connection.createStatement();

String sql = "select * from people";

ResultSet result = statement.executeQuery(sql);

while(result.next()) {

    String name = result.getString("name");
    long   age  = result.getLong("age");

    System.out.println(name);
    System.out.println(age);
}

result.close();
statement.close();
```

```
Statement statement = null;

try{
    statement = connection.createStatement();
    ResultSet result    = null;
    try{
        String sql = "select * from people";
        ResultSet result = statement.executeQuery(sql);

        while(result.next()) {

            String name = result.getString("name");
            long   age  = result.getLong("age");

            System.out.println(name);
            System.out.println(age);
        }
    } finally {
        if(result != null) result.close();
    }
} finally {
    if(statement != null) statement.close();
}
```

JDBC: Update the Database

In order to update the database you need to use a `Statement`. But, instead of calling the `executeQuery()` method, you call the `executeUpdate()` method.

There are two types of updates you can perform on a database:

1. Update record values
2. Delete records

The `executeUpdate()` method is used for both of these types of updates.

Updating Records

Here is an update record value example:

```
Statement statement = connection.createStatement();

String    sql       = "update people set name='John' where id=123";

int rowsAffected    = statement.executeUpdate(sql);
```

The `rowsAffected` returned by the `statement.executeUpdate(sql)` call, tells how many records in the database were affected by the SQL statement.

Deleting Records

Here is a delete record example:

```
Statement statement = connection.createStatement();

String    sql       = "delete from people where id=123";

int rowsAffected    = statement.executeUpdate(sql);
```

Again, the `rowsAffected` returned by the `statement.executeUpdate(sql)` call, tells how many records in the database were affected by the SQL statement.

JDBC Statement

The Java JDBC *Statement*, `java.sql.Statement`, interface is used to execute SQL statements against a relational database.

Create Statement

In order to use a Java JDBC `Statement` you first need to create a `Statement`. Here is an example of creating a Java `Statement` instance:

```
Statement statement = connection.createStatement();
```

Executing a Query via a Statement

Once you have created a Java Statement object, you can execute a query against the database. You do so by calling its executeQuery() method, passing an SQL statement as parameter. The Statement executeQuery() method returns a Java JDBC ResultSet which can be used to navigate the response of the query. Here is an example of calling the Java JDBC Statement executeQuery() and navigating the returned ResultSet:

```
String sql = "select * from people";

ResultSet result = statement.executeQuery(sql);

while(result.next()) {
   String name = result.getString("name");
   long   age  = result.getLong  ("age");
}
```

Execute an Update via a Statement

You can also execute an update of the database via a Java JDBC `Statement` instance. For instance, you could execute an SQL insert, update or delete via a `Statement` instance. Here is an example of executing a database update via a Java JDBC `Statement` instance:

```
tatement statement = connection.createStatement();

String    sql       = "update people set name='John' where id=123";

int rowsAffected    = statement.executeUpdate(sql);
```

Statement vs. PreparedStatement

The Java JDBC API has an interface similar to the `Statement` called PreparedStatement . The `PreparedStatement` can have parameters inserted into the SQL statement, so the `PreparedStatement` can be reused again and again with different parameter values. You cannot do that with a `Statement`. A `Statement` requires a finished SQL statement as parameter.

JDBC ResultSet

The Java JDBC *ResultSet* interface represents the result of a database query. The text about queries shows how the result of a query is returned as a `java.sql.ResultSet`. This `ResultSet` is then iterated to inspect the result.

A ResultSet Contains Records

A JDBC `ResultSet` contains records. Each records contains a set of columns. Each record contains the same amount of columns, although not all columns may have a value. A column can have a `null` value. Here is an illustration of a JDBC `ResultSet`:

Name	Age	Gender
John	27	Male
Jane	21	Female
Jeanie	31	Female

ResultSet example - records with columns

This `ResultSet` has 3 different columns (Name, Age, Gender), and 3 records with different values for each column.

Creating a ResultSet

You create a `ResultSet` by executing a `Statement` or `PreparedStatement`, like this:

```
Statement statement = connection.createStatement();

ResultSet result = statement.executeQuery("select * from people");

String sql = "select * from people";
PreparedStatement statement = connection.prepareStatement(sql);

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

Iterating the ResultSet

To iterate the `ResultSet` you use its `next()` method. The `next()` method returns true if the `ResultSet` has a next record, and moves the `ResultSet` to point to the next record. If there were no more records, `next()` returns false, and you can no longer. Once the `next()` method has returned false, you should not call it anymore. Doing so may result in an exception.

Here is an example of iterating a `ResultSet` using the `next()` method:

```
while(result.next()) {
    // ... get column values from this record
}
```

As you can see, the `next()` method is actually called before the first record is accessed. That means, that the `ResultSet` starts out pointing before the first record. Once `next()` has been called once, it points at the first record.

Similarly, when `next()` is called and returns false, the `ResultSet` is actually pointing after the last record.

You cannot obtain the number of rows in a `ResultSet` except if you iterate all the way through it and count the rows.

Accessing Column Values

When iterating the `ResultSet` you want to access the column values of each record. You do so by calling one or more of the many `getXXX()` methods. You pass the name of the column to get the value of, to the many `getXXX()` methods. For instance:

```
while(result.next()) {

    result.getString    ("name");
    result.getInt       ("age");
    result.getBigDecimal("coefficient");

    // etc.
}
```

There are a lot of `getXXX()` methods you can call, which return the value of the column as a certain data type, e.g. String, int, long, double, BigDecimal etc. They all take the name of the column to obtain the column value for, as parameter.

The `getXXX()` methods also come in versions that take a column index instead of a column name. For instance:

```
while(result.next()) {

    result.getString    (1);
    result.getInt       (2);
    result.getBigDecimal(3);

    // etc.
}
```

The index of a column typically depends on the index of the column in the SQL statement. For instance, the SQL statement

```
select name, age, coefficient from person
```

has three columns. The column name is listed first, and will thus have index 1 in the `ResultSet`. The column age will have index 2, and the column coefficient will have index 3.

Sometimes you do not know the index of a certain column ahead of time. For instance, if you use a `select * from` type of SQL query, you do not know the sequence of the columns.

JDBC PreparedStatement

A Java JDBC *PreparedStatement* is a special kind ofJava 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.

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);
```

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();
```

As you can see, the `executeQuery()` method returns a `ResultSet`

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();
```

A 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 batch updates:

1. Using a Statement
2. Using a PreparedStatement

This text explains both ways.

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.

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(?, ?)}");
```

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();
```