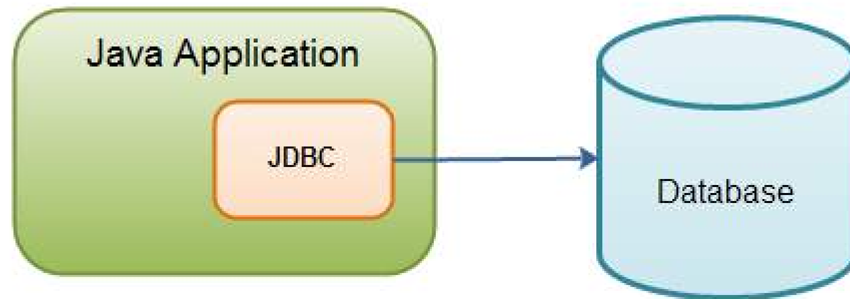
JDBC(JAVA DATABASE CONNECTIVITY)

The Java JDBC API enables Java applications to connect to relational databases via a standard API, so your Java applications become independent (almost) of the database the application uses.



Java application using JDBC to connect to a database.

JDBC 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 exeucte updates in the database. JDBC does not standardize the SQL sent to the database. This may still vary from database to database.

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.

I will explain both the core component and common use cases in the following sections.

Core JDBC Components

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. A JDBC driver implements a lot of the JDBC interfaces. When your code uses a given JDBC driver, it actually just uses the standard JDBC interfaces. The concrete JDBC driver used is hidden behind the JDBC interfaces. Thus you can plugin a new JDBC driver without your code noticing it.

Of course, the JDBC drivers may vary a little in the features they support.

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

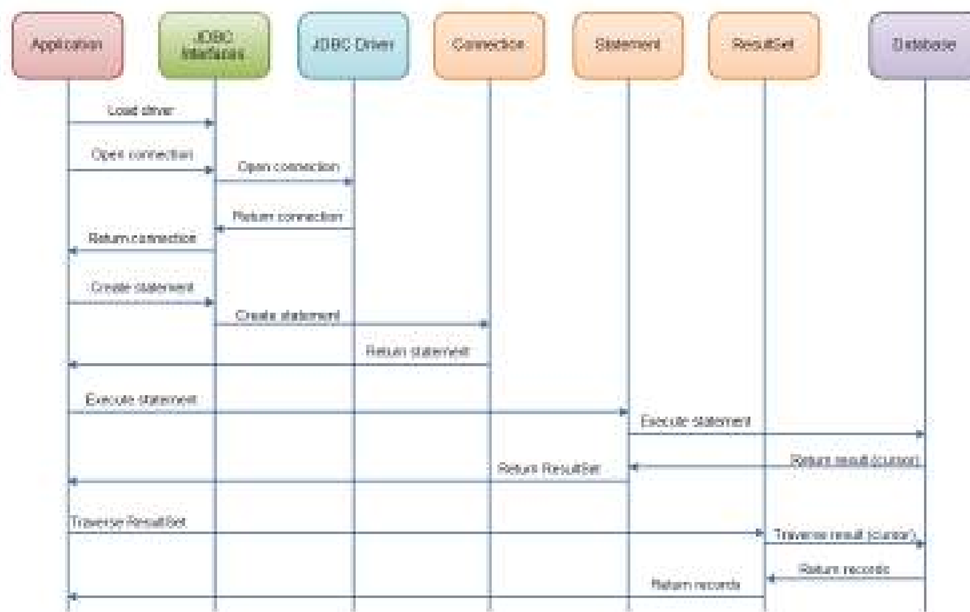
Another very common JDBC use case is to update the database. Updating the database means writing data to it. In other words, adding new records or modifying (updating) existing records.

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.

A JDBC Component Interaction Diagram

Here is an example of how the core components interact in during the execution of a database query (click image to view larger version):



Java JDBC: Interaction of the core JDBC components during the execution of a database query.

A JDBC driver is a set of Java classes that implement the JDBC interfaces, targeting a specific database. The JDBC interfaces comes with standard Java, but the implementation of these interfaces is specific to the database you need to connect to. Such an implementation is called a JDBC driver.

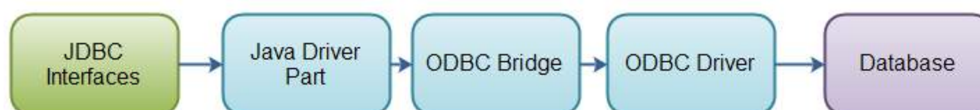
There are 4 different types of JDBC drivers:

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

Today, most drivers are type 4 drivers. Nevertheless, I will just discuss the 4 types of drivers shortly.

Type 1 JDBC Driver

A type 1 JDBC driver consists of a Java part that translates the JDBC interface calls to ODBC calls. An ODBC bridge then calls the ODBC driver of the given database. Type 1 drivers are (were) mostly intended to be used in the beginning, when there were no type 4 drivers (all Java drivers). Here is an illustration of how a type 1 JDBC driver is organized:



Type 1 JDBC driver.

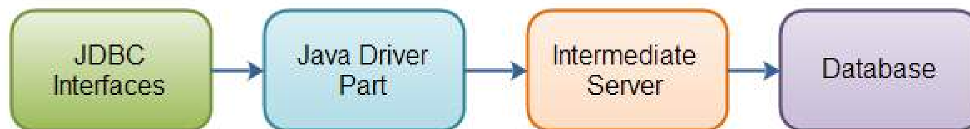
Type 2 JDBC Driver

A type 2 JDBC driver is like a type 1 driver, except the ODBC part is replaced with a native code part instead. The native code part is targeted at a specific database product. Here is an illustration of a type 2 JDBC driver:

Type 2 JDBC driver.

Type 3 JDBC Driver

A type 3 JDBC driver is an all Java driver that sends the JDBC interface calls to an intermediate server. The intermediate server then connects to the database on behalf of the JDBC driver. Here is an illustration of a type 3 JDBC driver:



Type 3 JDBC driver.

Type 4 JDBC Driver

A type 4 JDBC driver is an all Java driver which connects directly to the database. It is implemented for a specific database product. Today, most JDBC drivers are type 4 drivers. Here is an illustration of how a type 4 JDBC driver is organized:

Type 4 JDBC driver.

Before you can read or write data in a database via JDBC, you need to open a connection to the database. This text will show you how to do that.

Loading the JDBC Driver

The first thing you need to do before you can open a database connection 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 connection opened.

Opening the Connection

To open a database connection you use the `java.sql.DriverManager` class. You call its getConnection() method, like this:

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

String user     = "sa";

String password = "";



Connection connection =

    DriverManager.getConnection(url, user, password);
```

The `url` is the url to your database. You should check the documentation for your database and JDBC driver to see what the format is for your specific database. The url shown above is for a H2Database.

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

Closing the Connection

Once you are done using the database connection you should close it. This is done by calling the Connection.close() method, like this:

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

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");



}
```

The `ResultSet.next()` method moves to the next row in the `ResultSet`, if there are anymore rows. If there are anymore rows, it returns true. If there were no more rows, it will return false.

You need to call `next()` at least one time before you can read any data. Before the first `next()` call the`ResultSet` is positioned before the first row.

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.

Full Example

Here is a full query code example:

```
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();
```

And here the example is again, with `try-finally` blocks added. Notice, I have left out the `catch` blocks to make the example shorter.

```
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();

}
```

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.

The text about **queries** I shows how the result of a query is returned as a `java.sql.ResultSet`. This`ResultSet` is then iterated to inspect the result. This text will examine the `ResultSet` interface in a few more details.

A ResultSet Contains Records

A `ResultSet` consists of 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 `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");
```

Or like this:

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

PreparedStatement statement = connection.prepareStatement(sql);



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

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

The `PreparedStatement`'s primary features are:

- Easy to insert parameters into the SQL statement.
- Easy to reuse the `PreparedStatement` with new parameters.
- 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();
```

As you can see, 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.

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();
```