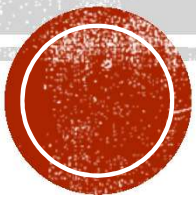
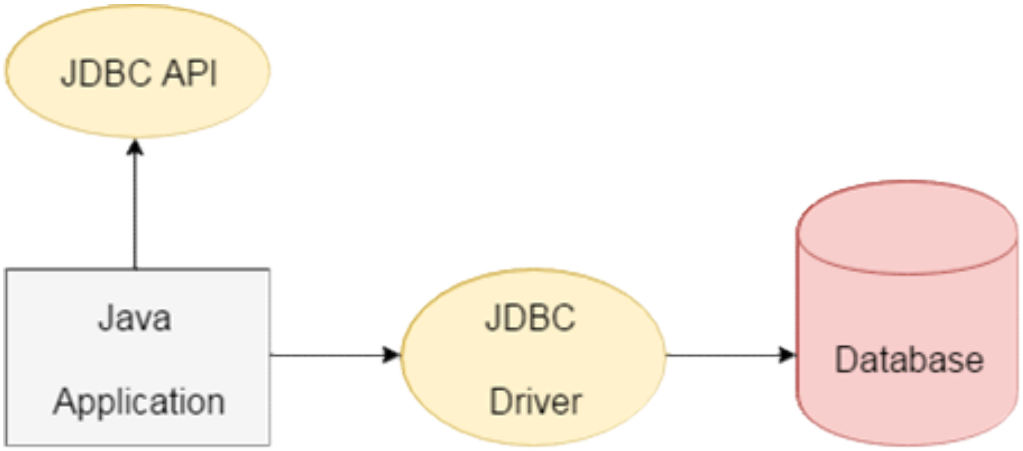
JDBC

Java Database Connectivity



WHAT IS JDBC ?

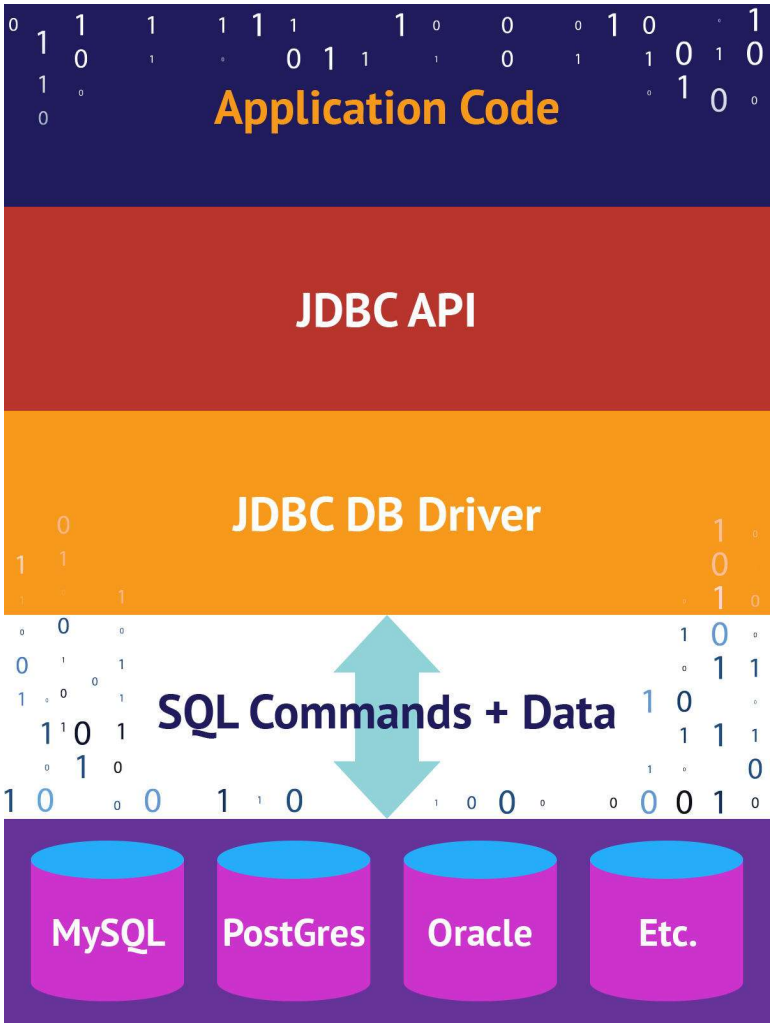
- JDBC is an standard API specification developed in order to move data from frontend to backend.
- It basically acts as a channel between your Java program and databases i.e it establishes a link between the two so that a programmer could send data from Java code and store it in the database for future use.



WHY JDBC?

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.



ARCHITECTURAL OVERVIEW OF JDBC

JDBC offers a programming-level interface that handles the mechanics of Java applications communicating with a database or RDBMS. The JDBC interface consists of two layers:

1. The JDBC API supports communication between the Java application and the JDBC manager.
2. The JDBC driver supports communication between the JDBC manager and the database driver.

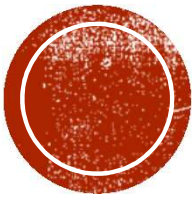
- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- CallableStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

INTERFACES IN JDBC API

Mainly used interfaces

CLASSES OF JDBC API

- DriverManager class
- Blob class
- Clob class
- Types class



STEPS TO CONNECT DB TO JAVA PROGRAM

STEPS

Step 1

Loading the Driver

To begin with, you first need load the driver or register it before using it in the program .

```
Class.forName("com.mysql.cj.jdbc.Driver");
```

Here we load the driver's class file into memory at the runtime.

STEPS

Step 2

Create the connections

After loading the driver, establish connections using :

```
Connection con =
DriverManager.getConnection (
"jdbc:mysql://localhost/batch?“ +
"user=root&password=amma“ );
```

Create an object of Connection and setup the connection with the DB.

STEPS

Step 3

Create a statement

Once connection is established interact with DB.

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

The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.

STEPS

Step 4

Execute the query

Executing the SQL queries

```
ResultSet rs= st.executeQuery("select * from batch.student");
```

ResultSet get the result of the SQL query

STEPS

Step 5

Close the connection

Once done with the interaction to db close the connection.

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

By closing connection, objects of Statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

JDBC CONNECTION

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.

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.

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

Each JDBC driver has a primary driver class that initializes the driver when it is loaded.

JDBC CONNECTION

You open a JDBC Connection by call the java.sql.DriverManager class method getConnection(). There are three variants of this method.

1. Open Connection With URL

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

Connection connection =
    DriverManager.getConnection(url);
```

2. Open Connection With URL and Properties

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

Properties properties = new Properties( );
properties.put( "user", "sa" );
properties.put( "password", "" );

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

3. Open Connection With URL, User and Password

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

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

STATEMENT

Executing a Query via a Statement

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

Once you have created a Java Statement object, you can execute a query against the database.

This is done 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.

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

ResultSet result = statement.executeQuery(sql);

while(result.next()) {

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

}
```

One could execute an SQL insert, update or delete via a Statement instance.

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

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

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

STATEMENT

Update via a Statement

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.

PREPAREDSTATEMENT

- 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.

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.

```
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

- Before you can use a PreparedStatement you must first create it. You do so using the Connection.prepareStatement()

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

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

PREPAREDSTATEMENT

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

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

- Once a PreparedStatement is created (prepared), then can insert parameters at the location of the question mark. This is done using the many setXXX() methods.

```
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.

PREPAREDSTATEMENT

- 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);
```

PREPAREDSTATEMENT

- You can have more than one parameter in an SQL statement. Just insert more than one question mark.

```
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

- To execute a query, call the executeQuery() or executeUpdate method.
  - executeQuery( )

```
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();
```

  - executeUpdate( )

```
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();
```

RESULTSET

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.

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

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

You create a ResultSet by executing a Statement or PreparedStatement.

```
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();
```

RESULTSET

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

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.

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

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.

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

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

    // etc.
}
```

ACCESSING COLUMN VALUES

- The getXXX() methods also come in versions that take a column index instead of a column name.

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

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

    // etc.
}
```

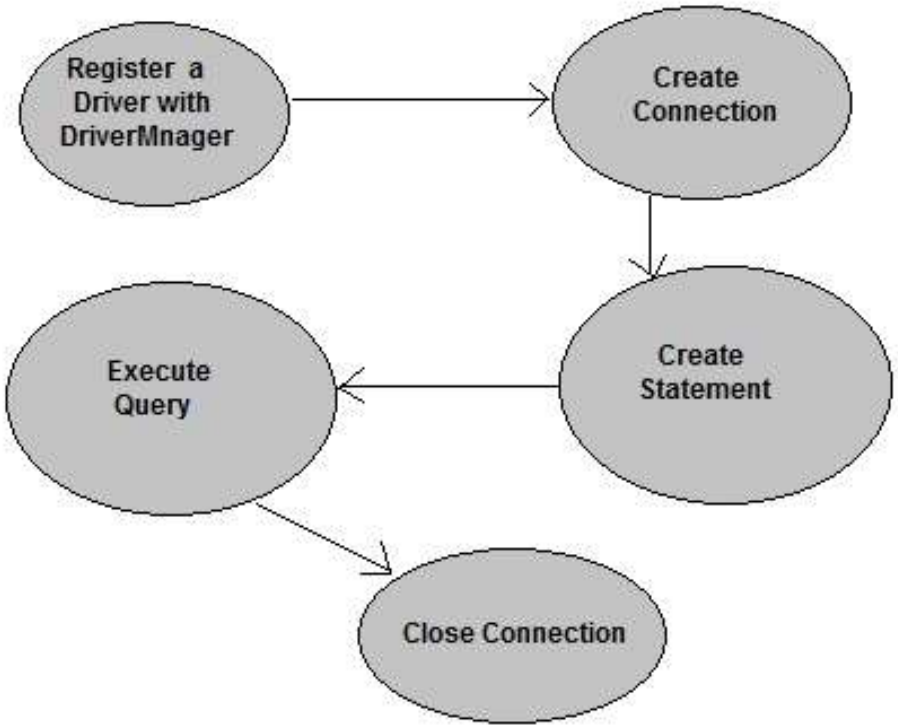
TO FIND THE INDEX OF A COLUMN

- If you do not know the index of a certain column you can find the index of that column using the ResultSet.findColumn(String columnName) method

```
int nameIndex   = result.findColumn("name");
int ageIndex    = result.findColumn("age");
int coeffIndex  = result.findColumn("coefficient");

while(result.next()) {
   String     name        = result.getString     (nameIndex);
   int        age         = result.getInt        (ageIndex);
   BigDecimal coefficient = result.getBigDecimal
(coeffIndex);
}
```

IN BRIEF


Register a Driver with DriverMnager
Create Connection
Create Statement
Execute Query
Close Connection