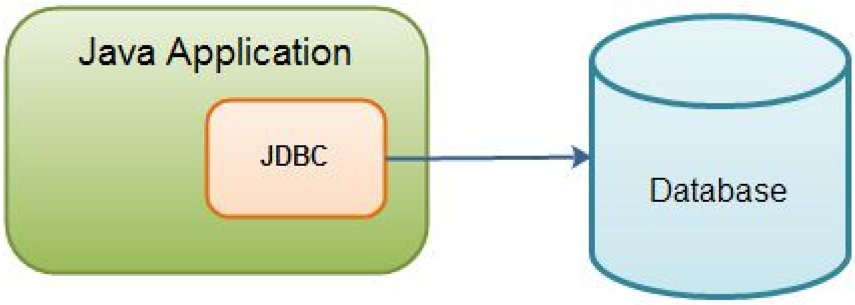
Link to mysql driver: **https://dev.mysql.com/downloads/connector/j/**

Java JDBC

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.

JDBC Overview

The JDBC API consists of the following core parts:

- JDBC Drivers
- Connections
- Statements
- Result Sets

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

Reading data from a database is called querying the database.

Update the database

Updating the database means writing data to it. In other words, adding new records or modifying (updating) existing records.

JDBC: Database Connections

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 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();
```

JDBC: Query the Database

Querying a database means searching through its data. You do so by 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.

Here is a full query code example:

```
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();
}
```

ResultSet Type, Concurrency and Holdability

When you create a ResultSet there are three attributes you can set. These are:

1. Type
2. Concurrency
3. Holdability

You set these already when you create the Statement or PreparedStatement, like this:

```
Statement statement = connection.createStatement(
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_OVER_COMMIT
   );

PreparedStatement statement = connection.prepareStatement(sql,
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_READ_ONLY,
    ResultSet.CLOSE_CURSORS_OVER_COMMIT
   );
```

Precisely what these attributes mean is explained later in this text. But now you now where to specify them.

ResultSet Types

A ResultSet can be of a certain type. The type determines some characteristics and abilities of the ResultSet.
Not all types are supported by all databases and JDBC drivers. You will have to check your database and JDBC driver to see if it supports the type you want to use. The DatabaseMetaData.supportsResultSetType(int type) method returns true or false depending on whether the given type is supported or not. The DatabaseMetaData class is covered in a later text.
At the time of writing there are three ResultSet types:

1. ResultSet.TYPE_FORWARD_ONLY
2. ResultSet.TYPE_SCROLL_INSENSITIVE
3. ResultSet.TYPE_SCROLL_SENSITIVE

The default type is TYPE_FORWARD_ONLY

TYPE_FORWARD_ONLY means that the ResultSet can only be navigated forward. That is, you can only move from row 1, to row 2, to row 3 etc. You cannot move backwards in the ResultSet.
TYPE_SCROLL_INSENSITIVE means that the ResultSet can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position, or jump to an absolute position. The ResultSet is insensitive to changes in the underlying data source while the ResultSet is open. That is, if a record in the ResultSet is changed in the database by another thread or process, it will not be reflected in already opened ResulsSet's of this type.
TYPE_SCROLL_SENSITIVE means that the ResultSet can be navigated (scrolled) both forward and backwards. You can also jump to a position relative to the current position, or jump to an absolute position. The ResultSet is sensitive to changes in the underlying data source while the ResultSet is open. That is, if a record in the ResultSet is changed in the database by another thread or process, it will be reflected in already opened ResulsSet's of this type.

Navigation Methods

The ResultSet interface contains the following navigation methods. Remember, not all methods work with all ResultSet types. What methods works depends on your database, JDBC driver, and the ResultSettype.

Method	Description
absolute()	Moves the ResultSet to point at an absolute position. The position is a row number passed as parameter to the absolute() method.
afterLast()	Moves the ResultSet to point after the last row in the ResultSet.
beforeFirst()	Moves the ResultSet to point before the first row in the ResultSet.
first()	Moves the ResultSet to point at the first row in the ResultSet.
last()	Moves the ResultSet to point at the last row in the ResultSet.
next()	Moves the ResultSet to point at the next row in the ResultSet.
previous()	Moves the ResultSet to point at the previous row in the ResultSet.
relative()	Moves the ResultSet to point to a position relative to its current position. The relative position is passed as a parameter to the relative method, and can be both positive and negative.

The ResultSet interface also contains a set of methods you can use to inquire about the current position of the ResultSet. These are:

Method	Description
getRow()	Returns the row number of the current row - the row currently pointed to by the ResultSet.
getType()	Returns the ResultSet type.
isAfterLast()	Returns true if the ResultSet points after the last row. False if not.
isBeforeFirst()	Returns true if the ResultSet points before the first row. False if not.
isFirst()	Returns true if the ResultSet points at the first row. False if not.

Finally the ResultSet interface also contains a method to update a row with database changes, if the ResultSet is sensitive to change.

Method	Description
refreshRow()	Refreshes the column values of that row with the latest values from the database.

ResultSet Concurrency

The ResultSet concurrency determines whether the ResultSet can be updated, or only read. Some databases and JDBC drivers support that the ResultSet is updated, but not all databases and JDBC drivers do. The DatabaseMetaData.supportsResultSetConcurrency(int concurrency) method returns true or false depending on whether the given concurrency mode is supported or not. The DatabaseMetaData class is covered in a later text.
A ResultSet can have one of two concurrency levels:

1. ResultSet.CONCUR_READ_ONLY
2. ResultSet.CONCUR_UPDATABLE

CONCUR_READ_ONLY means that the ResultSet can only be read.
CONCUR_UPDATABLE means that the ResultSet can be both read and updated.

Updating a ResultSet

If a ResultSet is updatable, you can update the columns of each row in the ResultSet. You do so using the many updateXXX() methods. For instance:

```
  result.updateString     ("name"       , "Alex");
   result.updateInt        ("age"        , 55);
   result.updateBigDecimal ("coefficient", new BigDecimal("0.1323");
   result.updateRow();
```

You can also update a column using column index instead of column name. Here is an example:

```
  result.updateString     (1, "Alex");
   result.updateInt       (2, 55);
   result.updateBigDecimal (3, new BigDecimal("0.1323");
   result.updateRow();
```

Notice the updateRow() call. It is when updateRow() is called that the database is updated with the values of the row. Id you do not call this method, the values updated in the ResultSet are never sent to the database. If you call updateRow() inside a transaction, the data is not actually committed to the database until the transaction is committed.

Inserting Rows into a ResultSet

If the ResultSet is updatable it is also possible to insert rows into it. You do so by:

1. call ResultSet.moveToInsertRow()
2. update row column values
3. call ResultSet.insertRow()

Here is an example:

```
result.moveToInsertRow();
result.updateString     (1, "Alex");
result.updateInt        (2, 55);
result.updateBigDecimal (3, new BigDecimal("0.1323");
result.insertRow();

result.beforeFirst();
```

The row pointed to after calling moveToInsertRow() is a special row, a buffer, which you can use to build up the row until all column values has been set on the row.
Once the row is ready to be inserted into the ResultSet, call the insertRow() method.
After inserting the row the ResultSet still pointing to the insert row. However, you cannot be certain what will happen if you try to access it, once the row has been inserted. Therefore you should move the ResultSet to a valid position after inserting the new row. If you need to insert another row, explicitly call moveToInsertRow() to signal this to the ResultSet.

ResultSet Holdability

The ResultSet holdability determines if a ResultSet is closed when the commit() method of the underlying connection is called.
Not all holdability modes are supported by all databases and JDBC drivers. TheDatabaseMetaData.supportsResultSetHoldability(int holdability) returns true or false depending on whether the given holdability mode is supported or not. The DatabaseMetaData class is covered in a later text.
There are two types of holdability:

1. ResultSet.CLOSE_CURSORS_OVER_COMMIT

2. ResultSet.HOLD_CURSORS_OVER_COMMIT

The CLOSE_CURSORS_OVER_COMMIT holdability means that all ResultSet instances are closed when connection.commit() method is called on the connection that created the ResultSet.
The HOLD_CURSORS_OVER_COMMIT holdability means that the ResultSet is kept open when the connection.commit() method is called on the connection that created the ResultSet.
The HOLD_CURSORS_OVER_COMMIT holdability might be useful if you use the ResultSet to update values in the database. Thus, you can open a ResultSet, update rows in it, call connection.commit() and still keep the same ResultSet open for future transactions on the same rows.

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: PreparedStatement

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.

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.

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 PreparedStatementby 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.

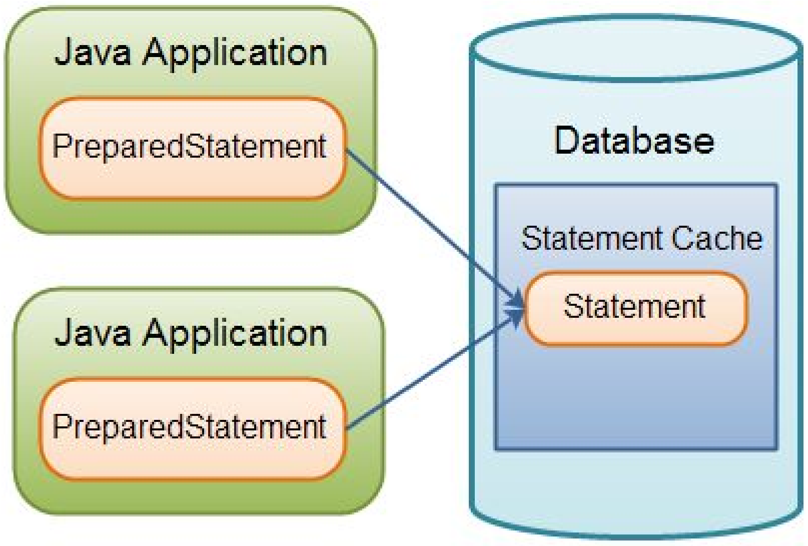
By reusing an existing PreparedStatement you can reuse both the SQL parsing and query plan for subsequent queries.

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:



JDBC: Batch Updates

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.