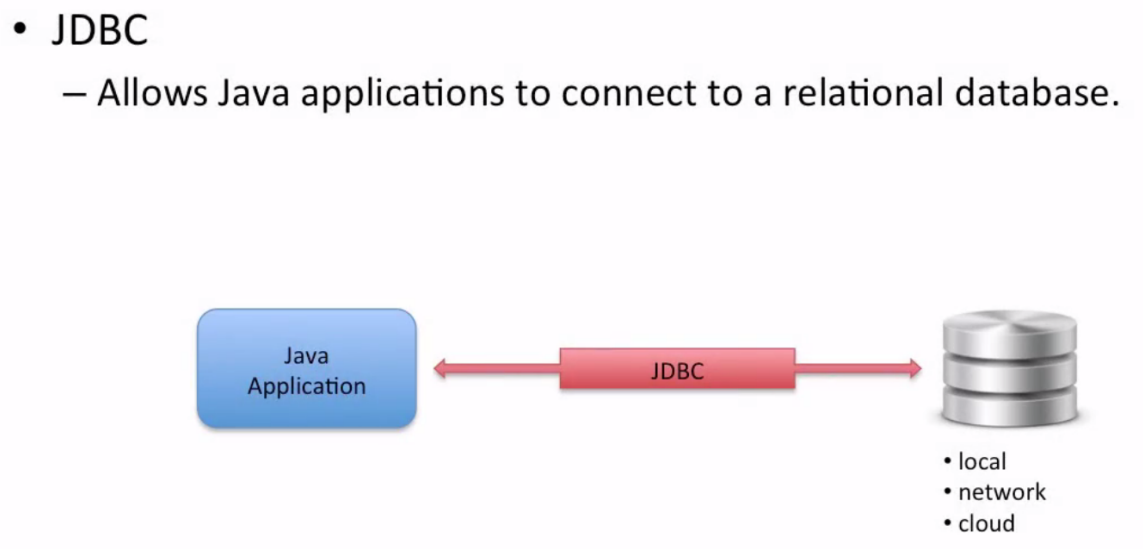
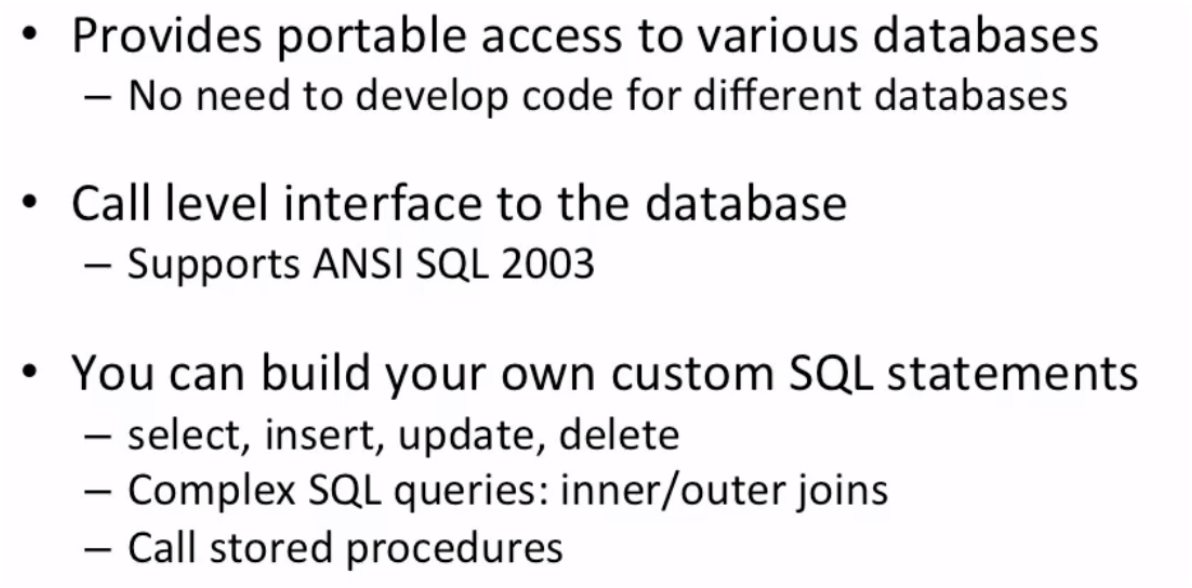
JDBC Notes

What is JDBC ?



JDBC is standard API --> Portable access to various databases. We don't need to develop code for different databases.

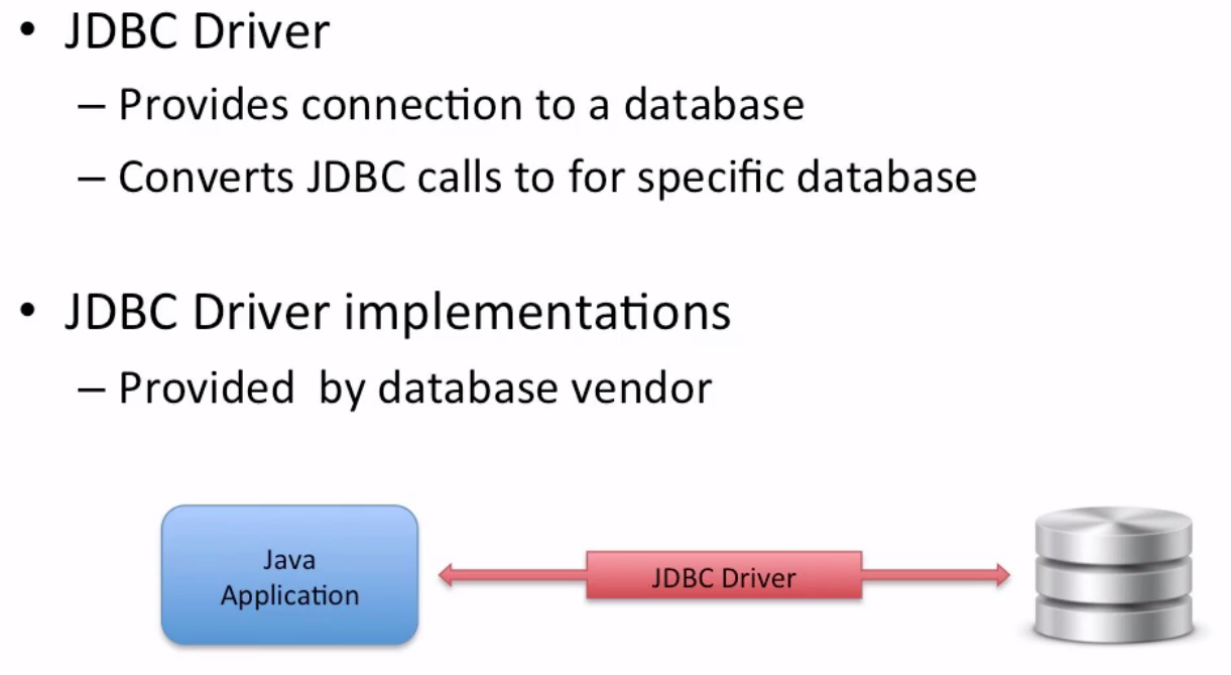


Databases Supported :

- JDBC supports a large number of databases.
- Oracle, DB2, MySQL, SQLServer, Postgress, etc.

JDBC Architecture :

Main and critical component is JDBC Driver, which is a connection between your application and Database.



Your database vendor typically provides a JDBC driver specific to the database.

JDBC Driver Manager – database connection string:

- Driver Manager helps to connect an application, based on the **database connection string**.
- In JDBC 4.0, the JDBC drivers are automatically loaded based on the classpath.
- Legacy JDBC 3.0 drivers have to be explicitly loaded with **Class.forName(theDriveName)**

JDBC API :

API is defined in two packages : java.sql and javax.sql

Key classes are :

- java.sql.DriverManager
- java.sql.Connection
- java.sql.Statement
- java.sql.ResultSet
- javax.sql.DataSource (for connection pooling)

JDBC Development Process :

- Get a Connection to Database
- Create Statement object
- Execute SQL query
- Process Result set

Step 1: Get a connection to the Database – use Connection string in form of JDBC URL

What you need here ?

Need a Connection String in the form of JDBC URL.

Basic syntax of JDBC URL :

jdbc:<driver_protocol>:<driver_connection_details>

Examples :

Database	JDBC URL
MS SQL Server	jdbc:odbc:DemoDSN
Oracle	jdbc:oracle:thin@myserver:1521:demodb
MySQL	jdbc:mysql://localhost:3306/demodb

- **Code snippet for connecting to MySQL:** Using DriverManager.getConnection() method.

```
import java.sql.*;

...
String dbUrl = "jdbc:mysql://localhost:3306/demo";
String user = "student";
String pass = "student";

Connection myConn = DriverManager.getConnection(dbUrl, user, pass);
```

Failure to connect will throw an exception:

- *java.sql.SQLException: bad url or credentials*
- *java.lang.ClassNotFoundException: JDBC driver not in classpath*

Step 2: Create Statement object :

- The Statement object is based on connection**.
- This statement will be used later to execute SQL query.

```
import java.sql.*;

...
String dbUrl = "jdbc:mysql://localhost:3306/demo";
String user = "student";
String pass = "student";

Connection myConn = DriverManager.getConnection(dbUrl, user, pass);

Statement myStmt = myConn.createStatement();
```

Step 3: Execute SQL query :

- pass in your SQL query to Statement object.
- The result of execution of query will give ResultSet.

```
import java.sql.*;

...
String dbUrl = "jdbc:mysql://localhost:3306/demo";
String user = "student";
String pass = "student";

Connection myConn = DriverManager.getConnection(dbUrl, user, pass);

Statement myStmt = myConn.createStatement();

ResultSet myRs = myStmt.executeQuery("select * from employees");
```

Step 4: Process the Result set

- ResultSet is initially placed before first row**.
- Method ResultSet.next()
  - moves forward one row
  - returns true if there are more rows to process

Looping through a ResultSet

```
...
ResultSet myRs = myStmt.executeQuery("select * from employees");

while (myRs.next()) {
    // read data from each row
}
```

Methods for reading data from ResultSet :

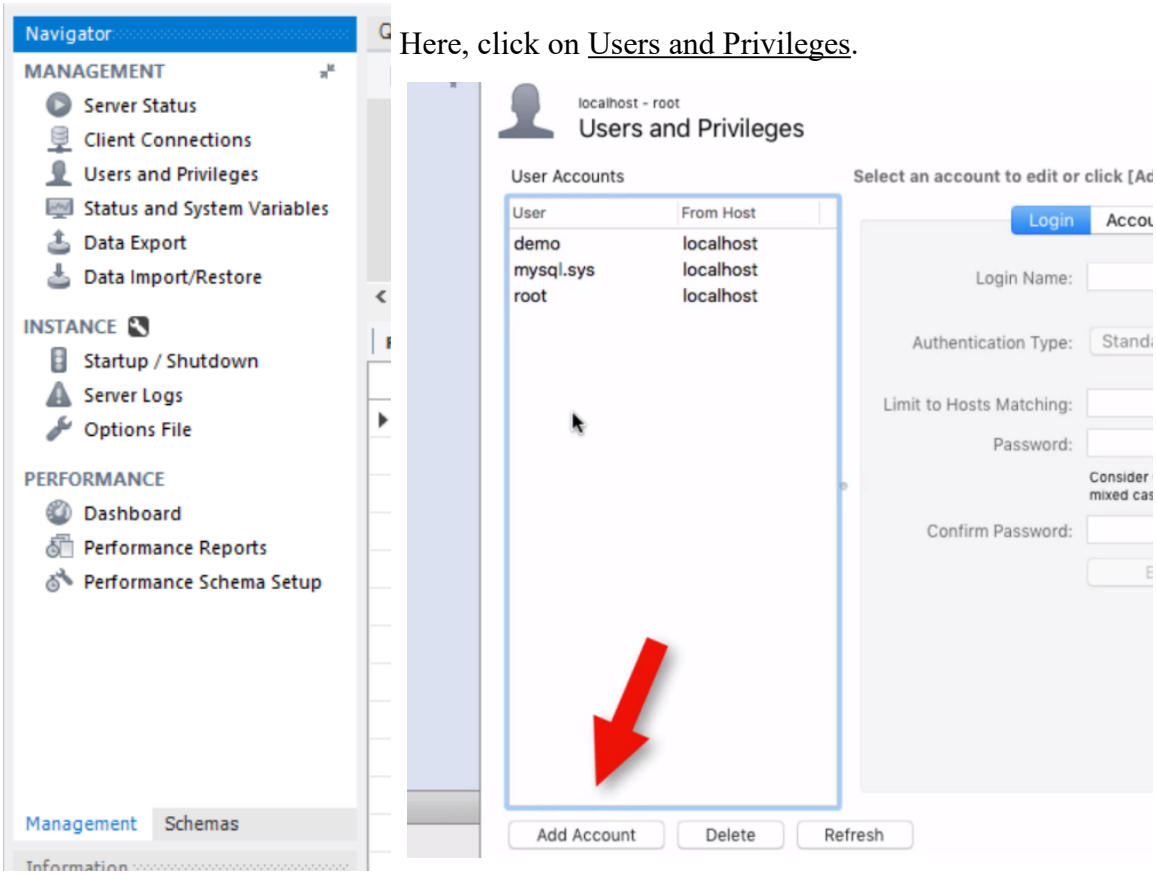
- getXXX(columnName)
- getXXX(columnIndex) --> ** columnIndex is one-based, unlike arrays which are zero-based.

```
...
ResultSet myRs = myStmt.executeQuery(
                    "select last_name, first_name from employees");

while (myRs.next()) {
    System.out.println(myRs.getString("last_name"));
    System.out.println(myRs.getString("first_name"));
}
```

For our Learning purpose, install MySQL Community server.

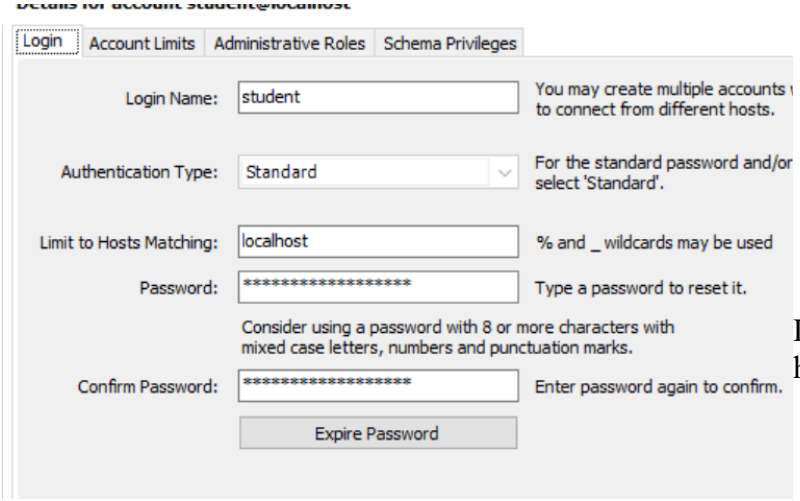
Create a new Database user in MySQL Workbench :

After you login to the MySQL workbench, click on Management section.



Here, click on Users and Privileges.

click on Add Account and enter the details like this :



Login name : student

Limit to hosts matching : localhost

password: student

In the Administrative Roles tab, select DBA so that this test user 'student' will have all privileges.

To execute already existing sql script :

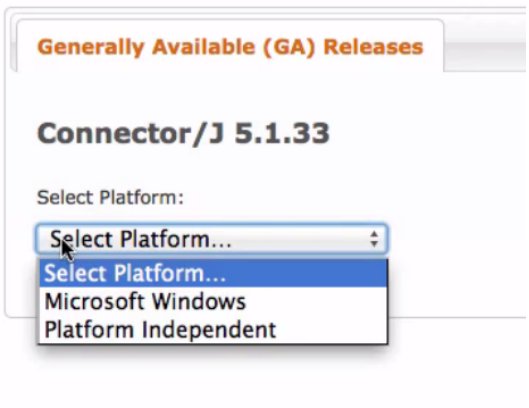
- After you login to MySQL workbench --> go to File -> Open SQL Script -> chose the SQL script you want to open and once it is loaded, click on Execute to run the script.

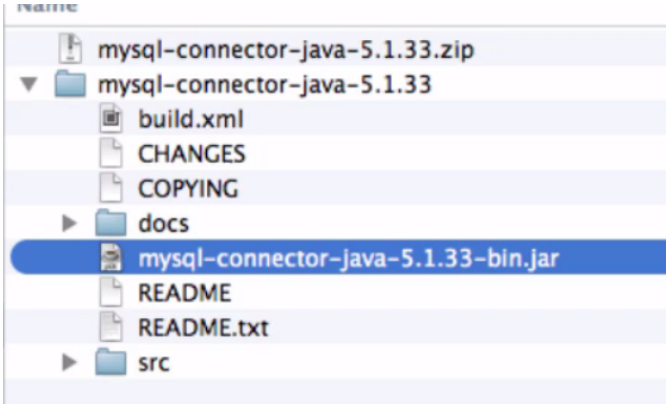
Install JDBC Driver :

- Download the JDBC driver for the database.
- Configure JDBC driver in Eclipse.

For MySQL :

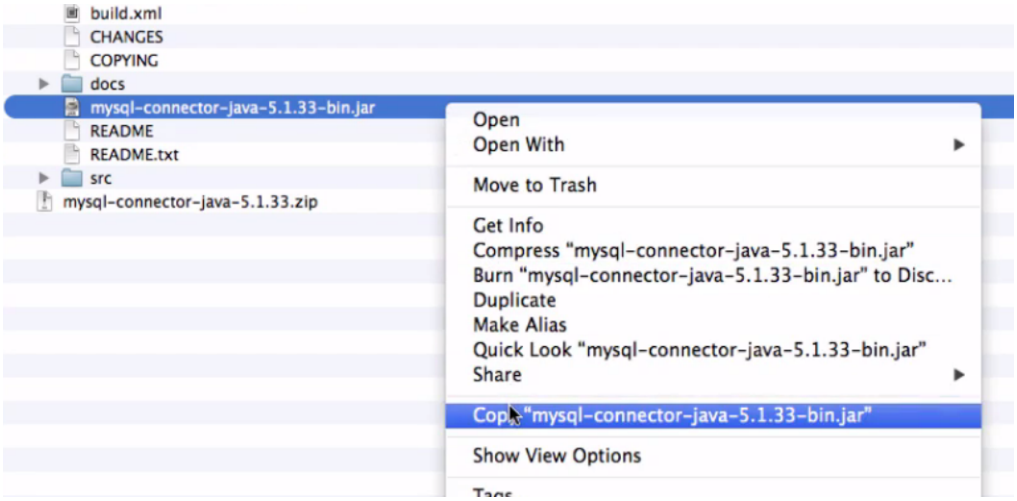
Look for Connector/J <version> --> and select platform as 'Platform Independent' --> Download the ZIP file. This ZIP is going to contain a **.jar** file which is the JDBC driver file.

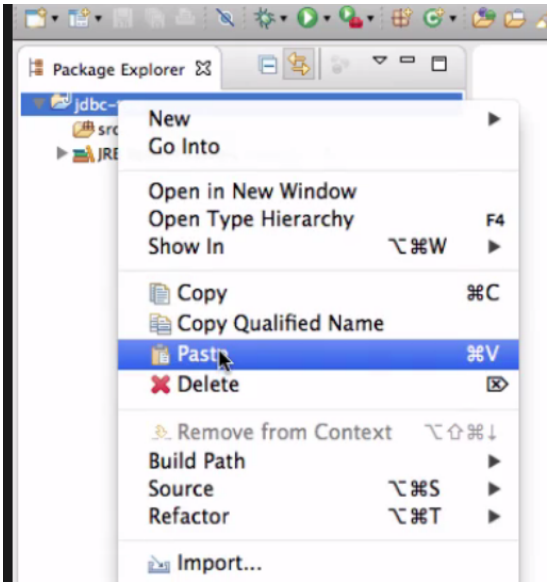




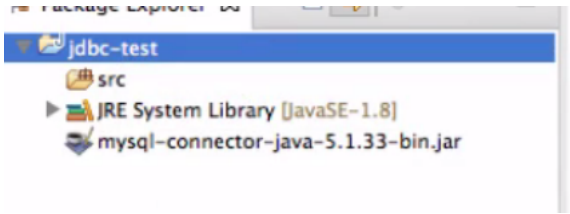
Now, **how to configure the JDBC driver in Eclipse :**

- Create a simple project in Eclipse.
- Copy the JDBC driver and paste it to the project roo folder. See below :

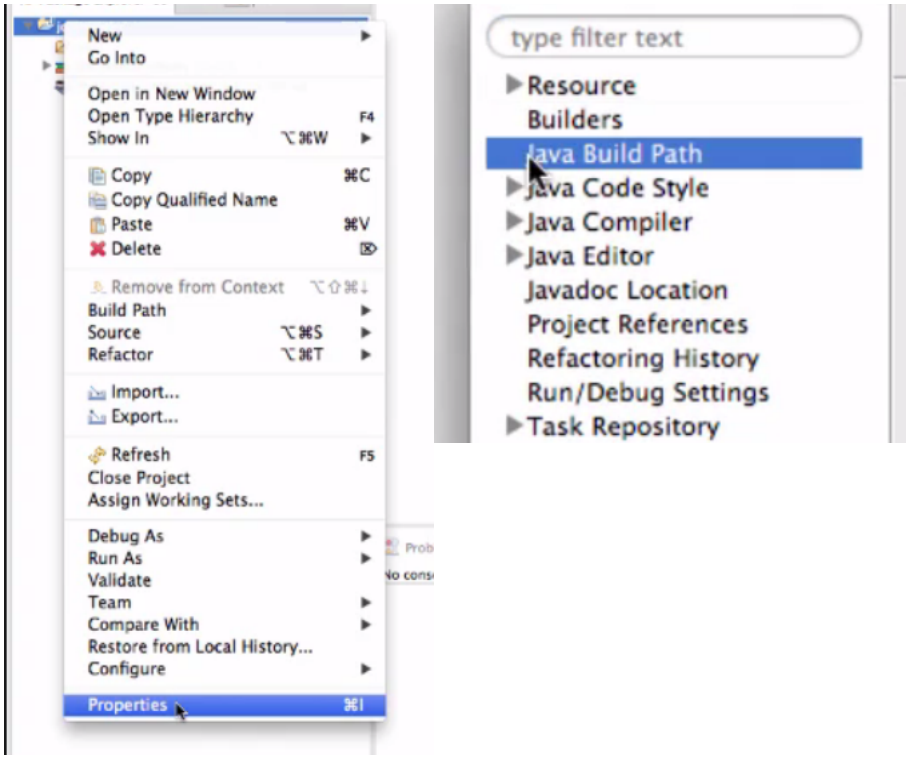


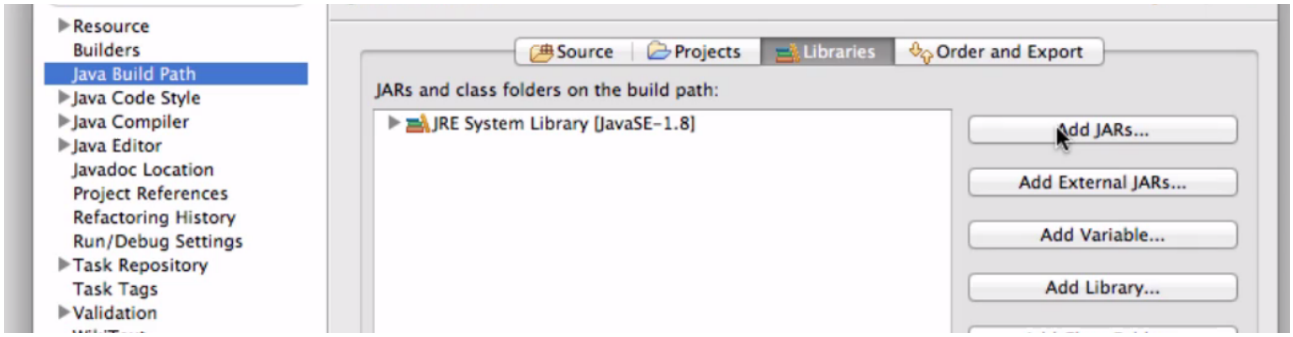


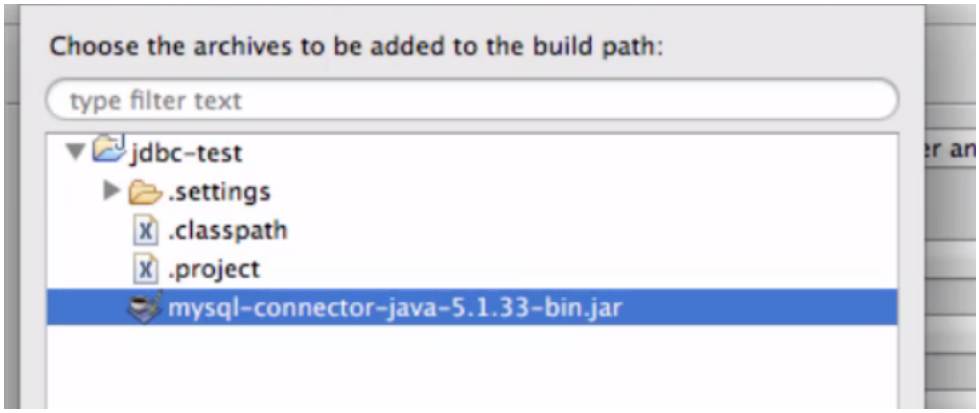
You can see the Driver file in the root of the project.

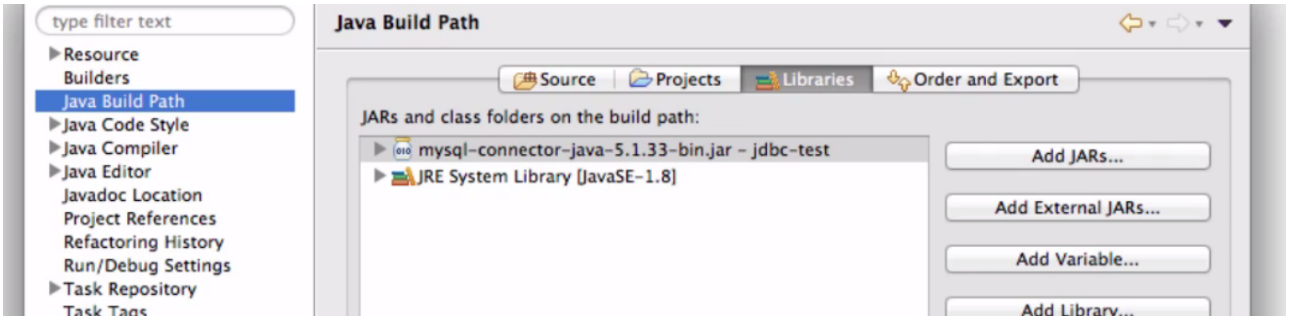


Now, we need to tell Eclipse how to associate this new .jar file with the project. Follow these steps :


New
Go Into
Open in New Window
Open Type Hierarchy F4
Show In
Copy
Copy Qualified Name
Paste
Delete
Remove from Context
Build Path
Source
Refactor
Import...
Export...
Refresh F5
Close Project
Assign Working Sets...
Debug As
Run As
Validate
Team
Compare With
Restore from Local History...
Configure
Properties
type filter text
Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Refactoring History
Run/Debug Settings
Task Repository


Resource
Builders
Java Build Path
Java Code Style
Java Compiler
Java Editor
Javadoc Location
Project References
Refactoring History
Run/Debug Settings
Task Repository
Task Tags
Validation
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
JRE System Library [JavaSE-1.8]
Add JARs...
Add External JARs...
Add Variable...
Add Library...


Choose the archives to be added to the build path:
type filter text
jdbc-test
.settings
.classpath
.project
mysql-connector-java-5.1.33-bin.jar



With this, now the Eclipse can reference to this JDBC driver file.

Simple Java program for testing JDBC Connection :

```
import java.sql.*;

public class JdbcTest {

	public static void main(String[] args) throws SQLException {

		Connection myConn = null;
		Statement myStmt = null;
		ResultSet myRs = null;

		try {
			// 1. Get a connection to database
			myConn = DriverManager.getConnection(
					"jdbc:mysql://localhost:3306/demo",
					"student" , "student");

			System.out.println("Database connection successful!\n");

			// 2. Create a statement
			myStmt = myConn.createStatement();

			// 3. Execute SQL query
			myRs = myStmt.executeQuery("select * from employees");

			// 4. Process the result set
			while (myRs.next()) {
				System.out.println(myRs.getString("last_name") +
						", " + myRs.getString("first_name"));
			}
		}
		catch (Exception exc) {
			exc.printStackTrace();
		}
		finally {
			if (myRs != null) {
				myRs.close();
			}

			if (myStmt != null) {
				myStmt.close();
			}

			if (myConn != null) {
				myConn.close();
			}
		}
	}

}
```

When you run the above program, it will run successfully, but you will get this warning message "*WARN: Establishing SSL connection without server's identity verification is not recommended* ". It is just the MySQL database yelling at you.

How to get rid of this message ?

To get rid of the warning message, append **?useSSL=false** to the end of your database connection string.

For example,

Replace – jdbc:mysql://localhost:3306/demo

With – jdbc:mysql://localhost:3306/demo**?useSSL=false**

Note that I appended **?useSSL=false** to the end.

That will get rid of the pesky message … whew!

Inserting Data into Database :

Here is regular SQL statement :

```
insert into employees
    (last_name, first_name, email, department, salary)
    values
    ('Wright', 'Eric', 'eric.wright@foo.com', 'HR', 33000.00)
```

However, inserting data in to database using JDBC call :

```
int rowsAffected = myStmt.executeUpdate(
        "insert into employees " +
        "(last_name, first_name, email, department, salary) " +
        "values " +
        "('Wright', 'Eric', 'eric.wright@foo.com', 'HR', 33000.00)");
```

executeUpdate( ) --> generic method that we can use to handle any updates to the database. So, we can use this method to perform SQL inserts, updates, delete operations.

The method will return the number of rows affected after the operation.

Updating Data in a Database :

Here is regular SQL statement :

```
update employees set   email='john.doe@luv2code.com'
                  where last_name='Doe'
                  and   first_name='John';
```

However, updating data in to database using JDBC call :

```
int rowsAffected = myStmt.executeUpdate(
        "update employees " +
        "set email='john.doe@luv2code.com' " +
        "where last_name='Doe' and first_name='John'");
```

executeUpdate( ) --> generic method that we can use to handle any updates to the database. So, we can use this method to perform SQL inserts, updates, delete operations.

The method will return the number of rows affected after the operation.

Deleting data from Database :

Here is regular SQL statement :

```
delete from  employees
       where last_name='Doe'
       and   first_name='John';
```

However, deleting data from database using JDBC call :

```
int rowsAffected = myStmt.executeUpdate(
       "delete from employees " +
       "where last_name='Doe' and first_name='John'");
```

executeUpdate( ) --> generic method that we can use to handle any updates to the database. So, we can use this method to perform SQL inserts, updates, delete operations.

The method will return the number of rows affected after the operation.

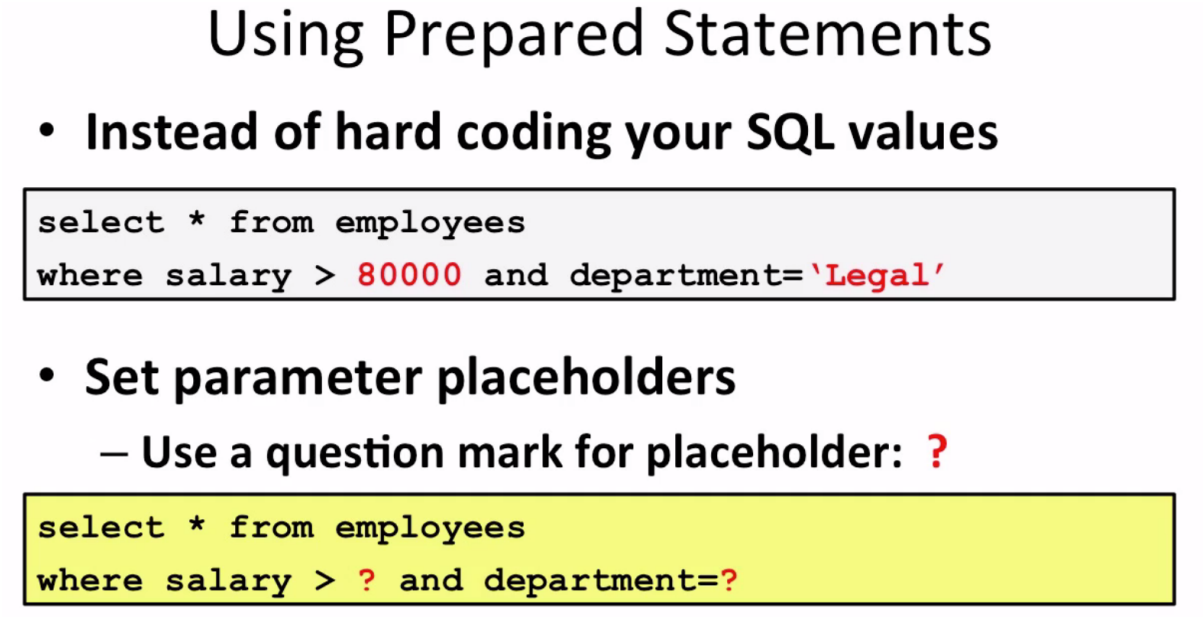
Prepared Statements – for reuse and to set parameter values:

- What are Prepared Statements?
- Create a Prepared Statement
- Setting parameter values
- Executing a Prepared statement
- Reusing a prepared statement**

Prepared Statement is a **pre-compiled** SQL statement.

- Makes it easier to set SQL parameter values.
- Prevent against SQL dependency injection attacks
- May improve application performance as SQL statement is pre-compiled.

Using Prepared Statement :



The main thing to note is, instead of using hard coding values, use placeholders using ? Symbol. Check above.

Creating Prepared Statement :

```
PreparedStatement myStmt =
   myConn.prepareStatement("select * from employees" +
                           " where salary > ? and department=?");
```

Setting parameter values & Execute query:

```
myStmt.setDouble(1, 80000);
myStmt.setString(2, "Legal");

// now execute the query
ResultSet myRs = myStmt.executeQuery();
```

Note: The parameter **positions are 1-based** starting from L -> R

We can also use Prepared Statements for Insert, Update, and Delete operations.

```
PreparedStatement myStmt =
    myConn.prepareStatement("delete from employees" +
                            " where salary > ? and department=?");
// set params
myStmt.setDouble(1, 80000);
myStmt.setString(2, "Legal");

// execute statement
int rowsAffected = myStmt.executeUpdate();
```

Stored Procedures

It is nothing but a **group of SQL statements** that perform a particular task.

You can think of it like a function in programming language.

A Stored Procedure can have any combination of input, output, and input/output parameters**.

Normally created by the DBA.

Stored procedures are created in SQL language, supported by native database. For example, MySQL has stored procedure language, Oracle uses PL/SQL and Microsoft SQL server uses T-SQL.

How to call stored procedures from Java?

JDBC API provides the **CallableStatement**.

```
CallableStatement myCall =
    myConn.prepareCall("{call some_stored_proc()}");

...

myCall.execute();
```

Observe the syntax : using curly braces and JDBC keyword, call and name of the stored procedure.

{ call <stored_procedure_name>( ) } // here we are not passing any arguments to the method.

JDBC API supports different parameters :

- IN (default)
- INOUT
- OUT

IN parameters :

Let's say, our DBA has created a stored procedure some thing like this :

```
PROCEDURE `increase_salaries_for_department`(
          IN the_department VARCHAR(64), IN increase_amount DECIMAL(10,2))

BEGIN
     UPDATE employees SET salary= salary + increase_amount
     WHERE department=the_department;
END
```

- **Increase salary for everyone in a department ☺**
  - **First param is the department name**
  - **Second param is the increase amount**

How to call this stored procedure from Java code :

You have to use CallableStatement. You can observe, we are using placeholders for parameters.

```
Callable myStmt = myConn.prepareCall(
          "{call increase_salaries_for_department(?, ?)}");
```

```
myStmt.setString(1, "Engineering");
myStmt.setDouble(2, 10000);

// now execute the statement
myStmt.execute();
```

CallableStatement with Oracle PL/SQL :

```
String lQuery = "declare b boolean; retval varchar2(1); " +
                "begin b := fnd_request.add_language(:1,:2, :3, :4); " +
                " if ( b ) then  retval := 'T'; else retval := 'F'; " +
                " end if; :5 := retval; end; " ;

CallableStatement lStmt = null;
try
{
   lStmt = mConn.prepareCall(lQuery);
   lStmt.setString(1, pLanguage);
   lStmt.setString(2, pTerritory);
   lStmt.setString(3, pNumChar);
   lStmt.setString(4, pNlsSort);
   lStmt.registerOutParameter(5, java.sql.Types.VARCHAR);
   lStmt.execute();
   String lRetVal = lStmt.getString(5);

   if( lRetVal.equals("F"))
   {
        throw new Exception(retrievePLSQLError());
   }
   lStmt.close();

} catch(SQLException e)
{
   throw new Exception(e.getMessage());
}
finally
{
   if ( lStmt != null )
      try
      {
        lStmt.close();
      } catch (SQLException e) { }
}
```

Please note, how we are registering an out parameter and retrieving value from that.

Calling Stored Procedures – INOUT parameters

```
Callable myStmt = myConn.prepareCall(
                                "{call greet_the_department(?)}");
```

```
myStmt.registerOutParameter(1, Types.VARCHAR); // use this for INOUT
myStmt.setString(1, "Engineering");

// Call stored procedure
myStmt.execute();

// Get the value of the INOUT parameter
String theResult = myStmt.getString(1);

System.out.println("The result = " + theResult);
```

"Hello to the awesome Engineering team!"

registerOutParameter( ) --> to register as OUT parameter, pass the parameter position of the placeholder in the arguments to stored procedure. Execute the stored procedure call.
Then, use getXXXX(<index>) method to get the return value.

What I observed is, in case of INOUT, we first do registerOutParameter for that parameter position and then set the parameter value to pass.
In case of OUT only parameter, we just call registerOutParameter( ), that's it.

Note : the placeholder indexes are 1-based.

Calling Stored Procedures – OUT parameters

Suppose, our DBA has created a stored procedure like this :

```
PROCEDURE `get_count_for_department`(
                              IN the_department VARCHAR(64), OUT the_count INT)
BEGIN
     SELECT COUNT(*) INTO the_count FROM employees
     WHERE department=the_department;
END
```

- Very simple OUT param example :
  - First param is the department name
  - Second param is the output for the count.

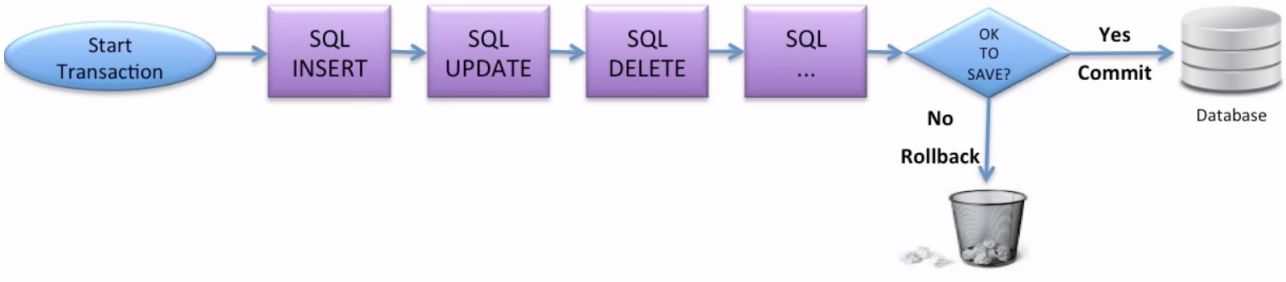
```
Callable myStmt = myConn.prepareCall(
                            "{call get_count_for_department(?, ?)}");
```

```
myStmt.setString(1, "Engineering");
myStmt.registerOutParameter(2, Types.INTEGER);

// Call stored procedure
myStmt.execute();

// Get the value of the OUT parameter
int theCount = myStmt.getInt(2);

System.out.println("The count = " + theCount);
```

We are using registerOutParameter( ) to register as OUT parameter. We also specify the type of value that it holds.

JDBC Transactions

- A transaction is a unit of work
- One or more SQL statements executed together
  - Either all of the statements are executed - Commit
  - Or none of the statements are executed - Rollback



[Default] The Database connection is **Auto-commit** true.

We need to explicitly turn off auto-commit.

- By default, the database connection is to auto-commit
  - Need to explicitly turn off auto-commit

```
myConn.setAutoCommit(false);
```

- Developer controls commit or rollback

```
myConn.commit();

// or

myConn.rollback();
```

This is how developer controls commit or rollback.

```
// start transaction
myConn.setAutoCommit(false);

// perform multiple SQL statements (insert, update, delete)
// ...

boolean ok = askUserIfOkToSave();

if (ok) {
  // store in database
  myConn.commit();
}
else {
  // discard
  myConn.rollback();
}
```

At the start of transaction, set auto-commit to false.

Accessing Database Metadata

- **Retrieve the metadata instance**

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

- **DatabaseMetaData methods**
  - `getDatabaseProductName()`
  - `getDatabaseProductVersion()`
  - `getDriverName()`
  - etc...

- See JavaDoc for details
  - Google: **jdbc databasemetadata**

We can also use DatabaseMetaData to get information about database schema.

We can list of tables.

We can list of column names.

```
public static void main(String[] args) throws SQLException {

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

    Connection myConn = null;
    ResultSet myRs = null;

    try {
        // 1. Get a connection to database
        myConn = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/demo", "student", "student");

        // 2. Get metadata
        DatabaseMetaData databaseMetaData = myConn.getMetaData();

        // 3. Get list of tables
        System.out.println("List of Tables");
        System.out.println("--------------");

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

        while (myRs.next()) {
            System.out.println(myRs.getString("TABLE_NAME"));
        }
```

To get list of column names :

```
// 4. Get list of columns
System.out.println("\n\nList of Columns");
System.out.println("---------------");

myRs = databaseMetaData.getColumns(catalog, schemaPattern, "employees", columnNamePattern);

while (myRs.next()) {
    System.out.println(myRs.getString("COLUMN_NAME"));
}
```

Reading ResultSet Metadata

- **Retrieve the metadata instance**

```
ResultSetMetaData myRsMetaData = myRs.getMetaData();
```

- **Sample of ResultSetMetaData methods**

getColumnName()	getColumnType()	getColumnTypeName()
getPrecision()	getScale()	getColumnCount()
isAutoIncrement()	isNullable()	isCurrency()
...	...	...

- See JavaDoc for details
  - Google: **javadoc jdbc resultset metadata**

Here, we are calling getMetaData( ) method on the result set, note here we are not calling on connection object.

```
// 3. Get result set metadata
ResultSetMetaData rsMetaData = myRs.getMetaData();

// 4. Display info
int columnCount = rsMetaData.getColumnCount();
System.out.println("Column count: " + columnCount + "\n");

for (int column=1; column <= columnCount; column++) {
    System.out.println("Column name: " + rsMetaData.getColumnName(column));
    System.out.println("Column type name: " + rsMetaData.getColumnTypeName(column));
    System.out.println("Is Nullable: " + rsMetaData.isNullable(column));
    System.out.println("Is Auto Increment: " + rsMetaData.isAutoIncrement(column) + "\n");
}
```

Notice, in JDBC columns are 1-based.

Reading and Writing BLOBs

- A **BLOB** (binary large object) is a collection of binary data stored as a single entity in a database.
- BLOBs are typically documents, images, audio or other binary objects.
- *Note: Database support for BLOBs is not universal.*

BLOBs are typically used for storing images, audio files, pdf files, text documents or other binary objects.

Note: Not all databases have support for BLOBs.

#create blob in table

- **When creating a table in MySQL**
  - **Add a column with BLOB datatype**

File: table-setup.sql

```
CREATE TABLE `employees` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `last_name` varchar(64) DEFAULT NULL,
  `first_name` varchar(64) DEFAULT NULL,
  `email` varchar(64) DEFAULT NULL,
  `department` varchar(64) DEFAULT NULL,
  `salary` DECIMAL(10,2) DEFAULT NULL,
  `resume` BLOB,

  PRIMARY KEY (`id`)
)
```

Add column with BLOB datatype

Writing BLOBs : #write blob

- **Add a resume for an employee**
  - **Read local PDF file: sample_resume.pdf**
  - **Update database with the binary data**

```
// Prepare statement
String sql = "update employees set resume=?"
               + " where email='john.doe@foo.com'";

PreparedStatement myStmt = myConn.prepareStatement(sql);

// Set parameter for resume file name
File theFile = new File("sample_resume.pdf");
FileInputStream input = new FileInputStream(theFile);

myStmt.setBinaryStream(1, input);

// Execute statement
myStmt.executeUpdate();
```

Handle for local file

Update DB with binary data

Reading BLOBs: #read blob

- **Read BLOB from DB and write to local file**

```
Statement myStmt = myConn.createStatement();
String sql = "select resume from employees "
               + " where email='john.doe@foo.com'";

// Execute query
myRs = myStmt.executeQuery(sql);

// Set up a handle to the output file
File theFile = new File("resume_from_db.pdf");
FileOutputStream output = new FileOutputStream(theFile);

// read BLOB and store in output file
if (myRs.next()) {
     InputStream input = myRs.getBinaryStream("resume");

     byte[] buffer = new byte[1024];
     while (input.read(buffer) > 0) {
          output.write(buffer);
     }
}
```

Read BLOB

Write to local file

Reading and Writing CLOBs

- A **CLOB** (character large object) is a collection of character data stored as a single entity in a database.
- CLOBs are typically used to store large text documents (plain text or XML)
- *Note: Database support for CLOBs is not universal.*

CLOBs are typically used for storing large text documents, plain text documents or **XML files**.

How to create CLOB column in MySQL : #create clob column

- **When creating a table in MySQL**
  - **Add a column with LONGTEXT datatype**
  - **Hold a maximum length of 4GB of characters**

File: table-setup.sql

```
CREATE TABLE `employees` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `last_name` varchar(64) DEFAULT NULL,
  `first_name` varchar(64) DEFAULT NULL,
  `email` varchar(64) DEFAULT NULL,
  `department` varchar(64) DEFAULT NULL,
  `salary` DECIMAL(10,2) DEFAULT NULL,
  `resume` LONGTEXT,

  PRIMARY KEY (`id`)
)
```

Add column with LONGTEXT datatype

We are using LONGTEXT data type.

Writing CLOBs to Database : #write clob

- **Add a resume for an employee**
  - **Read local text file: sample_resume.txt**
  - **Update database with the text data**

```
// Prepare statement
String sql = "update employees set resume=?"
               + " where email='john.doe@foo.com'";

PreparedStatement myStmt = myConn.prepareStatement(sql);

// Set parameter for resume file name
File theFile = new File("sample_resume.txt");
FileReader input = new FileReader(theFile);

myStmt.setCharacterStream(1, input);

// Execute statement
myStmt.executeUpdate();
```

Handle for local file

Update DB with CLOB data

Reading CLOBs from Databsae : #read clob

- **Read CLOB from DB and write to local file**

```
Statement myStmt = myConn.createStatement();
String sql = "select resume from employees "
                + " where email='john.doe@foo.com'";

// Execute query
myRs = myStmt.executeQuery(sql);

// Set up a handle to the output file
File theFile = new File("resume_from_db.txt");
FileWriter output = new FileOutputStream(theFile);

// read CLOB and store in output file
if (myRs.next()) {
     Reader input = myRs.getCharacterStream("resume");

     int theChar;
     while ((theChar = input.read()) > 0) {
          output.write(theChar);
     }
}
```

Read CLOB

Write to local file

Reading Database Connection info from Properties File

Ideally, it is recommended to use a properties file to configure database connection information rather than hardcoding the values within the code.

We are going to use **java.util.Properties** class.
Within the properties file, we have key-value pairs.

- Java has a class `java.util.Properties`
  - It can read and write a configuration file

File: demo.properties

```
dburl=jdbc:mysql://localhost:3306/demo
user=student
password=student
```

- Read the properties file