|  |  |  |  |
| --- | --- | --- | --- |
| **Student Name** |  | **Section** |  |

**PROJECT** - **Java Database (jdbc) 50 points**

**Objective** To create a database, insert records into it then retrieve data from it for evaluation.

***PROJECT DESCRIPTION***

You will create an inventory database to store inventory items (or record data) into a table. Then you will retrieve the records and evaluate the inventory items in a Last In First Out or LIFO manner to determine the Cost of Goods Sold for the particular company and Gross Profit Margin.

***Information about This Project***

This program illustrates an example of Java database processing and a stack based class.

***Steps To Complete This Project***

**STEP 1**  **Start Eclipse**

Create a Java Project in Eclipse and name it **Inventory**.

**STEP 2**  **Add a class for jdbc processing**

Create a class called **JDBC** in your project. You will add in code shortly to create

a database table within an existing database called Inventory located on the papanet server.

**STEP 3**  **Download and install mysql driver**

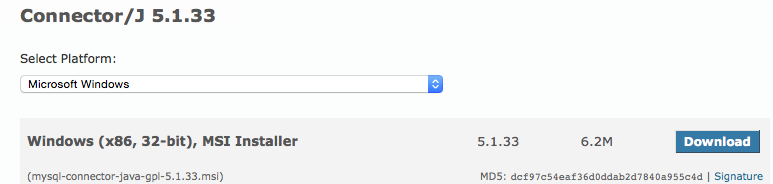
Go to the following website to download and install the jdbc windows driver. This driver is necessary to allow you to connect a mysql database off a server.

<http://dev.mysql.com/downloads/connector/j/>

Choose **Microsoft Windows** under the dropbox where it suggest you to Select a Platform (shown below). If you are a **Mac** user, choose from the drop down menu

**Platform Independent**.

Click Download at this point.



The site will take you to a page to create a free Oracle Web Account. You can bypass this by clicking the link at the bottom of the page where it says

No thanks, just start my download**.**

(Windows users): At this point save the **mysql-connector-java-gpl-5.1.33.msi** file to your download folder. When the download is complete go to your download folder and double click onto your file to install the **msi** file. The msi installation places a jar file- **mysql-connector-java-5.1.33-bin.jar** (you’ll need to add this to your project momentarily, in a lib folder for your project) located at the following path:

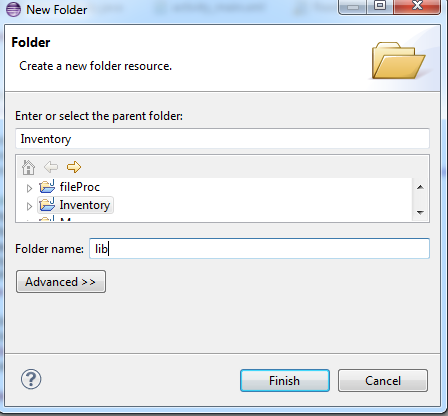
**C:\Program Files (x86)\MySQL\Connector J**

(Mac users)- you need to install the driver (.jar) file as well and place it into your lib folder as

per the specs above. At this point choose to download the **.tar** file or **.zip** file. Choose either one as both should open on the latest Mac platform. When the folder either untars or unzips depending on what you choose as a file type to download, you should see the needed

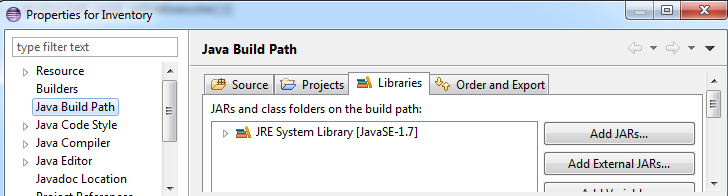
**mysql-connector-java-5.1.33-bin.jar** file for jdbc connectivity.

(All platforms): Right click on your **mysql-connector-java-5.1.33-bin.jar** file and choose Copy. Then go to your to your Inventory project in Eclipse and right click on the Inventory root folder and choose New > Folder. Type in **lib** in the Folder name: area as shown below. Click Finish when complete.

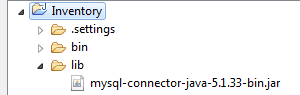


Now right click on your lib folder and choose Paste. You will now notice your .jar file has been copied in.

Now to officially *add* this jar to your build path so its recognized at compile time, right click on your Inventory root folder and choose Properties. On the left hand side click on Java Build Path. Then click on the Libraries tab at the top center area then click on Add JARS... as shown below...



At the JAR Selection dialog, drill down your Inventory folder to your lib folder and click on your mysql-connector-java-5.1.33-bin.jar file. Click OK at this point. Click OK as well again to close out of your Properties dialog box.



Now onto the next step for some coding.

**STEP 4**  **Open up your JDBC class file**

In your JDBC code enter the following code in to create a database table:

**import** java.sql.Connection;

**import** java.sql.DriverManager;

**import** java.sql.Statement;

**public** **class** JDBC {

**private** Connection connect = **null**;

**private** Statement statement = **null**;

**public** **void** createDataBase() **throws** Exception {

**try** {

// This will load the MySQL driver, each DB has its own driver

Class.*forName*("com.mysql.jdbc.Driver");

// Setup the connection with the DB

connect = DriverManager

.*getConnection*("jdbc:mysql://www.papademas.net/Inventory?"

+ "user=root&password=jamesp");

//create table

statement = connect.createStatement();

String sql = "CREATE TABLE jpapaInventory " +

"(id INTEGER not NULL AUTO\_INCREMENT, " +

" cost INTEGER, " +

" PRIMARY KEY ( id ))";

statement.executeUpdate(sql);

System.*out*.println("Created table in given database...");

//end create table

} **catch** (Exception e) {

System.*out*.println(e.getMessage());

}

}

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

JDBC dao = **new** JDBC();

dao.createDataBase();

}

}

NOTE- PLEASE REPLACE IN THE CODE ABOVE, THE TABLE NAME. INCLUDE *YOUR* **FIRST INTIAL**

**+ YOUR FIRST 4 LETTERS OF YOUR LAST NAME + INVENTORY** AS *YOUR* TABLE NAME.

NOTICE MY EXAMPLE ABOVE WAS jpapaInventory. PLEASE CHANGE THIS TO YOUR

INFORMATION BEFORE RUNNING YOUR APP INITIALLY!

**STEP 5**  **Run your initial application.**

Run your program and see if your table was created. If it was proceed to the next step otherwise correct your syntax/sql statements for any apparent errors. Check also if any firewalls maybe blocking you when you are creating the table if so as a thought.

Snapshot your output window when complete.

**STEP 6**  **Insert records into your table.**

After successfully creating your table, you can now **comment out** the line in main that calls your createDataBase() method.

Create the following method, insertIntoDataBase(), in your class to allow for inserts to be added to your database. Starter code is as follows:

**public** **void** insertIntoDataBase() **throws** Exception {

**try** {

// This will load the MySQL driver, each DB has its own driver

Class.*forName*("com.mysql.jdbc.Driver");

// Setup the connection with the DB

connect = DriverManager

.*getConnection*("jdbc:mysql://www.papademas.net/Inventory?"

+ "user=root&password=jamesp");

System.*out*.println("Inserting records into the table...");

statement = connect.createStatement();

String sql = "INSERT INTO jpapaInventory(cost) " +

"VALUES (400)";

statement.executeUpdate(sql);

sql = "INSERT INTO jpapaInventory(cost) " +

"VALUES (400)";

statement.executeUpdate(sql);

System.*out*.println("Inserted records into the table...");

} **catch** (Exception e) {

System.*out*.println(e.getMessage());

}

}

\*\*OF COURSE, ONCE AGAIN, REPLACE MY EXAMPLE TABLE NAME IN THE INSERT SQL STATEMENTS WITH YOUR TABLE NAME.

In a similar manner as above, go back into your code and also add in three more records for the cost field, in the amounts of 500, 500 and 600 respectively within your method.

To finish with your class file, add in a call to your new method in main via your dao object created in main, save your work and run your application. You should be seeing that the records have been inserted successfully in your Console window. If not please check your SQL statements carefully for any errors.

Additional help on sql statements, for selects, updates, inserts, deletes, etc. can be found at this rockin’ site:

[http://www.tutalspoint.com/jdbc](http://www.tutorialspoint.com/jdbc)

Snapshot your output window when complete.

Now onto the next step to process your inventory items in a new class file.

**STEP 7**  **Create class file to process inventory items by cost.**

Create a new class called **LIFO**. This class will read from the database and determine the profit and value of the inventory based on the LIFO method.

For a company’s accounting and financial & tax statements, inventory items can be evaluated and represented with either the LIFO or FIFO valuation method.

Consider the following accounting scenario comparing LIFO and FIFO which will serve as the logic for your **LIFO** class...

Your company bought 5 identical computers during the year at prices: $400, $400, $500, $500, $600; and then they sold 3 of them for $800 each. What is the profit? What is the value of the inventory? The answer depends on *which* computers were sold. Although the computers are physically identical, their accounting values are different. The following are different calculations using FIFO and LIFO accounting.

|  |  |  |
| --- | --- | --- |
|  | **LIFO** | FIFO |
| Costs of Goods Sold | **600 + 500 + 500 = 1600** | 400 + 400 + 500 = 1300 |
| Profit | **2400 – 1600 = 800** | 2400 – 1300 = 1100 |
| Ending Inventory | **400 + 400 = 800** | 500 + 600 = 1100 |

Notice under the LIFO method, the last items in (3 in this case) are evaluated first given 3 items sold. The FIFO method evaluates the first 3 items in. Notice how Cost of Goods Sold (COGS), Profit, and Ending Inventory are calculated under each method. You can see that under the FIFO method more profit is generated. Can you guess why? Note that although FIFO generates more profit, there will be more profit subject to tax.

Add in the following starter code to your LIFO class:

**import** java.sql.DriverManager;

**import** java.sql.ResultSet;

**import** java.util.Arrays;

**import** com.mysql.jdbc.Connection;

**import** com.mysql.jdbc.Statement;

**public** **class** LIFO <T>

{

**private** **static** Connection *connect* = **null**;

**private** **static** Statement *statement* = **null**;

**private** **static** ResultSet *resultSet* = **null**;

**private** **static** **int** *count*;

**private** T[] data;

**public** LIFO()

{

data = (T[]) **new** Object[5];

*count* = 0;

}

**void** expandCapacity()

{

data = Arrays.*copyOf*(data, data.length \* 2);

}

**void** push(T e)

{

**if** (*count* == data.length)

expandCapacity();

data[*count*++] = e;

}

T pop() **throws** Exception

{

**if** (*count* <= 0)

{

**throw** **new** Exception("stack empty");

}

*count*--;

**return** data[*count*];

}

**boolean** isEmpty()

{

**return** *count* == 0;

}

**static** **int** size()

{

**return** *count*;

}

**public** **static** **void** main(String[] args) **throws** Exception

{

LIFO<Integer> s = **new** LIFO<Integer>();

**try** {

// This will load the MySQL driver, each DB has its own driver

Class.*forName*("com.mysql.jdbc.Driver");

// Setup the connection with the DB

*connect* = (Connection) DriverManager

.*getConnection*("jdbc:mysql://www.papademas.net/Inventory?"

+ "user=root&password=jamesp");

// Statements allow to issue SQL queries to the database

*statement* = (Statement) *connect*.createStatement();

// Result set gets the result of the SQL query

*resultSet* = *statement*

.executeQuery("select cost from jpapaInventory");

// ResultSet is initially before the first data set

**while** (*resultSet*.next()) {

/\* column data may be retrieved via name

e.g. resultSet.getString("name");

or via the column number which starts at 1

e.g. resultSet.getString(1); \*/

**int** cost = *resultSet*.getInt(1); //retrieve cost

s.push(cost); //push cost value onto stack

System.*out*.println("Push item : " + s.*count* + " " + cost);

}

**while** (!s.isEmpty())//pop values from stack

System.*out*.println("Pop iem : " + s.*count* + " " + s.pop());

} **catch** (Exception e) {

System.*out*.println(e.getMessage());

}

}

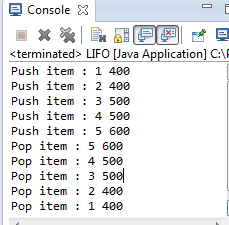
}

\*\*PLEASE MAKE SURE TO CHANGE THE TABLE NAME AGAIN TO YOUR TABLE NAME IN THE SELECT QUERY STATEMENT (SHOWN BELOW)

*resultSet* = *statement*

.executeQuery("select cost from jpapaInventory");

Run your LIFO application and verify your results against a sample running application below.



Notice how the stack process works here for our LIFO class from the snapshot above which really represents a stack’s logic namely to push values onto a stack one by one, but when we get or pop values from the stack the last items in are actually the first popped. This is perfect for our inventory valuation for the LIFO method!

So you see when we call the push() method in our code we are adding onto the class stack (stored into the array variable data which can handle any type thus a generic type T is given (*array declaration shown below*) for each record value we retrieve from the *resultSet* query.

**private** T[] data;

Then after the loop ends we call the pop() method to view each item added into the class stack.

Notice the outcomes or order how the cost items are pushed in the stack then popped or reversed when displayed. We got it right!

Okay –please study over the code mechanics! This is a beautiful, *classic* example of a stack. Especially notice the push and pop methods. Each serve an obvious purpose. Push takes in a value (a *setter* method really) and pop returns a given value (a *getter* method). Notice that a *count* variable MUST be included to keep track of the array’s elements position to determine where the values should be pushed onto the stack ifpossible and where they are to be popped from the stack if possible. Further notice the creation of a generic object type in main namely for the object s as shown below,

LIFO<Integer> s = **new** LIFO<Integer>();

Notice any feasible reference type can be used to grab data for the class. In our example we just will use the Integer type to retrieve the cost value to store from our table. Of course we didn’t have to use generics for this exercise, but a working knowledge of generics is very useful for code efficiency, thus we have incorporated this into our example!

Take a snapshot of the Console window to show the stacks’ pushed and popped values thus far as a checkpoint.

Okay now onto our last steps to modify a few things!

**STEP 8**  **Modify your LIFO class to properly evaluate your inventory items.**

Okay now you have printed out your stack, you can comment out those lines as you know how things work and that you were successful in grabbing data then displaying it for each cost item added to the class stack.

Go back to your LIFO class code and include any additional variables and calculations to

determine the following information for your company:

1. Determine the Goods Sold (COGS) under the LIFO method
2. Determine the Profit under the LIFO method
3. Determine the Ending Inventory under the LIFO method
4. Determine also the Inventory Turnover (which is a ratio) for the company which helps measure the number of times that the company sold its inventory during the year.

Note the Inventory Turnover formula is as follows:

Inventory Turnover = COGS/Average Inventory

where

Average Inventory = (Beginning Inventory amounts + Ending Inventory amounts)/2

Read over the scenario given again in step 7 to refresh your thinking on how to create your calculations. Remember the company orignally purchased 2400 in merchandise items (serves as your Begnning Inventory), but sold x amount of those items, which depending on the inventory evaluation method chosen, leads us to a different COGS scenario under LIFO vs. FIFO, which of course leads us to a different profit and thus ending inventory amounts.

Run your app and test your results and verify that your are displaying output for each calculation mentioned above for COGS, Profit, Ending Inventory and the Inventory Turnover ratio.

Take a screen snapshot of your results produced.

**STEP 9**  **Submission Requirements the Application**

Submit for full credit all your screen snapshots as well as your completed program code for each java file listed for this lab.