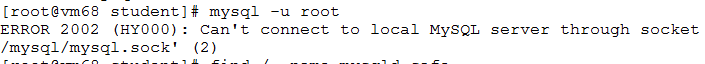
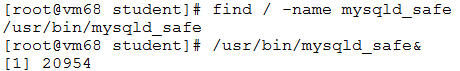
Database Programming with JDBC

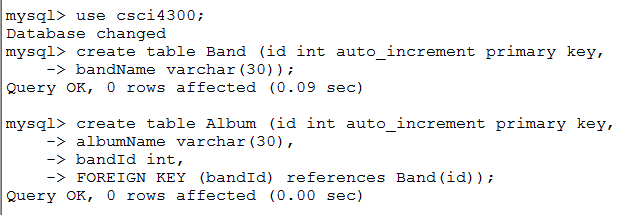
# CSCI 4300

Before you start: We will be running Java code in Eclipse that attempts to access the MySQL database server on your VM. So, you need to be able to log into your VM.

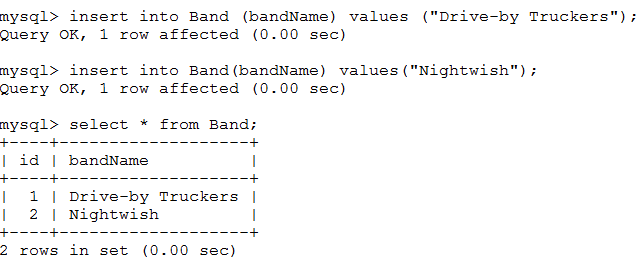
1. Log on to your VM and try to connect to the mysql database, using a user name of root and no password:

Yikes! This is an inauspicious way to start the day! This particular error message means not that you have the wrong password, but that the server is not running at all. I wanted you to see it once.

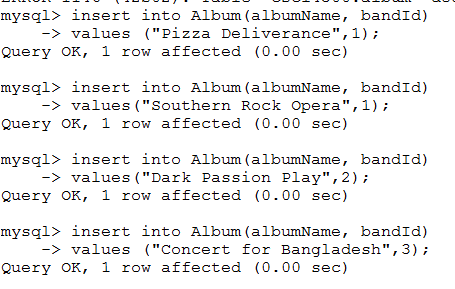
1. OK, how do we start the server? Fortunately, we have the magic power of root at our command! Temporarily become root, via the su command. Your root password is the same as the original “student” account password before you changed it. Please do not change the root password, as the TA and instructor need it to grade you work and help you when you get stuck.
2. Using the power of root, find the mysqld\_safe[[1]](#footnote-1) server daemon and start it in the background:
3. The “&” ampersand character tells Unix to start a new process, separate from the one you are using to run your command prompt, to run the mysqld\_safe daemon.
4. Now type the exit command to quit being root! In general, we want to resist the temptation to retain the godlike powers of root at all times; they can also get you into trouble. As a regular user, you are much less likely to do something to your system that we cannot undo!
5. Even though you are not running Unix commands as root, you can still run MySQL as the root user. Launch the **mysql** command line client as shown here. The temporary root password is ***mysql.***
6. Open a Web browser to the mySQL command reference at <http://dev.mysql.com/doc/refman/5.1/en/>. You will be using this frequently, so bookmark it!
7. It is recommended to reset the root password, following the instructions at <https://dev.mysql.com/doc/refman/5.5/en/set-password.html>. PLEASE write down your new root password!! [[2]](#footnote-2)
8. You should now be looking at the mysql command prompt. Enter the command **create database cs4300**. Normally we create a new database for each project, but we can consider all the lab exercises and assignments in this course to be part of the csci4300 project. Notice that mysql commands end with a semicolon, so that you can string a long command out across multiple lines. You will almost certainly forget this a few times!
9. Check out the mysql data types at <https://dev.mysql.com/doc/refman/5.5/en/data-types.html>, sections 11.2 and 11.4. Basically, “int” means “int” and “varchar” means “String” for now. Later on we’ll learn how to use the MySQL date and time types.
10. Enter the following commands to create the two tables we will use in this exercise.[[3]](#footnote-3)

 In this simple data model, there is a many-to-one relationship between bands and albums – each album belongs to a band, while one band may have many albums. The FOREIGN KEY constraint assures that an Album record must be linked to an existing record in the Band table. Each table is given an id field, whose value will be filled in by MySQL in increasing order 1, 2, 3, …. Because we may create and delete records, you should not assume that the keys in the table have values 1, 2, 3, … at any given time.

1. Let’s try entering a few bands and albums by hand. Feel free to replace these with your own favorite bands and albums:

Here you can see how the auto-increment feature works! The term PRIMARY KEY means that no two records can have the same value for this field.

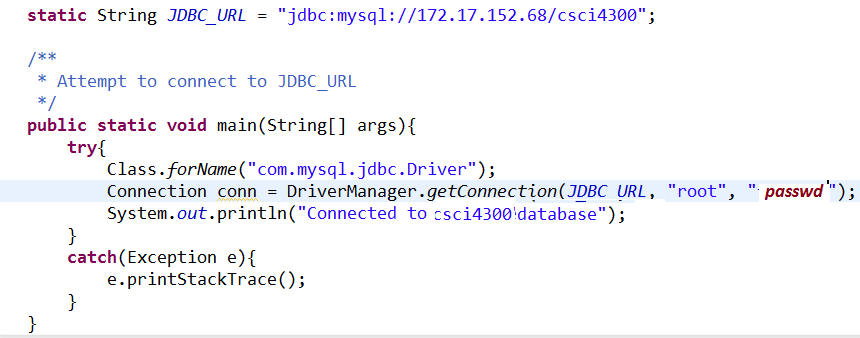
1. Now let’s add some albums to the bands. Note the deliberate error in the last insert:



We say that the last insertion violates ***relational integrity***. We’d like to set up the server to flag requests like this; we’ll show how to do this later on. For now on, we must watch out for relational integrity for ourselves.

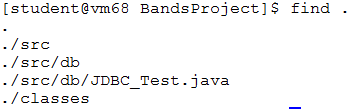
Now, let’s see if we can connect to this database through Eclipse. We assume you are working this exercise from within the UGA campus network; otherwise, you will need a VPN tunnel.

1. Launch Eclipse and create a new Dynamic Web project named **JDBC Practice**.
2. First write a class **JDBC\_Test** in package **db**, to verify that we can connect to the remote database. Here is the code for the main method of this class. Please change “68” to your own VM number.



1. Run this class as a Java application, not on the server! I expect that you will get the runtime error message shown here, which means that the Diver class is not part of the classpath in Eclipse.
2. Download the Connector/J **file mysql-connector-java-5.1.22-bin.jar** from eLC into your Downloads folder.
3. In Project Explorer, right-click the JDBC Practice node and choose Properties/Build Path. Click the  button and add mysql Jar file. Now run the JDBC\_Test class again. This time you will see a message like this:

Surprise! We have disabled remote access to the MySql databases, because it is a huge security hole, even among a mutually trusting developer communtiy such as ours. To proceed, we will need to build on your VM. I personally recommend writing and compiling on Eclipse, then uploading to your VM to actually run it.

1. First, let’s upload the JDBC\_Test.java file to your VM. *Note*: If you are using WinSSH, it helps to directly connect to host 172.17.152.xx instead of having to double-hop all your files from your local machine to nike to your VM. You may as well create an ANT formatted project directory while you are at it. Let’s call this project BandsProject. Here is a screenshot of the results of running the “find .” command from within the BandsProject directory:
2. Let’s try using the command line, instead of ANT build files, to compile and run this test file. Still in the BandsProject directory, enter the command

**javac –d classes src/db/\***

1. –the “-d” option specifies the destination of the generated class files. You should get no error messages, but no congratulations either – that’s the way the command line operates. To see whether anything happened, use the “find .” command again.
2. Now let’s run the generated class:

**java db.JDBC\_TEST**

1.  Notice how we prefix the class name with the package name, which is standard practice. We get the following error. Think about it: how would java know where we’ve stuck the class file?
2. Time for Dr. Classpath to fix us up! Try it again with the –cp option:

**java –cp classes db.JDBC\_TEST**

– Note that we specify the directory “classes” rather than “classes/db,” as java translates packages to directories.

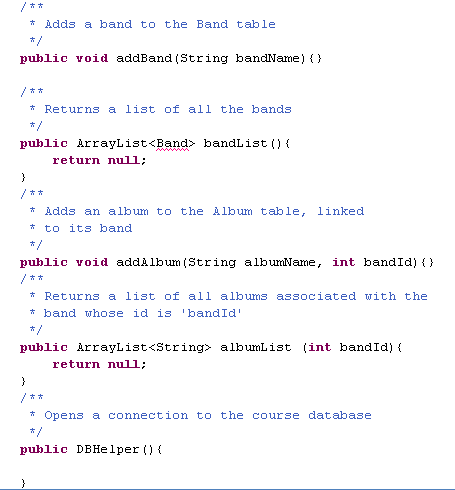
1. Oops! Now java can’t find the driver class. Upload the Connector/J jar file to its new home, which is in the Tomcat/lib directory. (To do this, you will need to upload the jar file to your home directory, then temporarily become root to move the file to /opt/tomcat/lib.) This will make the Driver class available to Tomcat when we run Web apps. There’s no reason to keep duplicate copies of the jar file all over your system.
2. Java uses the semicolon as the separator for classpath entries, so your command now becomes

**java –cp classes:/opt/tomcat/lib/mysql-connector-java-5.1.22-bin.jar db.JDBC\_TEST**

1. -- Is this fun or what? Moreover, we now get the same evil error about not being able to connect from “172.17.152.xx”, which is the very same machine.
2. Almost done, really! The problem is that mysql is too brain-dead to recognize a local request when it sees one. Edit JDBC\_Test.java to specify a “localhost” access, as shown here. You can either use **vi** or **emacs** to edit the code directly on your VM, or edit on Eclipse and upload. I encourage you to learn the Unix command line editors if you ever want to be taken seriously as a Unix wonk ☺
3. Now, from the command line, use your up-arrow key to return to the javac command and re-run it. Then, use the same trick to re-run the java command. It should work this time. Yippee!
4. This is a good time to take a break and reflect on what we’ve learned. For me the key lessons are about JDBC access and classpath issues:
   1. MySQL only allows access from “localhost,” so all our code to access this database must run on the VM itself;
   2. The javac compile step did not require the Connector/J jar file, but the run step did;
   3. Classpath entries are either directories, like “classes”, or Jar files, like “/opt/tomcat/lib/mysql-connector-java-5.1.22.-bin.jar.” We can’t specify a directory containing jar files;
   4. Typing these long path names is a pain in the rear.

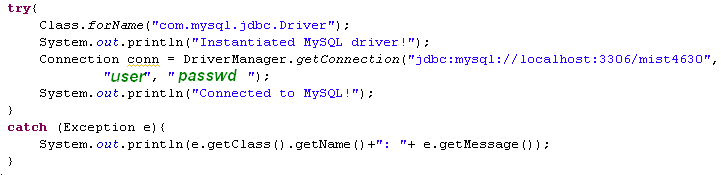
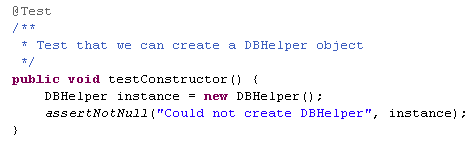
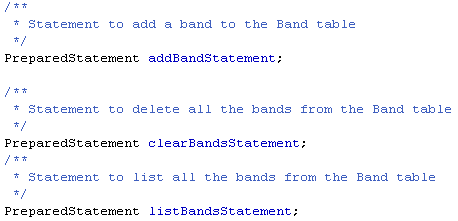
We can fix the last problem easily enough: by scripting our build process using ANT.

1. In Eclipse, create the **db.DBHelper** class, which will access the database for us. Start out by defining the basic methods specified on the next page. Please add comments to this file to tell the authors of the file.

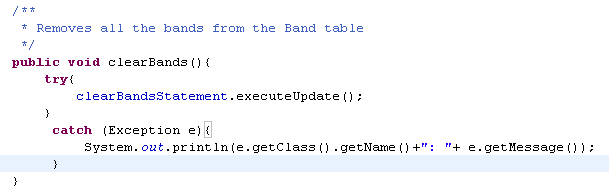


1. You will need to import java.util.ArrayList, which we commonly use to represent lists of stuff. These lists will be displayed in a JSP page later on.
2. Note that the identifier **Band** is also unknown. A Band object should contain the band name and its id, which will be needed in the JSP.

Create and define a **Band** class in package **db**. **Band** should have a constructor and two getter methods. As usual, its two data members should be **private**, so that the data can only be accessed through the getter methods. After creating the Band class, your DBHelper class should compile.

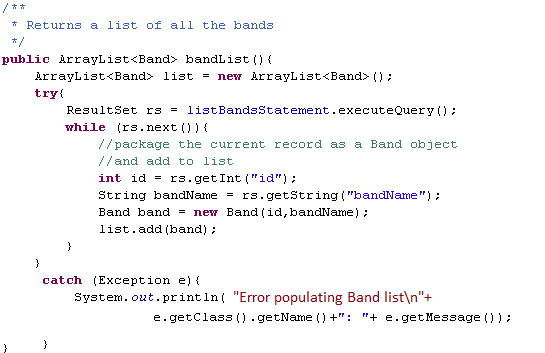
1. Create a JUnit test file for the Band class. (Remember to add the Junit 4 library to your project.) It should have a single test, testConstructor. Inside this test method, create a couple of Band objects and use the getters to verify that the objects have the correct value. Hopefully this testing process is becoming more routine by now.
2. Now it’s time to test the DBHelper class into existence. Our first step is to write the constructor, which will connect to the database. Here’s the code, which will become boilerplate for all your JDBC projects in this course: [[4]](#footnote-4)
3.  In the JUnit test code, we will simply test that the DBHelper object can be created without any Exceptions. Here is the test code:
4. Run the test code on Eclipse. It will not work, for the reasons we mentioned before. To make it work, you must now upload to your VM and run in there. You will have classpath issues again, which you will need to address by putting appropriate <classpath> elements inside your build.xml file. Recall that the compile step does not need the Connector/J jar file, but running the code does.
5. Hooray! Now it is time to start creating PreparedStatements to do the actual work. Add the following variable declarations to the top of the file:
6. To actually create the Prepared Statements, we need the Connection to the database. Add the following code to the constructor:. Note that adding a band requires one parameter, the band name. The other two statements do not require parameters.



1. Re-run the Junit test to make sure nothing blows up in our faces.
2. Add a void clearBands() method to the DBHelper class. This method should delete all the bands. It will be used for testing purposes.[[5]](#footnote-5) Here is the code:

In general, we use the executeUpdate method when the PreparedStatement does not return any records, as in this case.

1. Here is the code for the listBands method. Make sure you understand it, as you will be using this pattern a lot!



Now I am going to set you off on your own. Here is the plan I want you to follow:

1. Write a Junit method testBands, which does the following:
   1. Creates an instance;
   2. Clears the table of bands;
   3. Retrieves the list of bands, and asserts that it is empty;
   4. Adds a band to the list;
   5. Retrieves the list of bands, and asserts that it has length 1, and that the name of the band is what you put in;
   6. Adds a band to the list;
   7. Retrieves the list of bands, and asserts that it has length 2, and that the names of the bands are what you put in.

Do NOT test id values, as these will change as we add and delete records from the database. To implement addBand, you will need the PreparedStatement.setString method. Index numbers start with 1.

1. When this works, follow the same process for adding an album to the database. Note that you must have bands in the database before you can add albums; these should already exist if you do not call the clearBands method again. You must extract the band id from the list of bands in order to add an album.
2. Now we have the helper class working! Now create a JSP page to contain a list of all the bands. Use JSTL or Java scriptlets to populate the list from the bandList property of DBHelper.
3. Add a form to the JSP page to add a band. To make this work, you will need the following new methods in DBHelper:
   1. setBandName
   2. setCommand
4. Add a form to search for albums by a particular band. The user should be able to select the band name from a Select element. The form needs to send the band Id, not the band name to the DBHelper. You can figure out how to do this by looking up the <OPTION> element in HTML. Make sure the user does not need to enter band ids!
5. Finally, add a form to add a new album to the list.

When you are done, upload the three files you have created, or an archive of the project, and submit it.

1. The recommended way to start mysql on Unix systems. See http://dev.mysql.com/doc/refman/5.0/en/mysqld-safe.html [↑](#footnote-ref-1)
2. If you do forget the new root password, here are the directions to reset it: <https://dev.mysql.com/doc/refman/5.5/en/resetting-permissions.html>. [↑](#footnote-ref-2)
3. [↑](#footnote-ref-3)
4. If using MS Access, the class name is sun.jdbc.odbc.JdbcOdbcDriver, and the database url is jdbc:odbc:mist4630. Also please change the triumphant comments to refer to ODBC and Access. [↑](#footnote-ref-4)
5. This can create problems when running tests on an existing application that contains lots of valuable data. In this case, we would create a separate test database. [↑](#footnote-ref-5)