What is MySQL?
An open source relational database management system; therefore, available on most Unix versions, MS Windows, and macOS.

**mysql** command line utility is a simple interface for executing SQL statements.

**MySQL Workbench** is a GUI-based product for administering MySQL databases. Users can examine the contents of the MySQL catalog and execute SQL statements.

**Example #1:** Using the mysql command line utility

To launch the mysql command line utility to connect to server "db01" for user "hwk06":

```
$ mysql -h db01 -u hwk06 -p
```

It will prompt you to enter the password.

Some important commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>use database;</code></td>
<td>Connect to the specified database.</td>
</tr>
<tr>
<td><code>mysql&gt; use flightdb;</code></td>
<td>To specify flightdb.</td>
</tr>
<tr>
<td><code>exit;</code></td>
<td>Exit the mysql utility</td>
</tr>
</tbody>
</table>

When an SQL select statement is executed, it shows the results in a tabular format.

Example:

```
mysql> select * from Customer;
+-----+---------------+---------------+---------+------+
<table>
<thead>
<tr>
<th>custNr</th>
<th>name</th>
<th>preferAirline</th>
<th>birthDt</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Perry Noid</td>
<td>Spirit</td>
<td>2000-04-01</td>
<td>M</td>
</tr>
<tr>
<td>222</td>
<td>Melba Toast</td>
<td>American</td>
<td>1990-09-11</td>
<td>F</td>
</tr>
<tr>
<td>333</td>
<td>Pete Moss</td>
<td>Clampett</td>
<td>1992-03-03</td>
<td>M</td>
</tr>
<tr>
<td>444</td>
<td>Bill Board</td>
<td>American</td>
<td>1985-04-04</td>
<td>M</td>
</tr>
<tr>
<td>480</td>
<td>Anita Vacay</td>
<td>Spirit</td>
<td>1975-06-01</td>
<td>F</td>
</tr>
<tr>
<td>555</td>
<td>Jerry Tall</td>
<td>PoDunk</td>
<td>1927-04-15</td>
<td>M</td>
</tr>
<tr>
<td>666</td>
<td>Tom E Gunn</td>
<td>Clampett</td>
<td>1976-06-06</td>
<td>M</td>
</tr>
<tr>
<td>777</td>
<td>Bob Wire</td>
<td>PoDunk</td>
<td>1973-07-07</td>
<td>M</td>
</tr>
<tr>
<td>888</td>
<td>Ron DeVoux</td>
<td>American</td>
<td>1990-08-08</td>
<td>M</td>
</tr>
<tr>
<td>890</td>
<td>Ann T Freeze</td>
<td>American</td>
<td>1992-08-09</td>
<td>F</td>
</tr>
<tr>
<td>901</td>
<td>Peg Board</td>
<td>Delta</td>
<td>1987-04-04</td>
<td>F</td>
</tr>
<tr>
<td>902</td>
<td>Al B Tross</td>
<td>American</td>
<td>1957-07-12</td>
<td>M</td>
</tr>
<tr>
<td>903</td>
<td>B B Gunn</td>
<td>PoDunk</td>
<td>1976-09-09</td>
<td>F</td>
</tr>
<tr>
<td>904</td>
<td>Sally Mander</td>
<td>Delta</td>
<td>1995-09-04</td>
<td>F</td>
</tr>
</tbody>
</table>
```
MySQL Workbench
This provides a GUI for examining a database. When accessing this remotely, you must set up x-windows by using an x-server. We recommend using MobaXterm.

To launch MySQL workbench:
$ mysql-workbench (no spaces)

MySQL Workbench - Connect to Database Window
The example on the right shows what you will specify for program #3.

To examine the flightdb as hwk06:
- Use **hwk06** for Username
- Use **flightdb** for the Default Schema
- Press OK
- When prompted for the password, use **hwk06pw**

Example #2: Connecting MySQL Workbench to a database (i.e., schema)
On the MySQL Workbench window:
- Select the Database menu option
- Select Connect to Database

Example #2 continued: On the Connect to Database window:
- Use “db01” for Hostname
- Use your abc123 ID for Username
- Use “abc123db” for the Default Schema
- Press OK
- You will be prompted to enter your password which isn’t your Linux password. It is the last 3 digits of your banner id followed by **pw**
MySQL Workbench – Inspecting a Schema
You can see the tables, columns, and indexes in your database by using the Schema Inspector.

Example #3: Inspecting a schema
In the left subwindow:
- right click on your database
- Select Schema Inspector
- It will probably give you a warning window about lacking a privilege to see some events. Simply ignore that message and click OK.
MySQL Workbench – Inspecting a Schema (continued)
With that sub-window on the right, you can easily examine your tables, showing columns, indexes, and views.

Example #3 continued:
What is JDBC?
Java Database Connectivity (JDBC) is an application programming interface for Java providing access to databases.

Important Java classes:
- `DriverManager` connect to your database
- `Statement` execute an SQL statement
- `PreparedStatement` precompiles an SQL statement, allowing it to be repeatedly executed more efficiently
- `Connection` returned by the `DriverManager` for use in creating `Statement` and `PreparedStatement` instances
- `ResultSet` returned by queries and may contain many tuples

Example #4: create a sql folder and copy sample java code.
Create an sql folder to contain Java code.
Set that folder as your current directory and copy the sample code to that folder:

```
$ cp -R /usr/local/courses/clark/cs3743/sql .
```
That will create a cs3743 directory (for the package) and copy several java source files.

To compile the code, change directory to your sql directory and then do the following:

```
$ javac cs3743/cs3743ExampleMain.java
$ javac cs3743/MySQLExample.java
```

To execute the code:

```
$ java cs3743/cs3743ExampleMain
```

Set your CLASSPATH to include the mysql.jar
As of March 2018, we are using tcsh as our default shell. You should modify your ~/.cshrc file to specify the CLASSPATH to include /usr/share/java/mysql.jar

Example #5: Modify your .cshrc file for the CLASSPATH
Use vim or another editor to modify your ~/.cshrc file to specify the CLASSPATH. Example contents of .cshrc:

```
# New files are created without group/other permissions
umask 077
set path = ($path $HOME/bin)
set l3423 = /usr/local/courses/clark/cs3423
set l3723 = /usr/local/courses/clark/cs3723
set l3743 = /usr/local/courses/clark/cs3743
setenv CLASSPATH /usr/share/java/mysql.jar:
```

Connecting to the MySQL server
Example #6 shows a subset of the code for MySQLExample.java which connects to our database server.

The MySQLExample constructor class receives a user ID and a password as parameters.
- It uses the Java Class loader to load the MySQL jdbc driver.
- It uses `DriverManager` to get the connection to the specified database on the specified database server.

To run our example, you can literally use

```
package cs3743;
import java.sql.Connection;
import java.sql.DriverManager;
public class MySQLExample {
    private Connection connect = null;
    public MySQLExample (String user, String password) throws Exception {
        try {
            // This will load the MySQL driver, each DBMS has its own driver
            Class.forName("com.mysql.jdbc.Driver");
            this.connect = DriverManager.getConnection
```

For program #3, you will substitute your *abc123* ID followed by *db*.

```
("jdbc:mysql://contactInstructor:3306/abc123db",
  user
  , password);
}
catch (Exception e)
{
  throw e;
}
```

**Example #7: main method invoking our MySQLExample software**

Instead of including the main method in the MySQLExample class, we will put it in a separate class. You will rarely use the class that contains this main method; however, it does have your user id and special mysql password for this class. We don't want that shown to others very often.

```java
package cs3743;
public class cs3743ExampleMain
{
  public static void main(String[] args) throws Exception
  {
    MySQLExample pgm = new MySQLExample("eduex", "edupw");
    pgm.runProgram();
  }
}
```

**Invoking SQL in Java Code**

There are two approaches which use different JDBC classes:

- **Statement**
- **PreparedStatement**

Both of those will use the **ResultSet** class which returns the result of a query.

**Using the Statement Class to Select**

We can create an instance of the Statement class by executing the **createStatement** method on our connection instance (which was returned by the DriverManager).

A select statement can be executed by using the **executeQuery** instance method from the Statement class.

**Example #8: using createStatement and executeQuery methods**

```java
// MySQLExample.java
package cs3743;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.Statement;
import java.sql.ResultSet;
public class MySQLExample
{
  private Connection connect = null;
  private Statement statement = null;
  private ResultSet resultSet = null;
  private PreparedStatement preparedStatement = null;
...
  public void runProgram() throws Exception
  {
    try
    {
      // Statement allows us to issue SQL statements to the database
```
state
createStatement();
// Get the sections and save in resultSet
resultSet = statement.executeQuery("select * from abc123db.Section");
printSections("Beginning Sections", resultSet);

Using the ResultSet Class
Important instance methods:
next() advances to the next tuple in the ResultSet instance. If this is the first call, it advances to the first tuple. If there is a tuple, true is returned.
getString(colNm) get a string value for the specified column
getInt(colNm) get an integer value for the specified column
getDouble(colNm) get a double value for the specified column
gDate(colNm) get an java.sql.Date value for the specified column
If the specified column allows null values, the returned value could be null.

In general to process a ResultSet instance:
while (resultSet.next()) {
    // process a row
}

Example #9: processing the ResultSet
private void printSections(String title, ResultSet resultSet) throws SQLException {
    // The current position in resultSet is before the first row
    System.out.printf("%s\n", title);
    System.out.printf("%-8s %-7s %4s %-12s %-4s %-5s %s\n" , "Semester", "Course", "Sect", "Prof", "Days", "Time", "Room");
    // print each tuple in the result set
    while (resultSet.next()) {
        // It is possible to get the columns via name
        // also possible to get the columns via the column number
        // which starts at 1
        String  timeStr;  // can be null
        String semesterStr = resultSet.getString("semester");
        String courseNrStr = resultSet.getString("courseNr");
        int     sectNr = resultSet.getInt("sectNr");
        String  profStr = resultSet.getString("prof");  // can be null
        if (profStr == null) profStr = "---";
        String daysStr = resultSet.getString("classDays");  // can be null
        if (daysStr == null) daysStr = "---";
        java.sql.Time classTm = resultSet.getTime("classTime"); // can be null
        if (classTm == null) timeStr = "---";
        else timeStr = classTm.toString().substring(0,5);    // only use the first 5 chars
        String  roomStr = resultSet.getString("room");      // can be null
        if (roomStr == null) roomStr = "---";
        System.out.printf("%-8s %-7s %4d %-12s %-4s %-5s %s\n" , semesterStr
                    , courseNrStr
                    , sectNr
                    , profStr
                    , daysStr
                    , timeStr
                    , roomStr);
    }
    System.out.printf("\n");
}

Found vs not found
Example #10: Recognizing no rows found when multiple rows could be returned
How to check for no rows returned?
Approach 1:
Our software usually needs to know whether any rows were returned. Our code is fairly simple if we provide values for the unique primary key, returning only one row:

```java
if (resultSet.next())
{
    // process that one row
}
else
{
    // process for not found
}
```

The code is a little more difficult if the result could be multiple rows. Example #10 shows two approaches for handling this situation.

```java
count = 0;       // count the number of rows returned
while (resultSet.next())
{
    count += 1;
    // process a row
}
if (count == 0)
    System.out.println("No rows returned");
```

Approach 2:
```java
if (resultSet.next())
{
    do
    {
        // process the row
    } while (resultSet.next())
}
else
    System.out.println("No rows returned");
```

Using the Statement Class to Insert Rows
We can use `executeUpdate` from a `statement` instance to insert, delete or update rows.

Since the string values in a SQL statement are quoted, we have to escape those quotes using backslashes.

If `executeUpdate` fails, it will raise an `SQLException`. We will discuss that below.

```
define executeUpdate("insert into abc123db.Section "
    + "(`semester`, `courseNr`, `sectNr`, `prof`, `classDays`, "
    + "`classTime`, `room`)"
    + "values(\"20181Sp\", \"CS3743\", 001, \"Clark\", \"TuTh\","
    + \"10:00\", \"NPB1.202\")");
```

Using the PreparedStatement Class to Select
There are multiple advantages of the PreparedStatement Class to execute SQL SELECT statements:

- More efficient when the SQL statement is executed multiple times in the same program.
- Substitution parameter values can be passed in a more consistent manner, allowing us to easily pass values from variables. We show parameters as question marks.

To set a substitution parameter's value, `PreparedStatement` provides multiple set methods (similar to the get methods of ResultSet):

```java
setString(parmNr, value) sets the specified parameter which must be a string value.
```

Example #11: Using Statement's `executeUpdate` to insert a row
```
define executeUpdate("insert into abc123db.Section "
    + "(`semester`, `courseNr`, `sectNr`, `prof`, `classDays`, "
    + "`classTime`, `room`)"
    + "values(\"20181Sp\", \"CS3743\", 001, \"Clark\", \"TuTh\","
    + \"10:00\", \"NPB1.202\")");
```
**setInt** *(parmNr, value)*  sets the specified parameter which must be an int value.

**setDouble** *(parmNr, value)*  sets the specified parameter which must be a double value.

**setTime** *(parmNr, value)*  sets the specified parameter which must be a java.sql.Time value.

**setDate** *(parmNr, value)*  sets the specified parameter which must be a java.sql.Date value.

There are additional methods.

<table>
<thead>
<tr>
<th>Using the PreparedStatement Class to Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>We have the same advantages as above.</td>
</tr>
<tr>
<td>Example #13: setting multiple substitution parameters for an insert</td>
</tr>
<tr>
<td>// Instead of just passing a string as is done with</td>
</tr>
<tr>
<td>// statement.executeQuery and statement.executeUpdate,</td>
</tr>
<tr>
<td>// we can use preparedStatement which allows us</td>
</tr>
<tr>
<td>// to provide variable values to replace &quot;?&quot; values.</td>
</tr>
<tr>
<td>preparedStatement = connect.prepareStatement</td>
</tr>
<tr>
<td>&quot;Insert into abc123db.Section values</td>
</tr>
<tr>
<td>+(&quot;20181Sp&quot;, ?, ?, ?, ?, ?, ?)&quot; );</td>
</tr>
<tr>
<td>// semester, courseNr, sectNr, prof, classDays, classTime, room</td>
</tr>
<tr>
<td>// Parameters start with 1</td>
</tr>
<tr>
<td>String courseNr = &quot;CS3723&quot;;</td>
</tr>
<tr>
<td>int sectNr = 1;</td>
</tr>
<tr>
<td>String prof = &quot;Al Gall&quot;;</td>
</tr>
<tr>
<td>String classDays = &quot;MWF&quot;;</td>
</tr>
<tr>
<td>java.sql.Time classTime = java.sql.Time.valueOf(&quot;10:00:00&quot;);</td>
</tr>
<tr>
<td>preparedStatement.setString(1, courseNr);</td>
</tr>
<tr>
<td>preparedStatement.setInt(2, sectNr);</td>
</tr>
<tr>
<td>preparedStatement.setString(3, prof);</td>
</tr>
<tr>
<td>preparedStatement.setString(4, classDays);</td>
</tr>
<tr>
<td>preparedStatement.setTime(5, classTime);</td>
</tr>
<tr>
<td>preparedStatement.setString(6, &quot;Flying Saucer&quot;);</td>
</tr>
<tr>
<td>// insert it</td>
</tr>
<tr>
<td>preparedStatement.executeUpdate();</td>
</tr>
</tbody>
</table>

**Handling Exceptions**

One of the errors that must be detected for insertions, is a duplicate entry error. In Java, we will use try catch to assist in handling the error.

Unfortunately, the error codes have not been standardized across all implementations of SQL.

<table>
<thead>
<tr>
<th>Example #14: Handling a duplicate key error</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class MySQLExample</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>private Connection connect = null;</td>
</tr>
<tr>
<td>private Statement statement = null;</td>
</tr>
<tr>
<td>private ResultSet resultSet = null;</td>
</tr>
<tr>
<td>public static final int ER_DUP_ENTRY = 1062;</td>
</tr>
<tr>
<td>public static final int ER_DUP_ENTRY_WITH_KEY_NAME = 1586;</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>// insert it</td>
</tr>
<tr>
<td>try</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>// Execute that insert statement</td>
</tr>
</tbody>
</table>
```java
catch (SQLException e)
{
    switch (e.getErrorCode())
    {
        case ER_DUP_ENTRY:
        case ER_DUP_ENTRY_WITH_KEY_NAME:
            System.out.printf("Duplicate key error: %s
", e.getMessage());
            break;
        default:
            throw e;
    }
}
catch (Exception e)
{
    throw e;
}
```

### JDBC type vs Java type

<table>
<thead>
<tr>
<th>JDBC Type</th>
<th>Java Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR</td>
<td>String</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>BIGINT</td>
<td>long</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>TIME</td>
<td>java.sql.Time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>java.sql.Timestamp</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>java.math.BigDecimal</td>
</tr>
</tbody>
</table>

See the product-specific JDBC documentation for additional data types.

### Other Implementations

IBM DB2 and Oracle use a colon notation for substitution parameters.

### Example #15: Another Embedded SQL Implementation

Inserting using parameters based on colon notation:

```sql
EXEC SQL
  INSERT INTO SUPPLIER (SNUM, SNAME, STATUS, CITY)
VALUES ( :Supplier.szSupplierNum
        , :Supplier.szName
        , :Supplier.iStatus
        , :Supplier.szCity
    );
```

Deleting using parameters based on colon notation:

```sql
EXEC SQL
  DELETE FROM SHIPMENT
WHERE SNUM = :Supplier.szSupplierName
  AND PNUM = :Part.szPartNum;
```