

# An Introduction to Computer Science with Java



Copyright 2013 by C. Herbert, all rights reserved.

Last edited August 21, 2013 by C. Herbert

This document is a draft of a chapter from *Computer Science with Java*, written by Charles Herbert with the assistance of Craig Nelson. It is available free of charge for students in Computer Science 111 at Community College of Philadelphia during the Fall 2013 semester. It may not be reproduced or distributed for any other purposes without proper prior permission.

# Introduction to Computer Science with Java



## Chapter 21 – Java Database Connectivity

---

*This chapter is about Java database connectivity – Using Java to connect to a modern relational database management system so we can perform SQL queries on the database. The chapter includes a quick overview of relational database management systems and SQL, and then focuses on connecting to a database and running SQL queries from within Java software.*

---

### Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe what a modern relational database is and how data is organized in such a database.
- define key relational database terminology, including *entity*, *instance*, *attribute*, *primary key* and *metadata*.
- describe what Structured Query Language (SQL) is, who defines the language, and how it is used; and list some of the major modern relational database management systems that use SQL.
- describe the format for an SQL query to extract given data from a database, and create such queries, including queries that extract given columns, queries that extract rows meeting given conditions, and queries that extract data in a specified order.
- describe how to connect a Java application to a remote database server and write java code that establishes such a connection.
- describe the four types of JDBC drivers used to connect Java applications to a remote database, and how to incorporate a JDBC type-four driver into a NetBeans project for database connectivity.
- create Java code that uses SQL statements to query a remote database and retrieve both metadata and data from database tables.
- create java code to display the result set from an SQL query on the console, and to send data from a result set to a text data file and to a CSV data file that can be opened in Microsoft Excel.

## 19.1 Relational Database Management Systems

In this chapter we will learn how to communicate with a database from inside Java software. We will start with a discussion of relational database management systems. Most modern databases use Structured Query Language (SQL) which we will discuss in the next section.

A **database** is an organized collection of data, usually one that can be stored and processed using a computer system. It can be thought of as an electronic filing system. Almost all modern databases are relational databases. A **relational database** is a database with data organized as a set of tables, based on concepts from relational algebra. A **relational database management system** is a software system used to establish, maintain and manipulate a modern relational database.

The things we wish to keep track of in a relational database are called **entities**, with a table for each entity. The word entity refers to the general object that each table describes. Each specific object is called an **instance** of the entity, similar to instances of a class of objects. In fact, one relational model for database management is called the object-relational model, which includes features such as inheritance found in object-oriented programming.

Just as with properties of an object, entities do not need to be physical objects. A teaching assignment is an example of a non-physical entity – it is not a physical object, but it is still something we can record in a database. A bank transaction and a phone call are other examples of entities that are not physical.

Information about each instance of an entity is stored as an **attribute**, similar to the properties of an object in Java programming. Each attribute has a data type.

Database tables have one row for each instance of an entity and one column for each attribute. The table below shows data for instances of the entity *student*.

snum	last	first	street	city	state	zip	phone	major	gpa
S00188	Smith	John	111 Oak Ave.	Princeton	NJ	08541	555-1111	CIS	3.43
S03154	Jones	Mary	212 Maple St.	Hanover	NH	03755	555-2222	Computer Science	3.62
S10843	Jarvis	Fred	123 Spruce St.	Providence	RI	02901	555-4321	English	3.07
S11013	Ali	Ben	327 Pine St.	Cambridge	MA	02140	555-1212	Accounting	2.91
S11783	Washington	Martha	419 Beech Lane	New York	NY	10027	555-1532	History	3.67
S36912	Williams	Cathy	231 Ash Terrace	New Haven	CT	06520	555-3251	Biology	2.64
S41134	Franklin	Richard	213 Birch Drive	Philadelphia	PA	19104	555-1334	Finance	3.14
S72394	Banner	Blythe	222 Larch St.	Ithaca	NY	14853	555-4422	Agriculture	3.01

The intersection of a row and a column in a database table is called a **field**. Each field holds the value of one attribute for one instance of the entity. The term field is often used interchangeably with attribute, even though they are not exactly the same thing.

Relational database management systems work by cross-referencing the data in their tables. In order for this work properly, we must be able to uniquely identify each instance of an entity. A **primary key** is an attribute that has a different value for each instance of an entity, so we can uniquely identify that instance. Every table in a relational database must have a primary key. In our student example above,

student number is the primary key for students. Two students can have the same name, but no two students can have the same student number, so student number works as a primary key. A registration record in a registration table would only have the student number – the primary key for students. The database software would use this to obtain other attributes, such as first name or last name, to create a printed copy of a registration record with more detail. (Some primary keys require more than one column and are called composite keys, but many database specialists say that composite keys should be avoided.)

This is that nature of a relational database. We have a table for each entity, with a row for each instance of the entity and a column for each attribute of the entity. A special attribute, called the primary key, uniquely identifies each instance of the entity.

The full database name for a column includes the table name and the column name, separated by a comma, similar to objects and properties, but we do not use the names of the instances in a database. In fact, instances do not have names in a database. So, the columns in the student table would have the names *student.snum* , *student.last* , *student.gpa* , and so on.

Many modern databases are enterprise-wide databases. An **enterprise-wide database** is a single database with all of the data for an enterprise, such as a business or a government agency. The trend in modern computing is to have all of the major operational data– accounting, sales, scheduling, personnel, and so on –in one centralized enterprise-wide database management system, administered by database professionals. This makes it increasingly important for Java to be able to interface with modern database management systems. Logos of the major relational database management software systems are shown below. Currently Oracle, MySQL, SQL Server and DB2 dominate the market.

### Database Metadata

Information about how a database is organized – in other words, data about the data – is called **metadata**. The metadata for a table tells us the name of each column in the table, the type of data in the column, and sometimes the size of the column.

Here is the metadata for the student table:

- snum, char, 10
- last, char, 20
- first, char, 20
- street, char, 35
- city, char, 20
- state, char 2
- zip, char, 5
- phone, char, 12
- major, char, 15
- gpa, numeric, 5,2



As you can see, most of the attributes have the *char* or character data type, analogous to a String in Java. Many modern databases use *UTF-8* encoding by default, so it may be necessary to find out about this when working with a particular database. Recall that Java uses UTF-16.

Most relational database management systems use the standard SQL numeric data types – INTEGER and SMALLINT for integers; and DECIMAL, NUMERIC, FLOAT, REAL, and DOUBLE PRECISION for floating point values. (Data types are named using all capital letters in SQL.)

NUMERIC and DECIMAL are known as precision data types with the precision specified by the size of the field. If there is only one number for the field size, then the data is stored as an integer allowing for that many decimal digits. If the field size is specified by two numbers, then it is a fractional decimal number with the total number of digits and then the number of digits to the right of the decimal point specified.

Most relational database systems have several other data types for specialized data, such as dates, time and so on. For a more complete list of data types in the most commonly used database management systems see the *W3 Schools* page for SQL Data Types, on the Web at:

[http://www.w3schools.com/sql/sql\\_datatypes.asp](http://www.w3schools.com/sql/sql_datatypes.asp)

---

## 19.2 Structured Query Language (SQL)

**Structured Query Language** – SQL for short – is a language we can use to query a relational database. SQL is both a **data definition language** (DDL) used to define the structure of a database, and a **data manipulation language**, (DML) used to manipulate data in a database, including extracting data from a database. SQL is based on relational algebra. A standard definition of the SQL language is maintained by the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). Note that SQL should be pronounced “S-Q-L” and not “sequel”, since *Sequel* is the name of a specific database management software package.

SQL is most often used to query a database. Technically, any SQL statement is called a **query**, but people generally use the word query to specifically mean statements that extract data from a database. We will learn about queries that extract data from a database, but not about SQL statements to create or modify databases. A more detailed look at database management systems and SQL is included in specialized database management courses, such as *CIS 205 –Database Management Systems* at Community College of Philadelphia.

Almost all modern relational database management systems understand and accept standard SQL queries, but they each have some added commands, so their specialized versions of SQL are slightly different. In this section we will learn how to write simple SQL standard queries that work on all of these systems to extract data from a database table. In subsequent sections we will learn how to include these queries in our Java software.

Most Java programmers who write software to access databases are either trained in database management or work with a database professional who has such training. However, access to working relational database management systems is often restricted for security and data integrity reasons, even for a programmer who is also a highly trained database professional.

In this section we will see how to format a query to extract data from a database table, and how to use comparison operators and Boolean logic in queries with different data types.

## Formatting a Query

It is fairly easy to extract information from a single table in a relational database. The proper format for a basic SQL query to extract information from a table is:

```
SELECT [attributes] FROM [table] WHERE [condition];
```

For example, if we wish to get a list of the names and phone numbers from our student table for all of the students who live in New York, the proper SQL query would be:

```
SELECT first, last, phone FROM student WHERE state = "NY";
```

This query would return:

First	Last	Phone
Martha	Washington	555-1532
Blythe	Banner	555-4422

The attribute list tells the database what attributes, or columns, we wish to see in our result. The condition tells the database which instances, or rows, we wish to include in our result. The database will look at the rows one at a time and include any row that meets the condition. In this case, the attribute lists tells the database to show us the data in the *first*, *last*, and *phone* columns and the condition tells the database to include every row that has *NY* in the *state* column.

Notice that our result is in the format of a new table. A SELECT... FROM... WHERE... query will use the criteria we specify to create and return a new table from existing tables in a database. The table that an SQL query returns is called a **result set**. The response to a query to retrieve data is either a result set or an error message. The SQL query above returns a table with the *first*, *last* and *phone* columns for the two rows with *NY* as the value in the *state* column.

The columns in a result set may be columns from the database, such as *student number* from a student table in the database, or derived columns. **Derived columns** contain data that is not in the database, but is derived from data in the database, such as *gross pay* derived from *hours* and *rate* in a database that has attributes for hours and rate but not gross pay. Some derived data is summary data, such as the number of rows in a database that meet certain criteria, or the average of the data in a particular column.

The SQL language is not case sensitive, so capitalization doesn't matter for SQL commands, table names, and column names. However, capitalization does matter for the data in the database, so be careful. The column name *state* could be *STATE* or *State*, but "*ny*", "*Ny*" and "*NY*" are not all the same data. We must ask for the data the way it was entered into the database.

Usually, SQL programmers and database administrators put SQL commands in UPPERCASE and table and column names in lowercase to make queries easier to read, so that's a good practice for us to adopt in our SQL queries in Java code. In our simple extraction queries, SELECT, FROM, and WHERE should be UPPERCASE, but column names like *last* and *first*, and table names like *student*, should be lowercase.

SELECT... FROM... WHERE... queries are often written on three lines to make them easier to read and understand, like this:

```
SELECT first, last, phone
```

```
FROM student
```

```
WHERE state = "NY";
```

SQL queries end with a semicolon. A computer will use the semicolon to recognize the end of the query, so it doesn't matter if the query is on one line, or spread out over several lines.

Creating SQL SELECT... FROM... WHERE... queries for a single table is fairly easy, but multiple table queries can be difficult to create, because factors based on relational algebra need to be considered, such as the data dependencies between tables. So, we will not attempt to write multiple table SQL queries in this course.

### How Data Types affect Queries

As mentioned above, character data, also called text data, is simply a string of text stored in a database, similar to Java Strings. Traditional text data, such as names and addresses, are stored as text data, but so are numbers that will not be used for arithmetic, such as phone numbers and zip codes.

Text data literals in SQL queries have quotes around them, similar to String literals in Java. The first three queries in the examples below use text data with quotes.

Numeric data is a number stored in a database in a format that allows computers to perform arithmetic with it, similar to integer and floating point data in Java. Data that might be used in an arithmetic operation, such as hours or rate in a payroll database, or temperature, mass, length or width in a scientific database, should be stored as numeric data.

Numeric data is used in database queries without quotes around it. The last three queries in the example below use numeric data without quotes.

Some numbers, such as social security numbers and student numbers, are used as ID numbers, and not to perform arithmetic, so they should be stored as text data, not numeric data, just as with java Strings.

Boolean data is data that has a true or false value, similar to boolean data in java. The name of a Boolean field used in a query means *true*, and NOT([name of Boolean field]) means *false*, as shown in queries 3 and 4 below.

There are many other data types, such as date and time, and a special data type called binary large object (BLOB) for objects such as picture files, videos, and sound clips. There are even specialized number formats for integers, floating point numbers, and different currencies. For the rest of this chapter, we will only use the text, numeric, and Boolean data types so that we can focus on how to extract data and format our results.

## SQL Query Examples

The table below contains metadata for a table of cars. It will be used in the example queries below to show how Boolean logic and comparison operators work in SQL queries.

Table: car			
column name	data type	size	notes about the data
VIN	Text	20	Vehicle Identification Number (primary key)
make	Text	40	manufacturer ("Ford", "Toyota", etc.)
model	Text	20	model name ("Focus", "Camry", etc.)
color	Text	2	
year	text	4	
engine	text	10	
auto	Boolean		true = automatic transmission; false = manual
value	Number		value of the car in US dollars

The following examples each show a request for data, followed by an SQL command to extract the desired data.

Example 1 – list the VIN, model, year and value for all Fords.

```
SELECT VIN, model, year, value FROM car WHERE make = "FORD";
```

Example 2 – list the make, model, year, and value for all cars worth more than \$20,000.

```
SELECT make, model, year, value FROM car WHERE value > 20000;
```

Example 3 – list the model, year and value for all cars that have an automatic transmissions.

```
SELECT model, year, value FROM car WHERE auto;
```

Example 4 – list the model, year and value for all cars that do not have an automatic transmissions.

```
SELECT model, year, value FROM car WHERE NOT(auto);
```

Example 5 – list the VIN, year and value for all Honda Civics.

```
SELECT VIN, year, value
FROM car
WHERE make = "Honda" AND model = "Civic";
```

Example 6 – list the make, model, year and value for all Subaru and Volvos.

```
SELECT make, model, year, value
FROM car
WHERE make = "Subaru" OR make = "Volvo";
```

Example 7 – list the model, year and value for all red Porsches valued at less than \$40,000.

```
SELECT model, year, value
FROM car
WHERE color = "red" AND make = "Porsche" AND value <= 40000;
```

Example 8 – list the make model, year and value for cars that are not Chevrolets.

```
SELECT make, model, year, value FROM car WHERE make <> "Chevrolet";
SELECT make, model, year, value FROM car WHERE NOT(make = "Chevrolet");
```



## Query Conditions

The conditions in SQL queries are Boolean conditions that are *true* or *false* – similar to boolean conditions in Java. They involve the same six comparison operations as Java – *equals*, *does not equal*, *is less than*, *is not less than*, *is greater than*, or *is not greater than* – shown below.

Condition	In Math	In SQL	Examples
A equals B	$A = B$	$A = B$	zipcode = "19130"
A is not equal to B	$A \neq B$	$A <> B$	status <> "active"
A is less than B	$A < B$	$A < B$	name < "Miller"
A is greater than B	$A > B$	$A > B$	temperature > 98.6
A is less than or equal to B	$A \leq B, A \nless B$	$A \leq B$	rate <= 12.50
A is greater than or equal to B	$A \geq B, A \nless B$	$A \geq B$	hours >= 40

Comparisons are not needed for Boolean data, since it is already true or false, So, Boolean attributes are used by themselves in logical conditions to mean true.

```
SELECT first, last FROM delegate WHERE citizen;
```

The condition is simply WHERE citizen, ~~not WHERE citizen = TRUE.~~

Similarly, proper SQL command to return a list of all rows where the citizen attribute is not true would be:

```
SELECT first, last FROM delegate WHERE NOT(citizen);
```

## Boolean Functions in Database Queries

There are three basic Boolean logic functions used in SQL queries: *AND*, *OR*, and *NOT*. They are used to form compound logical conditions from simple conditions, as shown in the examples above. They have the same meaning as the *&&*, *//* and *!* operators in Java.

For example, if we wish to find the names of employees in the Accounting department with at least 40 years of experience then we need to use two conditions: department = "Accounting", experience >= 40. We use the AND operation to join the two simple conditions into one compound condition:

```
SELECT first, last
FROM employee
WHERE department = "Accounting" AND experience >= 40;
```

The AND operation is a binary conjunction operation, that needs two operands. As in Java, If both operands are true, then the result is true. Otherwise, the result is false. Both operands must be true for the result to be true.

The OR operation also has two operands. If either operand is true, then the result is true. Otherwise, the result is false. Both operands must be false for the result to be false.

The NOT operation is a unary operation with only one operand. It simply reverses the true or false value of its operand. If the operand is true, then the result is false. If the operand is false, then the result is true.

---

## 19.3 Database Connectivity

The **Java Database Connectivity (JDBC)** API is a Java core API that “*can access any kind of tabular data, especially data stored in a Relational Database.*”<sup>1</sup> The JDBC API enables Java code to connect to a data source, send a query to a database, and receive and process the result set from the query. Official information about using JDBC is available from the Oracle Technology Network, online at:

<http://www.oracle.com/technetwork/java/overview-141217.html>

A JDBC driver is necessary to connect an application using the JDBC API to the system and software hosting an SQL-based relational database. There are four different types of JDBC drivers:

Type 1 – a **JDBC to ODBC bridge driver**, which connects java applications to an *Open Database Connectivity* (ODBC) driver, which in turn connects to a database management system. It requires the use of an ODBC driver.

Type 2 – a **native API JDBC driver**, which requires an API from the database software vendor (Oracle, etc.) to be installed on the machine with the Java software. It is platform and vendor specific.

Type 3 – a **middleware JDBC driver**, also called a **JDBC-net driver**, which is used on a network communications server between the machine with the Java application and the database server. It is written entirely in Java, often implemented as a JAR file. *middleware JDBC drivers* are platform independent, but specific drivers must be used for specific database management systems.

Type 4 – a **pure Java JDBC driver**, which is a JAR file made accessible to the Java application, on the same machine as the application, usually in a JAR library file. It contains the driver software as Java class files. *Pure Java JDBC drivers* are platform independent, but specific drivers must be used for specific database management systems.

Type 3 and Type 4 drivers are similar to each other and easiest to use. Type 3 drivers are located on a middleware system when many client systems are connecting to a single database management system over a network, such as the Internet. Type 4 drivers do the same thing, but are located on the same machine as the Java application.

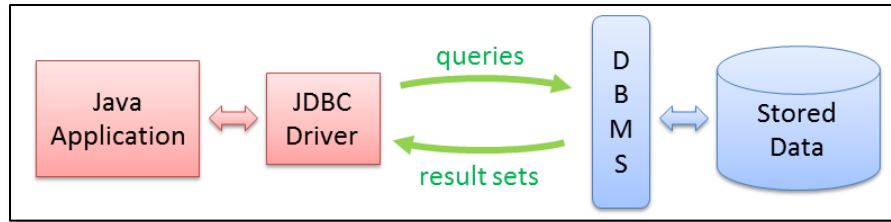
In this chapter, we will use **Connector/J**, a Type 4 pure Java JDBC driver, to connect over the Internet to a *MySQL* database established by the author on his Web site. *Connector/J* and the *MySQL* relational database management system are both freely available open source software. For more information on *MySQL*, including the *MySQL* documentation, download, installation instructions, see:

<http://www.mysql.com>. For more information on the *Connector/J* JDBC driver, including download and documentation links, see: <https://dev.mysql.com/downloads/connector/j>. The *Connector/J* JAR file is included with the files for this chapter. The *MySQL* manual is also included with the files for this chapter.

The connection process is shown in the diagram below:

---

<sup>1</sup> from the Oracle's *The Java Tutorials: JDBC Introduction* Web page, online at:  
<http://docs.oracle.com/javase/tutorial/jdbc/overview/index.html>



### Information Needed to Establish a JDBC Connection

Several pieces of information are needed to successfully connect to a database using JDBC from within a Java application:

- **the host IP address** - This is the Internet address of the server that is hosting the database management system. It could be a URL, a fixed IP address, or a reference to a *local host* if the database management system is on the same system as the Java application. A fixed IP address usually works better than a URL for remote access. A fixed IP address is a four-part or six-part Internet Protocol address. IP version 4 (IPv4) addresses are the traditional form for Internet addresses – such as *68.178.216.151*, the fixed IPv4 address for the system hosting author's database used in this chapter. IPv6 addresses look like this: *2001:db8:0:1234:10:567:12:11*.
- **database** - We need to know the name of the database that the Java application will access. For example, the database which we will use in this chapter is *CWHDemo*. It is possible that a single host system has more than one database.
- **username** – A username is needed for the database account that the application will use to access the database. This must be set up by someone with administrative rights to the system the application will access. For our work in this chapter, the username is *student*.
- **password** – A password is needed to go with the username for the database account that the application will use. Our password is *Student%123* (with a capital S).

### Establishing a JDBC connection

Two steps are necessary to connect to a database from within a Java application using a type 4 JDBC driver, which is a JAR file that contains the software needed to connect to a database as Java classes. First, we must make the JDBC driver class with the communication software for our database available to our code, then we must instantiate a connection to the database as a `Connection` class object.

The driver class can be made accessible several different ways. With an IDE such as NetBeans, the JDBC driver can be added as a local library for our software. We could also download the JAR file and make it accessible to our code by using the correct import statement or by using a class loader instruction in our java code. You may see code with one of the following instruction:

- `import com.mysql.jdbc.Driver;`
- `Class.forName(com.mysql.jdbc.Driver)`

The first is an import statement for the JDBC driver class. The second is a Java instruction to load the required class. Either of these commands could work, but only if the JAR file is in a location available to our software. Since we will load the JAR file as a library accessible to our NetBeans project, neither instruction is needed. NetBeans will make sure the correct classes from the Library are accessible.

## Software Exercise – Adding the JDBC Driver to a NetBeans Library

### STEP 1.

Download the necessary Jar file. The file we need is: **mysql-connector-java-5.1.29-bin.jar**. It is included with the files for this chapter. It can also be downloaded as a zip file from the MYSQL website, at: <https://dev.mysql.com/downloads/connector/j>

Remember where you put the driver when you download it. If you download the zip file, it needs to be unzipped before it can be used.

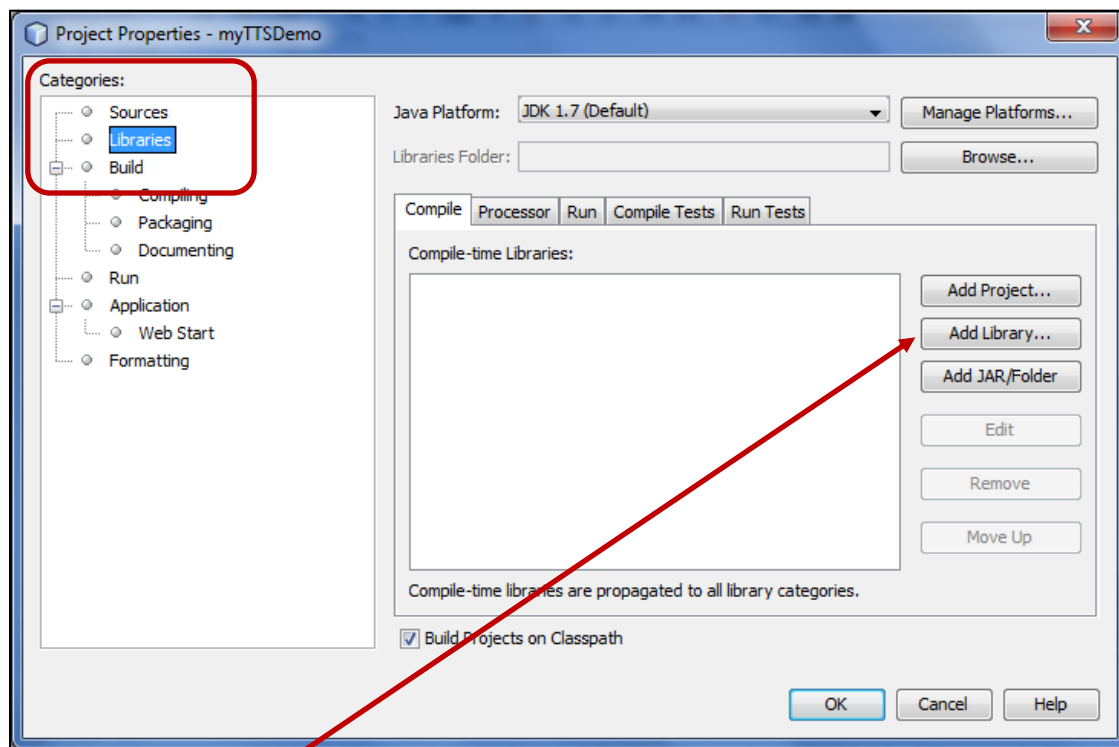
### STEP 2.

Begin the NetBeans project that will connect to a database. In this exercise we will use the NetBeans project **RemoteMySQLDemo**, included as a zipped file with the files for this chapter. Download the file, unzip it, and open the project in NetBeans.

### STEP 3.

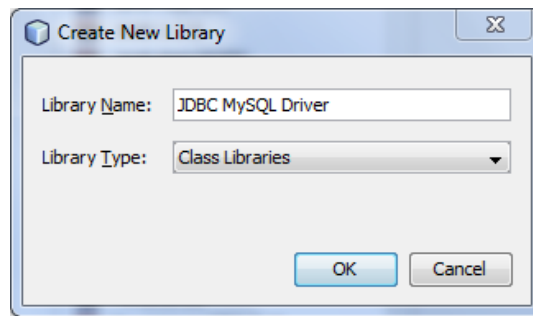
We need to create a library with the JAR file and add it to this project. This is similar to adding an external API to a NetBeans project, as discussed in Chapter 17 - Using External APIs.

Right-click the **RemoteMySQLDemo** project's node in the NetBeans project window, then select **Properties** at the bottom of the menu that appears. The Project Properties Window will open. Select the **Libraries** category, circled in the image below.



### STEP 4.

Click the **Add Library** button, then click the **Create** button on the *Add Library* window that appears. A *Create New Library* window will appear on top of the *Add library* window.

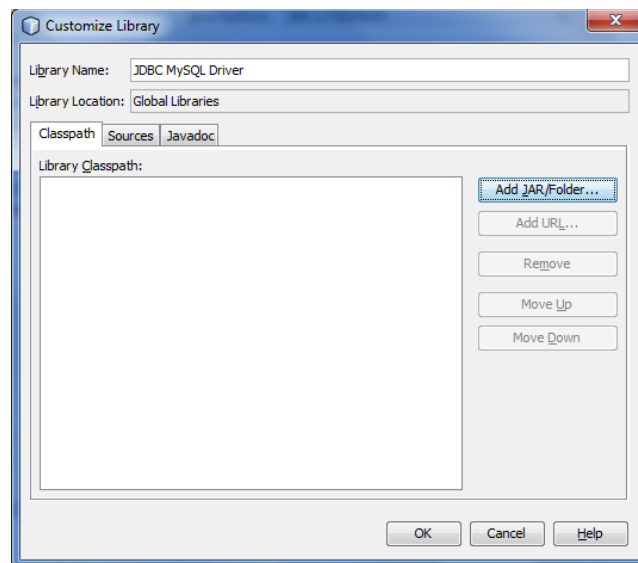


We will create a new Library with the JAR file needed to use the driver in our Java projects in NetBeans. We will name the library **JDBC MySQL Driver** and add the JAR files to the library.

**STEP 5.**

Type **JDBC MySQL Driver** as the *Library Name* in the *Create New Library* window, then click the **OK** button.

A *Customize Library* window will now appear.



**STEP 6.**

Click the **Add JAR/Folder...** button in the *Customize Library* window, then navigate to the **lib** folder with the FreeTTS Jar files – this is the same JAR file you opened in step 1, above.

**STEP 7.**

Find and select the JAR file **mysql-connector-java-5.1.29-bin.jar**, then click the **Add JAR/Folder** button. Click the **OK** button when the *Customize Library* window reappears, then click the **Add Library** button when the *Add Library* window reappears.

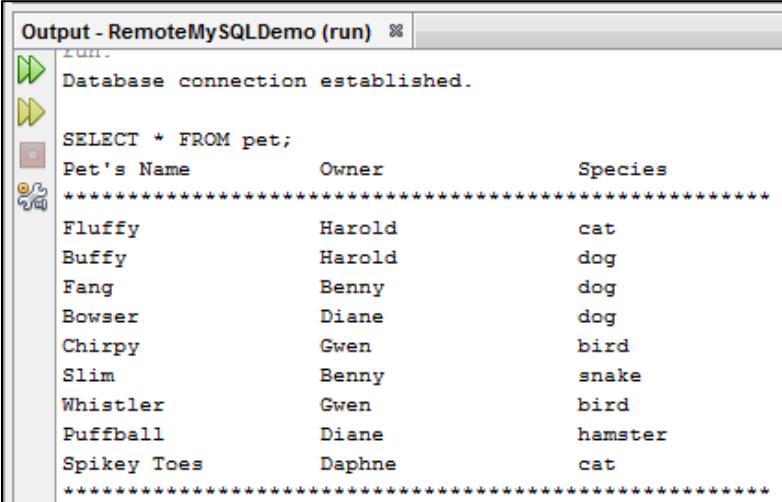
The JDBC driver for MySql is now in the **JDBC MySQL Driver** library in NetBeans on this computer and will be available for use in all of your NetBeans projects. Now we can test it by running this project.

### Using an existing JDBC MySQL Driver Library

Once you set up the library with the JAR file containing the MySQL driver, you can use it whenever you want to write code to connect to a MySQL database. To do so, you should right-click the project icon in the Netbeans Project window, then select properties from the menu that appears, then select the *Libraries* category on the window that appears, and add the library to your project.

#### STEP 8.

Run the project. The Java application should connect remotely to the database on the author's Website, extract the requested data from the database, and display it on the screen. You should see a result similar to the one shown below. In the next section we will examine the code that does this.



```

Output - RemoteMySQLDemo (run) %
Database connection established.

SELECT * FROM pet;
Pet's Name      Owner      Species
*****
Fluffy          Harold     cat
Buffy           Harold     dog
Fang            Benny      dog
Bowser          Diane      dog
Chirpy          Gwen       bird
Slim            Benny      snake
Whistler        Gwen       bird
Puffball        Diane      hamster
Spikey Toes     Daphne     cat
*****
  
```

## 19.4 Using SQL within Java Applications

Here is the code from the **RemoteMySQLDemo** application used in the last section. It connects to the remote database server, queries the database, returns a result set and then prints part of the result set.

```

/* RemoteMySQLDemo.Java      last edited 3/31/2014 by C.Herbert
 * This code connects to a remote MYSQL database on the website CWHerbert.com
 * It connects via a fixed IP address (IPv4) to the database CWHDemo and
 * the table "pet", which is the same as the table used in the MySQL Documentation,
 * Chapter 3 tutorial.  Read only access to the database and table is granted for
 * the puposes of learning to write java code to connect to and query a database
 * using SQL from within Java.
 *
 * host IP address: 68.178.216.151
 * database:        CWHDemo
 * username:        student
 * password:        Student%123  (Note capital "S" in password, but not in username.)

Table metadata can be retrieved with the query "DESCRIBE pet;"
  
```

```

name    varchar(20)
species varchar(20)
sex      char(1)
birth    date
death    date
*/

/* The class for the JDBC communications driver for MySQL must be available.
 * It can be included in a locally accessible library,
 * by using the import statement    import com.mysql.jdbc.Driver;
 * or by using the Class.forName(com.mysql.jdbc.Driver) within your code
 */

package remotemysqldemo;
import java.sql.*;

public class RemoteMySQLDemo {

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

        // Connect to a database by establishing a Connection object
        Connection conn = DriverManager.getConnection
            ("jdbc:mysql://68.178.216.151/CWHDemo", "student", "Student%123");

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

        // Create a statement Object for this database connection
        Statement st = conn.createStatement();

        // call a method that performs a query using Statement st
        selectAll(st);

        // Close the connection
        conn.close();
    } // end main()

    /**
     * The following method performs an SQL query
     * The parameter must be a Statement object with an established connection
     * to an SQL database.
     */
    public static void selectAll(Statement s)
        throws SQLException, ClassNotFoundException {

        String queryString;    // a String to hold an SQL query
        ResultSet rs;          // the result set from an SQL query as a table

        // Create an SQL query as a String for this statement
        // this query returns all rows and all columns from the database
        queryString = "SELECT * FROM pet;";

        // Send a statement executing the query and saving the result set
        rs = s.executeQuery(queryString);

        // print headings for the output

```

```

        System.out.println(queryString);
        System.out.printf("%-20s%-20s%-20s\n", "Pet's Name", "Owner", "Species");
        System.out.println("*****");

        // Iterate through the result set and print name, owner, and species attributes
        while (rs.next())
            System.out.printf("%-20s%-20s%-20s\n", rs.getString(1), rs.getString(2),
                               rs.getString(3));

        System.out.println("*****");

    } // end selectAll()
} // end class

```

The java instruction `Connection conn = DriverManager.getConnection(...);` establishes a connection to the database as an instance of the Connection class named **conn**. The declaration for the Connection class is on the JAR file with the JDBC driver.

The parameters (`"jdbc:mysql://68.178.216.151/CWHDemo", "student", "Student%123"`) for the connection include:

- the database address – 68.178.216.151
- the name of the database – CWHDemo
- the username for the database account – student
- the password for the database account – student%123

Notice that each of these parameters is enclosed quotes, because each is passed to the connection as a String parameter.

The database address and the name of the database are incorporated in a connection String for the JDBC to MySQL connection. It has the form :

**`jdbc:mysql://[database server address here]/[database name here]`**

**`jdbc:mysql://`** identifies this as a java JDBC to MySQL connection, just as **`HTTP://`** identifies a *HyperText Transfer Protocol* connection or **`FTP://`** identifies a *File Transfer Protocol* connection.

The username and password must come from the database administrator who must set up a database user account to grant someone access to the database. The name *student* and password *student%123* were set up by the author to give students read-only access to the example database.

Once the connection to the database is established, a statement containing an SQL query needs to be created and sent to the database. It will return a result set.

In the application above, this is done in three parts. First an instance of the *Statement* class is created for this connection. Statements are commands that can be sent over the connection to the database server. They are not limited to SQL queries – administrative commands, such as commands to login, change users, etc., could also be sent over the connection.



The instruction `Statement st = conn.createStatement();` creates an instance of the `Statement` class named **st** and matches it to the **conn** instance of the `Connection` class.

The `Statement` object **st** is passed to the method `SelectAll()`, which performs the query.

Within the method `SelectAll()`, the SQL query to be sent over the statement's connection is saved as `String` named **queryString**.

This query – `SELECT * FROM pet;` – selects all rows and all columns in the database table **pet**. The `*` character is a wildcard, which, in this case, means all columns in the database. This query is a `SELECT...FROM...WHERE` query, but without the `WHERE` clause that would tell the database which rows to return, so it returns all rows. So, this query returns all rows and all columns in the table.

The Java statement `rs = st.executeQuery(queryString);` will execute the query on the database and return the result set, which is saved in the instance of the `Result set` class name **rs**. The **ResultSet** class is defined in the JDBC driver JAR file, just as the **Connection** and **Statement** classes are.

The result set will be in the form of a table with a column for each attribute retrieved from the database and a row for each instance retrieved from the database. You must know what the metadata for the database table is in order to work with the result set. The organization of the data in the result set is defined by your SQL query.

## Working with Result Sets

The `ResultSet` Class has many methods allowing Java programmers to work with a result set from a query. Some of the more commonly used methods from the `ResultSet` class are described below. More complete documentation for the `ResultSet` class is available online at:

<http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html>

The method in the application `rs.next()`, returns a true value if there is another row in the result set and moves to the next row. It returns a false value if there is no next row in the result set.

Commonly Used <i>ResultSet</i> Class Methods	
Name and Description	Return Type
<b><code>getBoolean(int columnIndex)</code></b> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a boolean in the Java programming language.	Boolean
<b><code>getBoolean(String columnLabel)</code></b> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a boolean in the Java programming language.	Boolean
<b><code>getBytes(int columnIndex)</code></b> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a byte in the Java programming language.	byte
<b><code>getBytes(String columnLabel)</code></b> Retrieves the value of the designated column in the current row of this <code>ResultSet</code> object as a byte in the Java programming language.	byte

<b><i>getDouble(int columnIndex)</i></b>	double
Retrieves the value of the designated column in the current row of this ResultSet object as a double in the Java programming language.	
<b><i>getDouble(String columnLabel)</i></b>	double
Retrieves the value of the designated column in the current row of this ResultSet object as a double in the Java programming language.	
<b><i>getFloat(int columnIndex)</i></b>	float
Retrieves the value of the designated column in the current row of this ResultSet object as a float in the Java programming language.	
<b><i>getFloat(String columnLabel)</i></b>	float
Retrieves the value of the designated column in the current row of this ResultSet object as a float in the Java programming language.	
<b><i>getInt(int columnIndex)</i></b>	int
Retrieves the value of the designated column in the current row of this ResultSet object as an int in the Java programming language.	
<b><i>getInt(String columnLabel)</i></b>	int
Retrieves the value of the designated column in the current row of this ResultSet object as an int in the Java programming language.	
<b><i>getLong(int columnIndex)</i></b>	long
Retrieves the value of the designated column in the current row of this ResultSet object as a long in the Java programming language.	
<b><i>getLong(String columnLabel)</i></b>	long
Retrieves the value of the designated column in the current row of this ResultSet object as a long in the Java programming language.	
<b><i>getObject(String columnLabel, Class&lt;T&gt; type)</i></b>	<T> T
Retrieves the value of the designated column in the current row of this ResultSet object and will convert from the SQL type of the column to the requested Java data type, if the conversion is supported.	
<b><i>getShort(int columnIndex)</i></b>	short
Retrieves the value of the designated column in the current row of this ResultSet object as a short in the Java programming language.	
<b><i>getShort(String columnLabel)</i></b>	short
Retrieves the value of the designated column in the current row of this ResultSet object as a short in the Java programming language.	
<b><i>getString(int columnIndex)</i></b>	String
Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.	
<b><i>getString(String columnLabel)</i></b>	String
Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.	

Notice that most of the methods that return values can identify the column in the database table from which the data will be extracted using either the column index or the column name. For example, the code from the **RemoteMySQLDemo** application shown above has the loop:

```
while (rs.next())
    System.out.printf("%-20s%-20s%-20s\n", rs.getString(1),rs.getString(2),rs.getString(3) );
```

The method `rs.getString(1)` will return the value of the first column in the result set as a Java String. It uses the index 1 to refer to the first column in the result set, which is the ***name*** column from the table

*pets*. The index values start at 1; there is no column 0 in the result set. Which column is column 1? That depends on your SQL query. The query in the **RemoteMySQLDemo** application returns all columns in the table in order. Remember, the columns that are included in the result set and their order are determined by the on the query.

We could have also used the column names as String parameters for most of these get methods. `rs.getString("column name")` will return the same column as `rs.getString(1)`. In the *RemoteMySQLDemo* application, the instruction:

```
System.out.printf("%-20s%-20s%-20s\n", rs.getString(1), rs.getString(2),
                rs.getString(3));
```

could be replaced with:

```
System.out.printf("%-20s%-20s%-20s\n", rs.getString("name"), rs.getString("owner"),
                rs.getString("species"));
```

The code below shows how to retrieve metadata telling us the column names and data types.

### Retrieving a Table's Metadata from Within Java

The SQL command to retrieve the metadata from a table is: `DESCRIBE tablename;`

The SQL command to show what tables are included in a database: `SHOW TABLES;`

The following method performs a query to retrieve the metadata for the *pet* table and display the first two columns in the result set, which show us the *name* and *data type* for each column in the pet table.

```
/* This method performs as SQL query that returns the metadata for the pet table
 * The parameter must be a Statement with an established Connection.          */
public static void showColumns(Statement s)
    throws SQLException, ClassNotFoundException {

    String queryString;    // a String to hold an SQL query
    ResultSet rs;         // the result set from an SQL query as a table

    // Create an SQL query as as String for this statement
    // this query returns all rows and all columns from the database
    queryString = "Describe pet;";

    // Send a statement executing the query and saving the result set
    rs = s.executeQuery(queryString);

    // print headings for the output
    System.out.println("Columns in the pet table:");

    System.out.printf("%-10s%-10s\n", "Column", "Datatype");
    System.out.println("*****");

    // Iterate the result set and print name, owner, and species attributes
    while (rs.next()) {
        System.out.printf("%-10s%-10s\n", rs.getString(1), rs.getString(2));
    }

    System.out.println("*****\n");
} // end showColumns
```

The NetBeans project **RemoteSQLMetaDataDemo** is similar to the **RemoteMySQLDemo** project, but with the *showColumns()* method added to the project. It is included with the files for this chapter.

The metadata for the table is:

Table: Pets	
name	varchar(20)
species	varchar(20)
sex	char(1)
birth	date
death	date

---

## 19.5 Programming Exercises – Modifying SQL Queries within a Java Application

This section contains exercises with modifications to the remote MySQL application to help better understand how SQL queries work in Java code. We will do each of the following:

1. Modify the code to use the column names instead of idenx values for the result set.
2. Modify the *selectAll()* method to show the owner first in the result set, then the species, then the pet's sex, then the pet's name in order according to the owner's name.
3. Add a new method to Select pets that are cats and include only name and owner in the result set.
4. Modify the previous query to send the results to a text data file.

**NOTE:** You must complete the programming exercise **Adding the JDBC Driver to a NetBeans Library** in section 19.3 above to add the required library to the project before doing these exercises.

### Programming exercise 1 – using column names in a result set

In this exercise we will modify the existing *selectAll()* method in the *RemoteMySQLDemo* application to use the column names when specifying which columns from the result set we should display. The column names in the result set are the same as the column names in the database, unless we explicitly change them in the SQL command.

#### STEP 1.

Open the **RemoteMySQLDemo** Netbeans project. If you have not already added the **JDBC MySQL Driver** library to the project, then do so now.

Run the application to make sure it works. If it does not work then you should debug the application before continuing.

The method *selectAll()* in the application contains the following code:

```
public static void selectAll(Statement s)
    throws SQLException, ClassNotFoundException {

    String queryString;      // a String to hold an SQL query
    ResultSet rs;           // the result set from an SQL query as a table
```

```

// Create an SQL query as a String for this statement
// this query returns all rows and all columns from the database
queryString = "SELECT * FROM pet;";

// Send a statement executing the query and saving the result set
rs = s.executeQuery(queryString);

// print headings for the output
System.out.println(queryString);
System.out.printf("%-20s%-20s%-20s\n", "Pet's Name", "Owner", "Species");
System.out.println("*****");

// Iterate through the result set and print name, owner, and species attributes
while (rs.next())
    System.out.printf("%-20s%-20s%-20s\n", rs.getString(1), rs.getString(2),
        rs.getString(3));

    System.out.println("*****");

} // end selectAll()

```

**STEP 2.**

Change the code in the while loop that outputs the desired columns from the result set. It currently is:

```

System.out.printf("%-20s%-20s%-20s\n", rs.getString(1), rs.getString(2),
    rs.getString(3));

```

Change it to be:

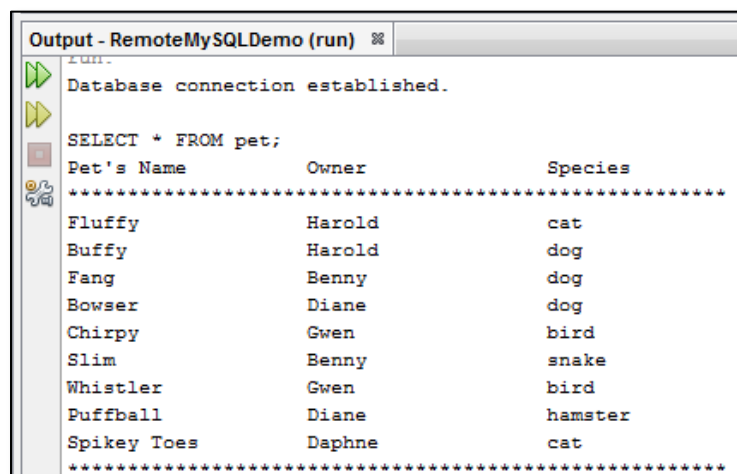
```

System.out.printf("%-20s%-20s%-20s\n", rs.getString("name"), rs.getString("owner"),
    rs.getString("species"));

```

**STEP 3.**

Run the application. The output should be the same as it was before the change:



```

Output - RemoteMySQLDemo (run)
Database connection established.
SELECT * FROM pet;
Pet's Name      Owner      Species
*****
Fluffy          Harold     cat
Buffy           Harold     dog
Fang            Benny      dog
Bowser          Diane      dog
Chirpy          Gwen       bird
Slim            Benny      snake
Whistler        Gwen       bird
Puffball        Diane      hamster
Spikey Toes     Daphne     cat
*****

```

## Programming exercise 2 – Modifying an existing SQL query

In this exercise we will modify the existing `selectAll()` method in the *RemoteMySQLDemo* application to show the owner first in the result set, then the species, then the pet's sex, then the pet's name.

This is a continuation of exercise 1 above. Complete exercise 1, and then with the **RemoteMySQLDemo** Netbeans project open, complete the steps below.

### STEP 1.

Change the line that sets the *queryString*. It is currently:

```
queryString = "SELECT * FROM pet;";
```

Change it to be:

```
queryString = "SELECT owner, species, sex, name FROM pet;";
```

### STEP 4.

Change the heading for the output accordingly. It is currently:

```
System.out.printf("%-20s%-20s%-20s\n", "Pet's Name", "Owner", "Species");
```

Change it to be:

```
System.out.printf("%-12s%-12s%-6s%-12s\n", "Owner", "Species", "Sex", "Pet's Name");
```

### STEP 5.

Change the line in the while loop that displays the data from the result set. It should match the headings. It is currently:

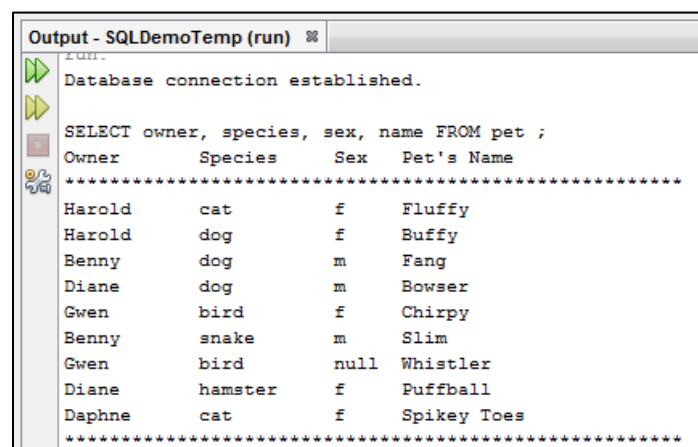
```
System.out.printf("%-20s%-20s%-20s\n", rs.getString("name"), rs.getString("owner"),
    rs.getString("species"));
```

Change it to be:

```
System.out.printf("%-12s%-12s%-6s%-12s\n", rs.getString("owner"), rs.getString("species"),
    rs.getString("sex"), rs.getString("name") );
```

### STEP 6.

Run the application. Your output should look something like this:



```

Output - SQLDemoTemp (run)
Database connection established.

SELECT owner, species, sex, name FROM pet ;

Owner      Species    Sex      Pet's Name
*****
Harold      cat        f        Fluffy
Harold      dog        f        Buffy
Benny       dog        m        Fang
Diane      dog        m        Bowser
Gwen       bird        f        Chirpy
Benny      snake       m        Slim
Gwen       bird        null     Whistler
Diane      hamster     f        Puffball
Daphne     cat         f        Spikey Toes
*****
  
```

**STEP 7.**

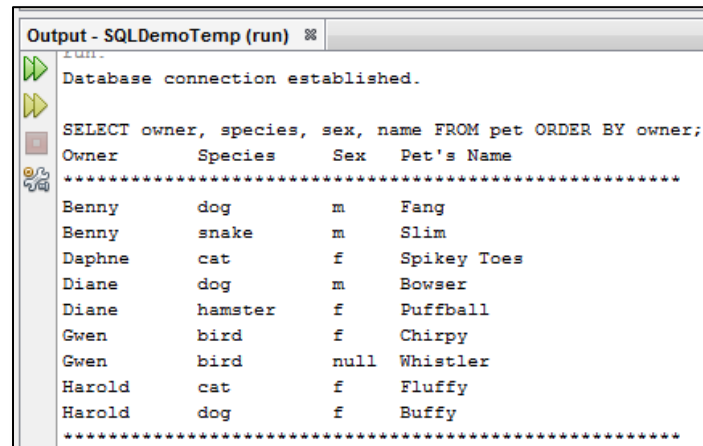
Now we will change the query again, this time to put the result set on order according to the owner's name. We will add an `ORDER BY` clause to the end of the query. It is currently:

```
queryString = "SELECT owner, species, sex, name FROM pet;";
```

Change it to be:

```
queryString = "SELECT owner, species, sex, name FROM pet ORDER BY owner;";
```

The output should now be:



```

Output - SQLDemoTemp (run)
Database connection established.
SELECT owner, species, sex, name FROM pet ORDER BY owner;
Owner      Species  Sex  Pet's Name
*****
Benny      dog      m    Fang
Benny      snake   m    Slim
Daphne     cat      f    Spikey Toes
Diane     dog      m    Bowser
Diane     hamster  f    Puffball
Gwen       bird     f    Chirpy
Gwen       bird     null Whistler
Harold     cat      f    Fluffy
Harold     dog      f    Buffy
*****

```

An SQL result set can be put in order by more than one column. To do so, list the columns in the *order by* clause separated by a comma, such as *ORDER BY owner, name*. The rows in the result set will be sorted by the first column listed, then by next column if they have the same value for the first column.

### Programming exercise 4 – Creating an SQL query with a WHERE clause

In this exercise we will create a new method in the *RemoteMySQLDemo* to select pets that are cats and include only name and owner in the result set.

**STEP 1.**

Open the **RemoteMySQLDemo** Netbeans project. You can start with the original *RemoteMySQLDemo* or use the modified project resulting from exercises 1 and 2 above.

If you have not already added the **JDBC MySQL Driver** library to the project, then do so now.

**STEP 2.**

The application contains the method `selectAll()`.

Copy the `selectAll()` method and paste a new copy in the project between the end of the existing `selectAll()` method and the end of the class.

```

    } // end selectAll()
    ← Paste a copy of the
    selectAll() method here.
} // end class

```

**STEP 3.**

Change the name of copy of the method to be ***selectSome()***. Change the comment at the end of the method accordingly.

**STEP 4.**

In the main method, add an instruction to run the method ***selectSome()***. It should come after the instruction to run the method ***selectAll()***. You modified code in the main method should now have two method calls, look like this:

```
// call a method that performs a query using Statement st
selectAll(st);

// call a method that performs a query using Statement st
selectSome(st);
```

Run the method to see if the copy works. If it does, you will get two copies of the current output.

**STEP 5.**

Change the SQL queryString in the new ***selectSome*** method to show only pets that are cats and include only name and owner in the result set. The new instruction should look like this:

```
queryString = "SELECT owner, name FROM pet WHERE species = 'cat';";
```

The added *WHERE* clause tells the database to include in the result set only rows where the species is equal to "cat". Quotes are needed around the *species* name cat because it is a String literal in the SQL command. We can use single quotes because the entire queryString is already in double quotes.

**STEP 6.**

Change the instruction to print the heading for the output. It should be:

```
System.out.printf("%-12s%-12s\n", "Owner", "Pet's Name");
```

**STEP 7.**

Change the instruction to print the rows from the result set accordingly. It should be:

```
System.out.printf("%-12s%-12s\n", rs.getString("owner"), rs.getString("name") );
```

**STEP 8.**

Run the code. Your output should now look like this:

```
Output - SQLDemoTemp (run) %
SELECT owner, name FROM pet WHERE species = 'cat';
Owner    Pet's Name
*****
Harold    Fluffy
Daphne    Spikey Toes
*****
```



## Programming exercise 5 – Sending an SQL query result set to a data file

In this exercise we will modify the *selectSome()* method from the previous exercise to store the result of the query in a text data file. You should complete the previous exercise before starting this exercise.

### STEP 1.

You should have the NetBeans project from the previous exercise open to start this exercise. It should contain the method *selectSome()*.

### STEP 2.

We need to add an import statement to the project to allow file operations in java. Add the following statement to the project below the package directive near the beginning of the file:

```
import java.io.*;
```

The order of the import statements is not critical. Your code should look something like this:

```
package sqldemotemp;  
import java.sql.*;  
import java.io.*;
```

### STEP 3.

Next, we need to create a File class Object identifying the destination data file and a PrintWriter to send data to the file. Add the following code near the beginning of the *SelectSome()* method:

```
// create a File class object and give the file the name SQLresult.txt  
java.io.File sqlResult = new java.io.File("SQLresult.txt");  
  
// Create a PrintWriter text output stream and link it to the file x  
java.io.PrintWriter outfile = new java.io.PrintWriter(sqlResult);
```

### STEP 4.

The method must now deal with file I/O exceptions in addition to SQL exceptions. For now, the easiest way to do this is by changing the *throws* clause for the file header. In a real world situation we should include appropriate try and catch blocks.

You need to make a similar change to the *throws* clause for the main method.

Change the *throws* clause for the *selectSome()* method header to throw all exceptions. The code for the beginning of the method should now look something like this:

```
public static void selectSome(Statement s) throws Exception {  
  
    // create a File class object and give the file the name SQLresult.txt  
    java.io.File sqlResult = new java.io.File("SQLresult.txt");  
  
    // Create a PrintWriter text output stream and link it to the File sqlResult  
    java.io.PrintWriter outfile = new java.io.PrintWriter(sqlResult);  
  
    String queryString;    // a String to hold an SQL query  
    ResultSet rs;          // the result set from an SQL query as a table
```

**STEP 5.**

Now that we have a File object and PrintWriter in place, this next step is the easy part. Simply change each `System.out` to refer to `outfile` instead. The code for the output should be similar to this:

```
// print headings for the output

outfile.println(queryString);
outfile.printf("%-12s%-12s\n", "Owner", "Pet's Name");
outfile.println("*****");

// Iterate through the result set and print owner, and pet's name attributes
while (rs.next())
    outfile.printf("%-12s%-12s\n", rs.getString("owner"), rs.getString("name") );

outfile.println("*****");

outfile.close();
```

Notice that an `outfile.close();` statement has been added. This is necessary to close the file and output stream when we are finished sending data to the file.

The data file `SQLresult.txt` should now be in the NetBeans project folder for this project. It should contain the following text:

```
SELECT owner, name FROM pet WHERE species = 'cat';
Owner      Pet's Name
*****
Harold      Fluffy
Daphne      Spikey Toes
*****
```

This code sends the data with a heading and lines of stars to the file. If you wish to send just the data in the result set to the file, then you can remove the lines that print the heading and the stars.

You could also modify the output from an SQL query to send the data to a CSV file, which could then be opened in Microsoft Excel.

---

## Chapter Review

**Section 1** introduced Relational database management systems. Key terms included *database*, *relational database*, *relational database management system*, *entities*, *instance*, *attribute*, *field*, *primary key*, *enterprise-wide database* and *metadata*.

**Section 2** discussed the use of Structured Query Language(SQL). Key terms included *Structured Query Language*, *data definition language(DDL)*, *data manipulation language(DML)*, *query*, *result set* and *derived columns*.

**Section 3** Java Database Connectivity(JDBC) including a description of the different type of JDBC drivers, the information needed to establish a connection between a Java application and a remote database, and how the process of establishing a connection works. An exercise in the chapter showed how to add a type 4 JDBC driver (a driver in as Java code in a JAR file) to a NetBeans project as an external library.

**Section 4** showed how to incorporate SQL queries into Java programs and work with the result set returned from the database being queried. An example at the end of the section showed how to retrieve a table's metadata from within a Java application.

**Section 5** included four programming examples showing different aspects of working with SQL queries and result sets from within java code, including how to refer to result set columns by name as well as by index number, how to modify an existing query to return only certain columns, how to return a result set sorted by a particular column, how to use a WHERE clause to return only certain rows, and how to send the results of a query to a text file.

---

## Chapter Questions

1. What are some of today's most commonly used database management systems? Which of these dominate the market?
2. What do almost all modern relational database management systems have in common?
3. What is a data definition language (DDL) used for? What is data manipulation language (DML) used for? Which is SQL? Who defines the standard definition of SQL?
4. Describe the proper format for a basic SQL query to extract information from a relational database table. Give an example of such a query.
5. How is the result set from an SQL query formatted?
6. What are some of the commonly used data types for an SQL query? How are values of different data types used in an SQL query?

Create SQL queries to extract the following from the table: house, whose metadata is shown in the table below:

7. The House ID, street address, and value for houses in the 19140 zip code under \$240,000.
8. The House ID, street address, and value for houses in the 19128 zip code with at least three bedrooms and two bathrooms.

9. The House ID, street address, zip codes and value for houses sold by Mark Jones.
10. The street address, value, number of bedrooms and number of bathrooms for houses in the 19116 and 19154 zip codes.
11. The House ID, street address, zip codes and values for house in the 19139 zip code that have not been sold.

<i>Table: house</i>			
<i>column name</i>	<i>data type</i>	<i>size</i>	<i>notes about the data</i>
<b>ID</b>	Text	5	house ID number (primary key)
<b>street</b>	Text	40	street address
<b>city</b>	Text	20	
<b>state</b>	Text	2	
<b>zip</b>	Text	5	5-digit zip codes only
<b>bedrooms</b>	Number		number of bedrooms; integer
<b>bathrooms</b>	Number		number of bathrooms; could end in .5
<b>value</b>	Number		appraised value or sale price of house
<b>first</b>	Text	20	first name of the listing or selling agent
<b>last</b>	Text	20	last name of the listing or selling agent
<b>sold</b>	Boolean		true = sold ; false = not sold

12. What are the four types of JDBC drivers used to connect java applications to a database? How do they differ from one another?
13. How can a JDBC type 4 driver be added to a NetBeans project?
14. What information is needed to establish a connection to query a remote database from within a Java application? Give an example of a connection String that might be used to connect a java application to a MYSQL database.
15. What is database metadata? Give an example of metadata for a typical database table.
16. What SQL command can be used to retrieve the metadata for a database table? What SQL command can be used to show the names of the tables included in a database?
17. An instance of what object needs to be established to connect to a remote database? An instance of what object is needed to send a query to a database once the application connects to the database? An instance of what object is used to capture the results returned from of a database query?
18. What two different ways can we identify the columns in a database queries result set? What determines the order of the columns in the result set?
19. How can we tell the database to return only certain rows from a table in a result set? How can we tell the database to return the rows in the result sorted set in a particular order?
20. How can we send the data resulting from a data base query to a text data file?

## Chapter Exercise

The database **CWHDemo** on the server at IP address **68.178.216.151** has a table named **fall2014** with information about computer courses offered at Community College of Philadelphia in the Fall 2014 semester.

Your task is to develop a Java application with two methods:

1. a method that queries the database, retrieving and writing in a CSV file the *crn*, *subject*, *course*, *section*, *days* and *time* for all CSCI courses, in order according to the course number. You should be able to open the CSV file in Microsoft Excel.
2. a method that queries the database with a query of your own design based on a question you will write. The results can be displayed neatly on the console.

Your query might answer a question such as *What 4 credit courses are available on Tuesday and Thursday?* or *What sections of OA courses are being offered online?* You should make up your own question, write the query to find the answer, and then create a method to get the answer from within a java application.

You can write a single application with methods for each of the two queries.

Here is annotated metadata for the table:

Column	Type	Notes
crn	char(20)	CRN primary key
subject	varchar(5)	CIS, CSCI, or OA
course	varchar(5)	course number
section	varchar(5)	section number
credits	integer	number of credits
time	varchar(20)	the time the course meets.
days	varchar(8)	the days the course meets: M T W R F S (No Sunday courses.)
term	varchar(5)	15A means the course meets for the entire 15-week term. 7A courses meet for the first half of the term. 7B or 7N courses meet for the second half of the term.
campus	varchar(5)	MAIN, NERC,NWRC, WEST
room	varchar(8)	the room number
enrollment	integer	The data in this column is not real. It was randomly generated. All other columns contain real data.

(The data in the table is real data, except for the enrollment, which was randomly generated before students began registering for the courses.)

## Contents

Chapter 21 – Java Database Connectivity .....	2
Chapter Learning Outcomes .....	2
19.1 Relational Database Management Systems .....	3
Database Metadata .....	4
19.2 Structured Query Language (SQL) .....	5
Formatting a Query .....	6
How Data Types affect Queries .....	7
SQL Query Examples .....	8
Query Conditions .....	9
Boolean Functions in Database Queries .....	9
19.3 Database Connectivity .....	10
Information Needed to Establish a JDBC Connection .....	11
Establishing a JDBC connection .....	11
Software Exercise – Adding the JDBC Driver to a NetBeans Library .....	12
Using an existing JDBC MySQL Driver Library .....	14
19.4 Using SQL within Java Applications .....	14
Result Set Methods .....	17
Commonly Used <i>ResultSet</i> Class Methods .....	17
Retrieving a Table’s Metadata from Within Java .....	19
19.5 Programming Exercises – Modifying SQL Queries within a Java Application .....	20
Programming exercise 1 – using column names in a result set .....	20
Programming exercise 2 – Modifying an existing SQL query .....	22
Programming exercise 4 – Creating an SQL query with a WHERE clause .....	23
Programming exercise 5 – Sending an SQL query result set to a data file .....	25
Chapter Review .....	27
Chapter Questions .....	27
Chapter Exercise .....	29