24.4 *(Employee Database) In Section 10.5, we introduced an employee-payroll hierarchy to calculate each employee's payroll. In this exercise, we provide a database of employees that corresponds to the employee-payroll hierarchy. (A SQL script to create the employees database is provided with the examples for this chapter.) Write an application that allows the user to:*

a) Add employees to the employee table.
b) Add payroll information to the appropriate table for each new employee. For example, for a salaried employee add the payroll information to the salariedEmployees table.

Figure 24.33 is the entity-relationship diagram for the employees database.

First, this problem requires a lot of set-up. The Java DB method described in the text has a lot of issues when using the latest edition of Java, so we decided to use MySQL instead to build our database. We simply download MySQL at dev.mysql.com/downloads/mysql/ and complete the install process with all additional connectors installed. During the configuration process, we make sure to create a local InnoDB cluster to create our database later (the employees.sql script from Deitel uses InnoDB). We also must create a root account,

Now, once MySQL is installed, we open command prompt and navigate to the folder where mysql.exe is installed (C:\Program Files\MySQL\MySQL Server 8.0\bin on my machine), then execute `mysql -h localhost -u root -p` to open a MySQL session with the root account. We then type `USE mysql`, to select the built-in database named mysql, which stores server information, such as user accounts and their privileges for interacting with the server.

Now, we will want to create a user account with name 'deitel' to access and modify the database we will create with employees.sql. To do this, we use the command `create user 'deitel'@'localhost' identified by 'deitel'; grant select, insert, update, delete, create, drop, references, execute on *.* to 'deitel'@'localhost';` then we exit our session with the root account by typing `exit;`

We then start a session with the user account we just created `mysql -h localhost -u deitel -p` and inputting the password when prompted. Now we are reading to use the employees.sql script provided by Deitel to make our database. Unfortunately, the script provided by Deitel has some issues we had to correct before this process worked, and we will discuss those changes in a moment. Move the corrected script to the mysql.exe installation folder (C:\Program Files\MySQL\MySQL Server 8.0\bin on my machine) and in the command prompt where the deitel account mysql session has been opened, type source employees.sql and hit enter. Type `exit;` to close session.

```
Command Prompt

C:\Users\Admin>cd\

C:\>cd "Program Files\MySQL\MySQL Server 8.0\bin"

C:\Program Files\MySQL\MySQL Server 8.0\bin>mysql -u deitel -p
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.20 MySQL Community Server - GPL

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> source employees.sql
Query OK, 1 row affected, 1 warning (0.01 sec)

Database changed
Query OK, 0 rows affected (0.04 sec)

Query OK, 0 rows affected (0.03 sec)

Query OK, 0 rows affected (0.04 sec)

Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.05 sec)

Query OK, 0 rows affected (0.06 sec)

Query OK, 0 rows affected (0.11 sec)

Query OK, 0 rows affected (0.11 sec)

Query OK, 0 rows affected (0.11 sec)

Query OK, 0 rows affected (0.12 sec)

Query OK, 1 row affected (0.02 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.01 sec)

Query OK, 1 row affected (0.01 sec)

mysql> exit
Bye
```

Now the database has been created. As mentioned, we had to modify the employees.sql script provided by Deitel in a couple of ways. Here is the modified version of the script:

```
CREATE DATABASE IF NOT EXISTS employees
;

USE employees;

DROP TABLE IF EXISTS salariedEmployees
;

DROP TABLE IF EXISTS commissionEmployees
;

DROP TABLE IF EXISTS basePlusCommissionEmployees
;

DROP TABLE IF EXISTS hourlyEmployees
;

DROP TABLE IF EXISTS employees
;

CREATE TABLE employees (
      socialSecurityNumber varchar (30) NOT NULL,
      firstName varchar (30) NOT NULL,
      lastName varchar (30) NOT NULL,
      birthday date NOT NULL,
      employeeType varchar (30) NOT NULL,
      departmentName varchar (30) NOT NULL,
      PRIMARY KEY (socialSecurityNumber)
) ENGINE=INNODB
;

CREATE TABLE salariedEmployees (
      socialsecurityNumber varchar (30) NOT NULL,
      weeklySalary double NOT NULL,
      bonus double,
      INDEX (socialSecurityNumber),
      FOREIGN KEY (socialSecurityNumber) REFERENCES employees
(socialSecurityNumber)
) ENGINE=INNODB
;

CREATE TABLE commissionEmployees (
      socialSecurityNumber varchar (30) NOT NULL,
      grossSales int NOT NULL,
      commissionRate double NOT NULL,
      bonus double,
      INDEX (socialSecurityNumber),
      FOREIGN KEY (socialSecurityNumber) REFERENCES employees
(socialSecurityNumber)
) ENGINE=INNODB
;
```

```
CREATE TABLE basePlusCommissionEmployees (
      socialSecurityNumber varchar (30) NOT NULL,
      grossSales int NOT NULL,
      commissionRate double NOT NULL,
      baseSalary double NOT NULL,
      bonus double,
      INDEX (socialSecurityNumber),
      FOREIGN KEY (socialSecurityNumber) REFERENCES employees
(socialSecurityNumber)
) ENGINE=INNODB
;

CREATE TABLE hourlyEmployees (
      socialSecurityNumber varchar (30) NOT NULL,
      hours int NOT NULL,
      wage double NOT NULL,
      bonus double,
      INDEX (socialSecurityNumber),
      FOREIGN KEY (socialSecurityNumber) REFERENCES employees
(socialSecurityNumber)
) ENGINE=INNODB
;

INSERT INTO employees VALUES ('111-11-1111', 'John', 'Smith', '1945-1-2',
'salariedEmployee', 'R&D')
;


INSERT INTO employees VALUES ('222-22-2222', 'Sue', 'Jones', '1961-2-3',
'commissionEmployee', 'SALES')
;

INSERT INTO employees VALUES ('333-33-3333', 'Bob', 'Lowis', '1958-10-5',
'basePlusCommissionEmployee', 'SALES')
;

INSERT INTO employees VALUES ('444-44-4444', 'Karen', 'Price', '1972-5-25',
'hourlyEmployee', 'HR')
;

INSERT INTO salariedEmployees VALUES ('111-11-1111', 2013.67, 0)
;

INSERT INTO commissionEmployees VALUES ('222-22-2222', 10100, 0.05, 0)
;

INSERT INTO basePlusCommissionEmployees VALUES ('333-33-3333', 5000, 0.04,
300, 0)
;

INSERT INTO hourlyEmployees VALUES ('444-44-4444', 30, 35.5, 0)
;
```

First, the modifications. Originally at the end of every TABLE declaration the script had TYPE=INNODB. This is deprecated in the newest version of MySQL, and ENGINE=INNODB

must be used. Secondly, the original script used reals instead of doubles (doubtlessly for efficiency's sake), but since our database is small, we have replaced these with doubles, as we are more familiar with doubles and this will make our Java app easier to program.

The first line `CREATE DATABASE IF NOT EXISTS employees` is pretty explicit: it creates a new database called employees if one does not already exist. Then the next line `USE employees;` indicates that the commands that follow will apply to the newly creates employees database. The next 5 Drop Table lines delete any preexisting tables `salariedEmployees, commissionEmployees, basePlusCommissionEmployees, hourlyEmployees`, and `employees`. This is handy if we made a mistake, modified the original database in some way, etc., and want to start from scratch (which happened several times during the course of working on this problem).

Now we start creating our tables. We create our main table (parent to our other tables), which has 6 columns: socialSecurityNumber (variable size string, max 30 characters, cannot be null), firstName (variable size string, max 30 characters, cannot be null), lastName (variable size string, max 30 characters, cannot be null), birthday (date format (SQL assures that date matches required YYYY-MM-DD format with correct day limits for months (i.e. no February 30th, no April 31st, etc) and if not throws an SQL error), employeeType (variable size string, max 30 characters, cannot be null), and departmentName (variable size string, 30 character max, cannot be null). Lastly we specify that the PRIMARY KEY for this table is socialSecurityNumber, since it uniquely identifies each employee record. Lastly, we specify `ENGINE=INNODB`, since MySQL uses InnoDB storage engine.

Now we create salariedEmployees table, with 3 columns: socialSecurityNumber (as above), weeklySalary (double, cannot be null), and bonus (a double). We create an INDEX using socialSecurityNumber, which will speed up data retrieval when using this column in the query predicate. Lastly, we create a FOREIGN KEY on socialSecurityNumber to link the socialSecurityNumber entry on this table with the socialSecurityNumber entry on the employees table (i.e. `FOREIGN KEY (socialSecurityNumber) REFERENCES employees (socialSecurityNumber)`). As with the first table, we specify `ENGINE=INNODB` on this table as well.

We then create the commissionEmployees table, with 4 columns: socialSecurityNumber (as above), grossSales (int, cannot be null), commissionRate (double, cannot be null), and bonus (a double). We create an INDEX using socialSecurityNumber, which will speed up data retrieval when using this column in the query predicate. Lastly, we create a FOREIGN KEY on socialSecurityNumber to link the socialSecurityNumber entry on this table with the socialSecurityNumber entry on the employees table (i.e. `FOREIGN KEY (socialSecurityNumber) REFERENCES employees (socialSecurityNumber)`). As with the first table, we specify `ENGINE=INNODB` on this table as well.

We then create the basePlusCommissionEmployees table, with 5 columns. As with the commission table, we have socialSecurityNumber (as above), grossSales (int, cannot be null), commissionRate (double, cannot be null), and bonus (a double). Obviously, the difference with

regular commission employees is that this table will have also has a baseSalary column (double, cannot be null). We create an INDEX using socialSecurityNumber, which will speed up data retrieval when using this column in the query predicate. Lastly, we create a FOREIGN KEY on socialSecurityNumber to link the socialSecurityNumber entry on this table with the socialSecurityNumber entry on the employees table (i.e. `FOREIGN KEY (socialSecurityNumber) REFERENCES employees (socialSecurityNumber)`). As with the first table, we specify `ENGINE=INNODB` on this table as well.

Finally, we create the hourlyEmployees table, with 4 columns: socialSecurityNumber (as above), hours (int, cannot be null), wage (double, cannot be null), and bonus (a double). We create an INDEX using socialSecurityNumber, which will speed up data retrieval when using this column in the query predicate. Lastly, we create a FOREIGN KEY on socialSecurityNumber to link the socialSecurityNumber entry on this table with the socialSecurityNumber entry on the employees table (i.e. `FOREIGN KEY (socialSecurityNumber) REFERENCES employees (socialSecurityNumber)`). As with the first table, we specify `ENGINE=INNODB` on this table as well.

Now that we have created all five tables, we then populate them with employee records. We put 4 employees into the employees table with the commands

```
INSERT INTO employees VALUES ('111-11-1111', 'John', 'Smith', '1945-1-2',
'salariedEmployee', 'R&D');
INSERT INTO employees VALUES ('222-22-2222', 'Sue', 'Jones', '1961-2-3',
'commissionEmployee', 'SALES');
INSERT INTO employees VALUES ('333-33-3333', 'Bob', 'Lowis', '1958-10-5',
'basePlusCommissionEmployee', 'SALES');
INSERT INTO employees VALUES ('444-44-4444', 'Karen', 'Price', '1972-5-25',
'hourlyEmployee', 'HR');
```

The arguments are in the same order the columns were set in the employees table, so we have socialSecurityNumber, firstName, lastName, birthdate, employeeType, and departmentName, in that order. We then use specific commands for each subtable to put each employee in the subtable according to their employeeType:

```
INSERT INTO salariedEmployees VALUES ('111-11-1111', 2013.67, 0);
INSERT INTO commissionEmployees VALUES ('222-22-2222', 10100, 0.05, 0);
INSERT INTO basePlusCommissionEmployees VALUES ('333-33-3333', 5000, 0.04,
300, 0);
INSERT INTO hourlyEmployees VALUES ('444-44-4444', 30, 35.5, 0);
```

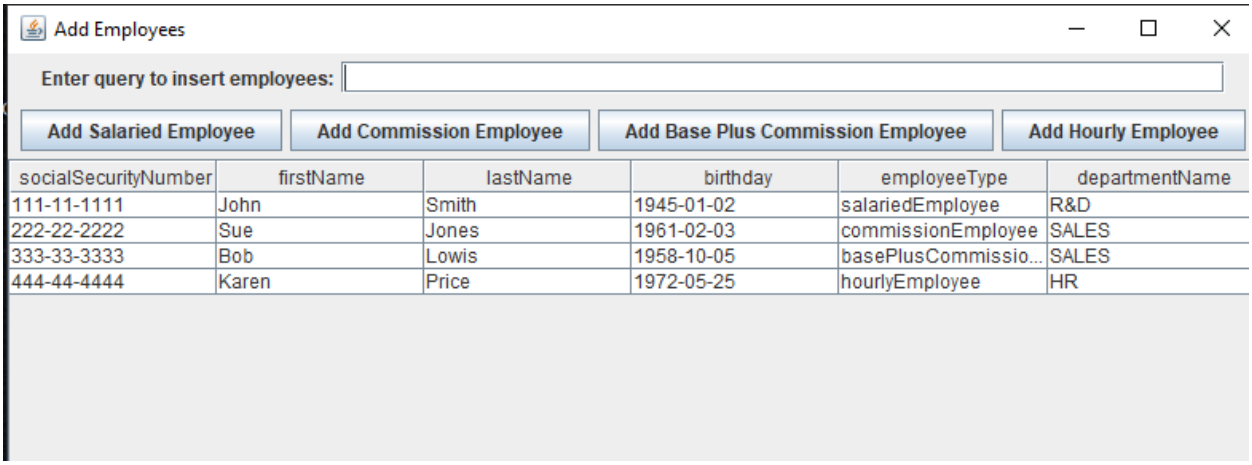
So now employee 111-11-1111 will be in the salaried employee table with 2013.67 weeklySalary and 0 bonus, employee 222-22-2222 will be in the commissionEmployees table with grossSales 10100, commissionRate 0.05, and bonus 0, etc.

At this point, our database is complete, we can then access it, manipulate/change it, add to it, etc., via Java, by installing the proper driver in Java. This is done by downloading the newest driver at https://dev.mysql.com/downloads/connector/j/5.1.html, unzipping the file and taking the mysql-connector-java-5.1.49-bin.jar file (the latest is version 5.1.49 at the time of this writing) and coping to the JDK \jre\lib\ext directory (this is C:\Program

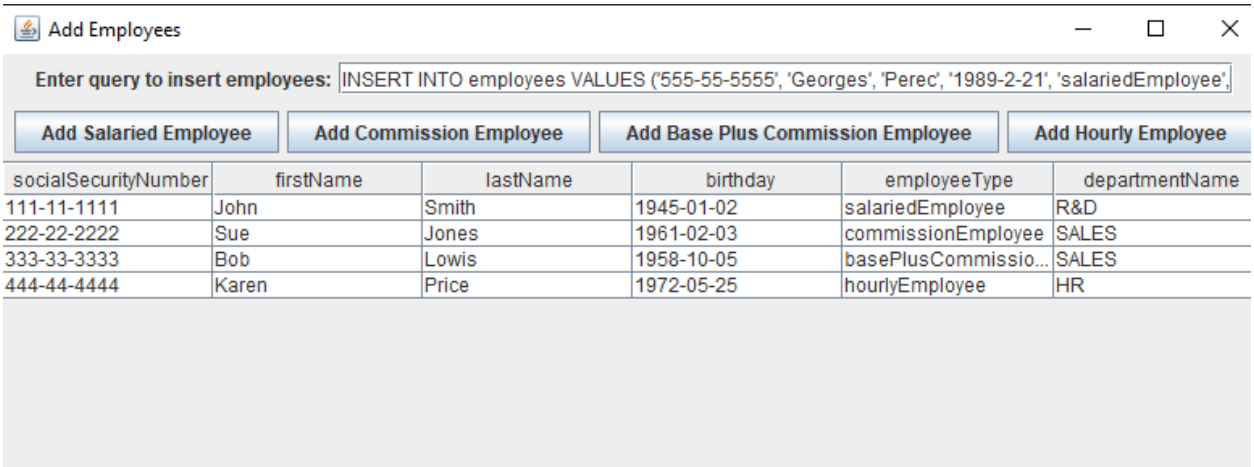
Files\Java\jdk1.8.0_221\jre\lib\ext on my machine). Once this is done, the driver is installed locally on our machine.

Now, lastly, in the program itself, in the application class that will access our database, we need to connect to our local database, so we first specify its location with the line String url = "jdbc:mysql://localhost/employees" you will see this in our Java file momentarily, then we connect with the line connection = DriverManager.getConnection(url, "deitel", "deitel"), where 'deitel' and 'deitel' are the user and pass for the user accont we created earlier to access the database (eventually this connection will need to be closed, which happens eventually in the shutdown method which consists of the line connection.close). This finalizes the basic set-up needed to make the application work, so let us finally take a look at what the application looks like.

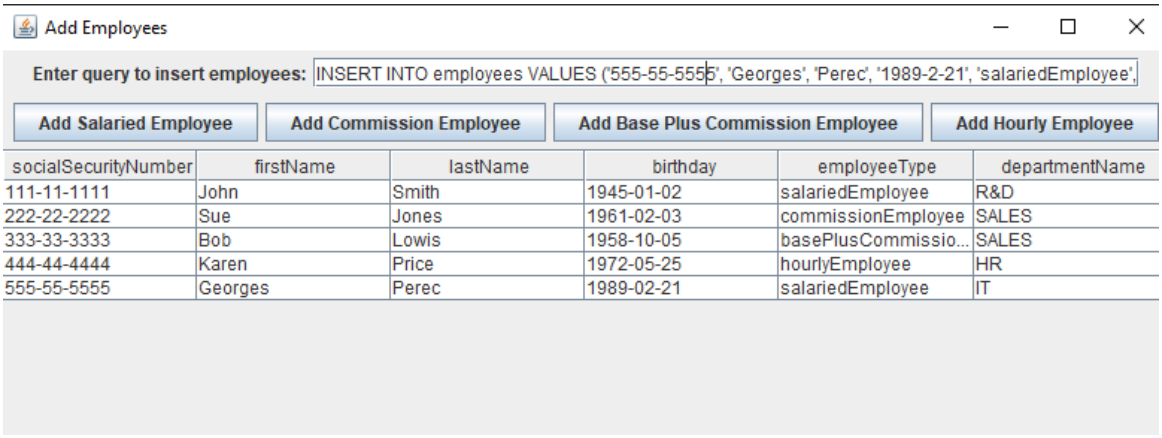
When we first run it, it displays the employees table with the 4 employees we added in the SQL script:



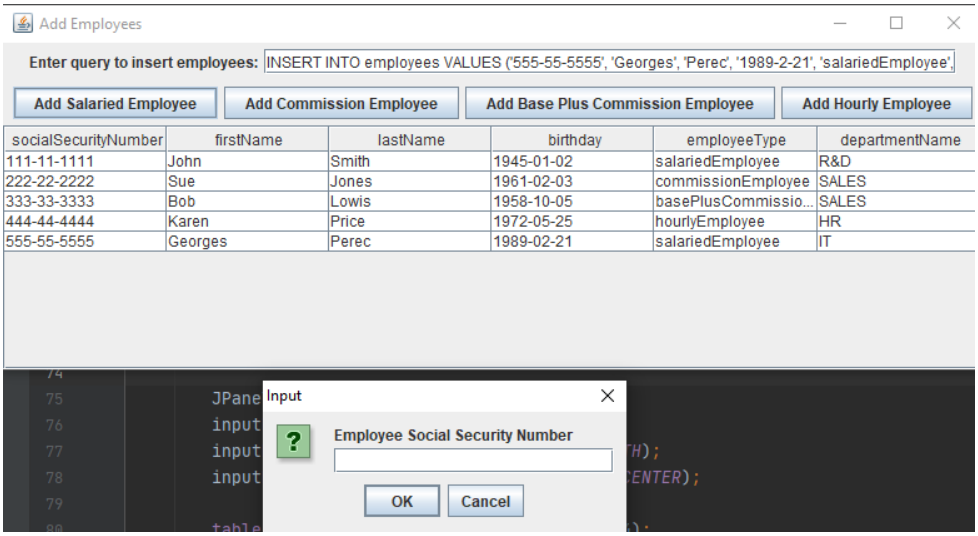
If we want to say, add another employee to the table employees, we can do by typing the appropriate query into the text field:

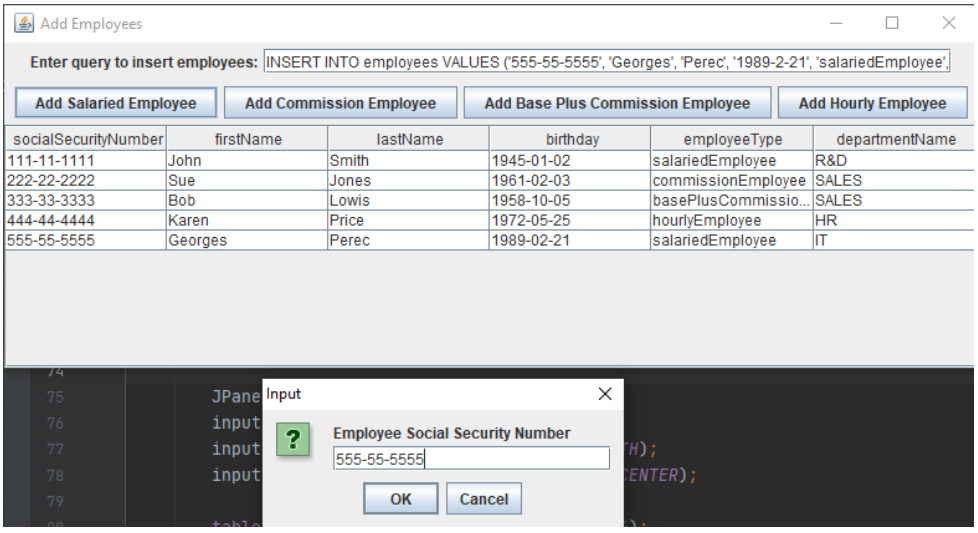


We hit enter and the new employee appears in the table:

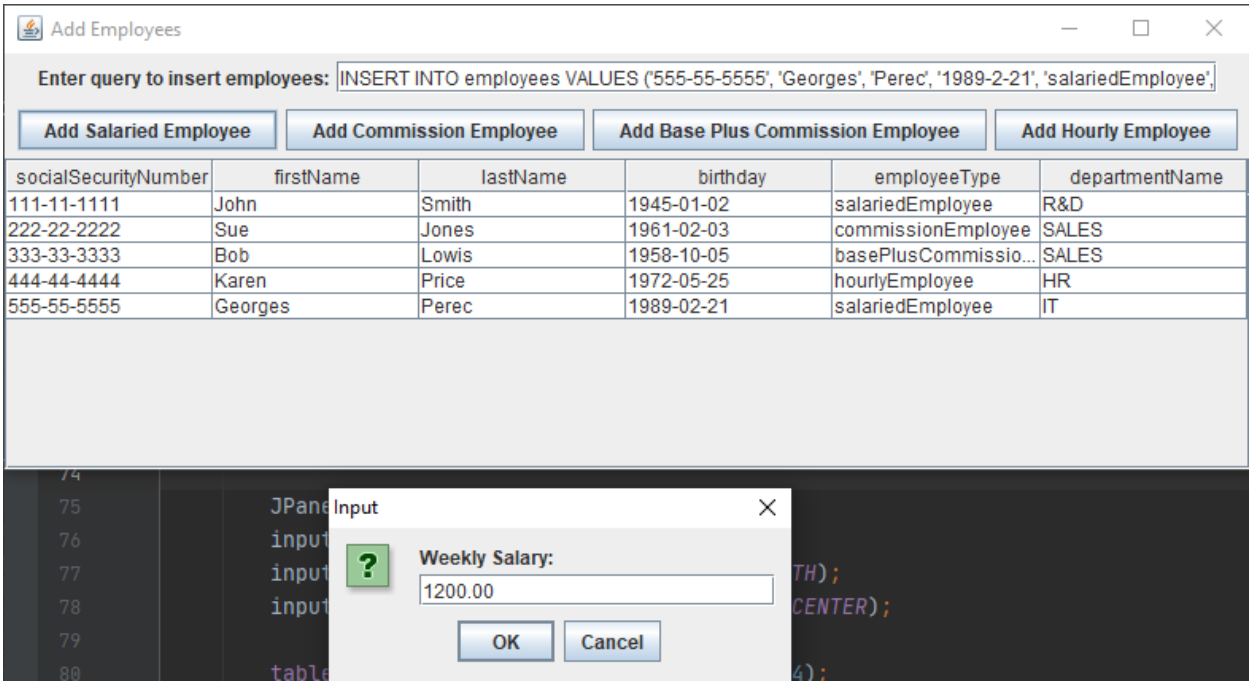


Of course, we will also want to add this employee to the salariedEmployees table. We can do that by clicking the addSalariedEmployee button, and a JOptionPane will appear with an InputDialog to specify the socialSecurityNumber of the new record:

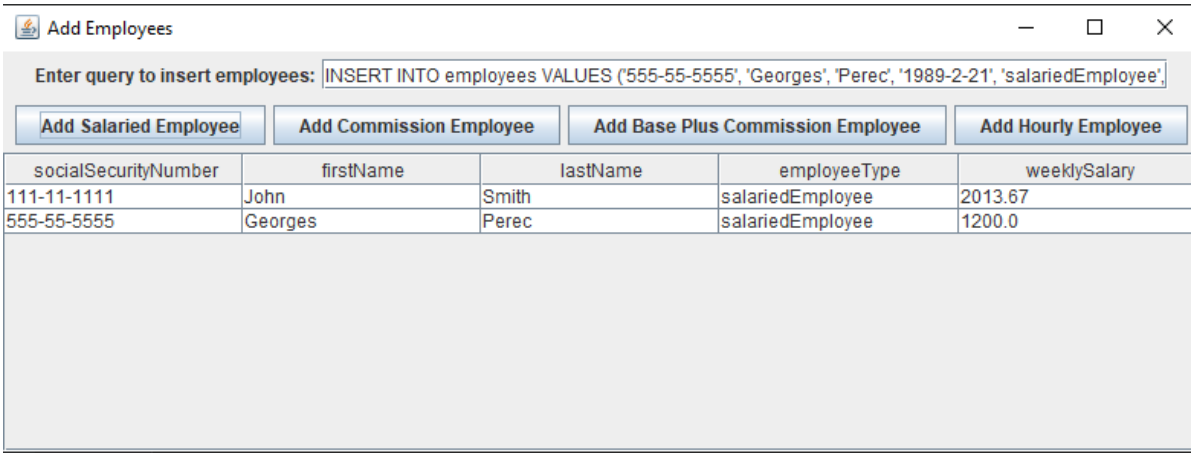




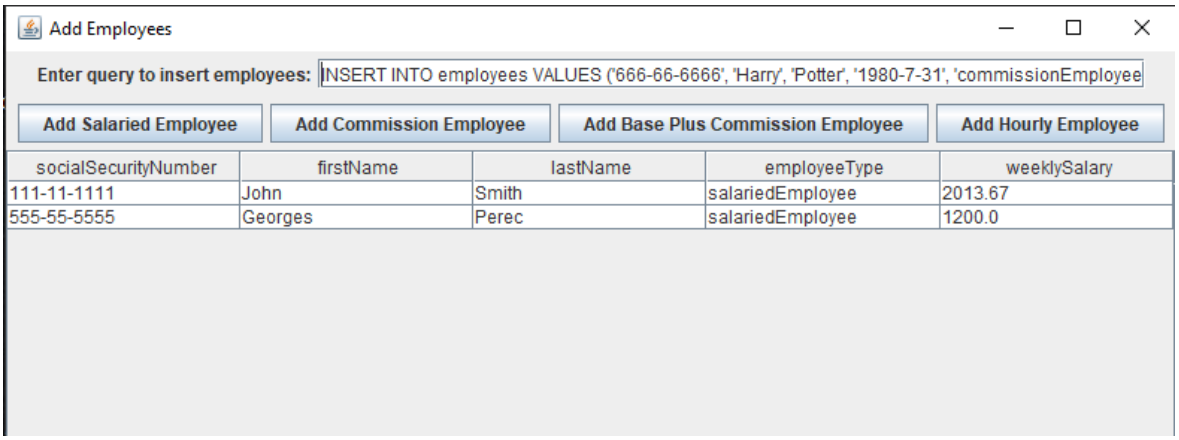
We then click ok and a new JOptionPane appears requesting we input weeklySalary for this employee:



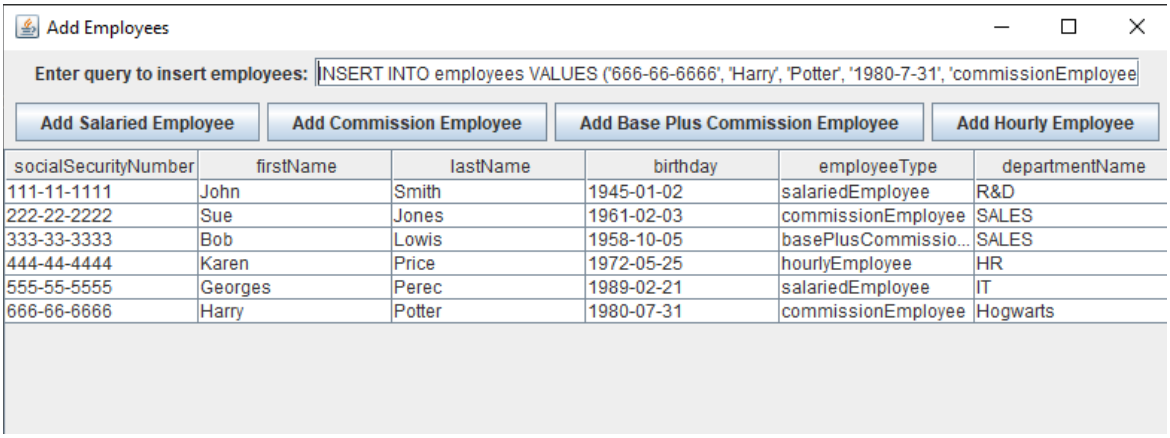
We click OK, and the salariedEmployee table will appear, showing that the employee has been added to this table:



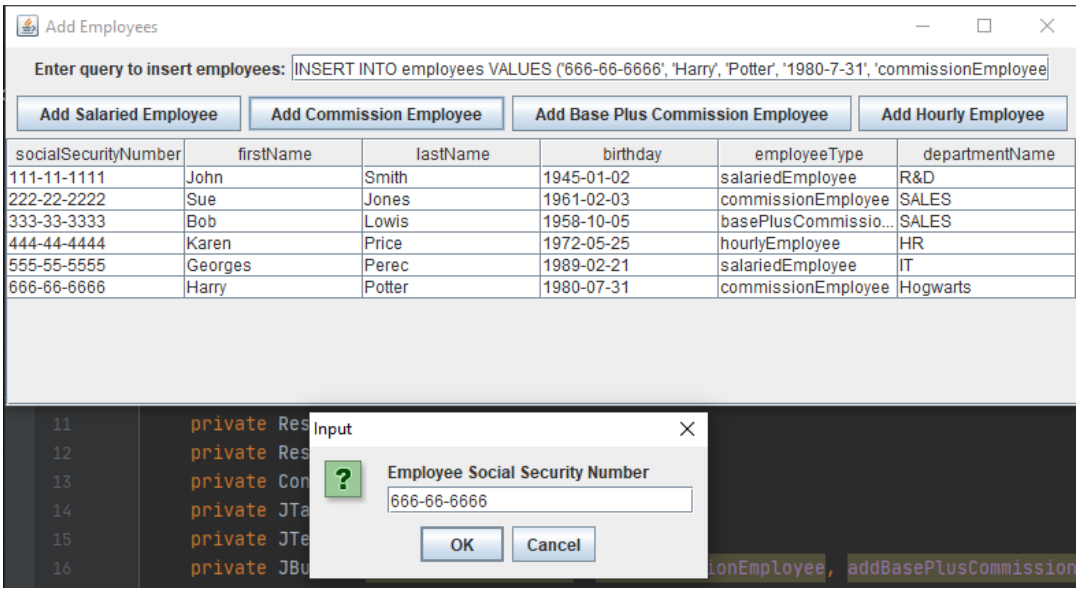
Notice that we also display information from the employees table in the GUI, in order to make it more readable (we will go over the SQL code to make this happen momentarily). Now let us try adding employees of different types; first, a commission employee. We type the SQL query in the text box…



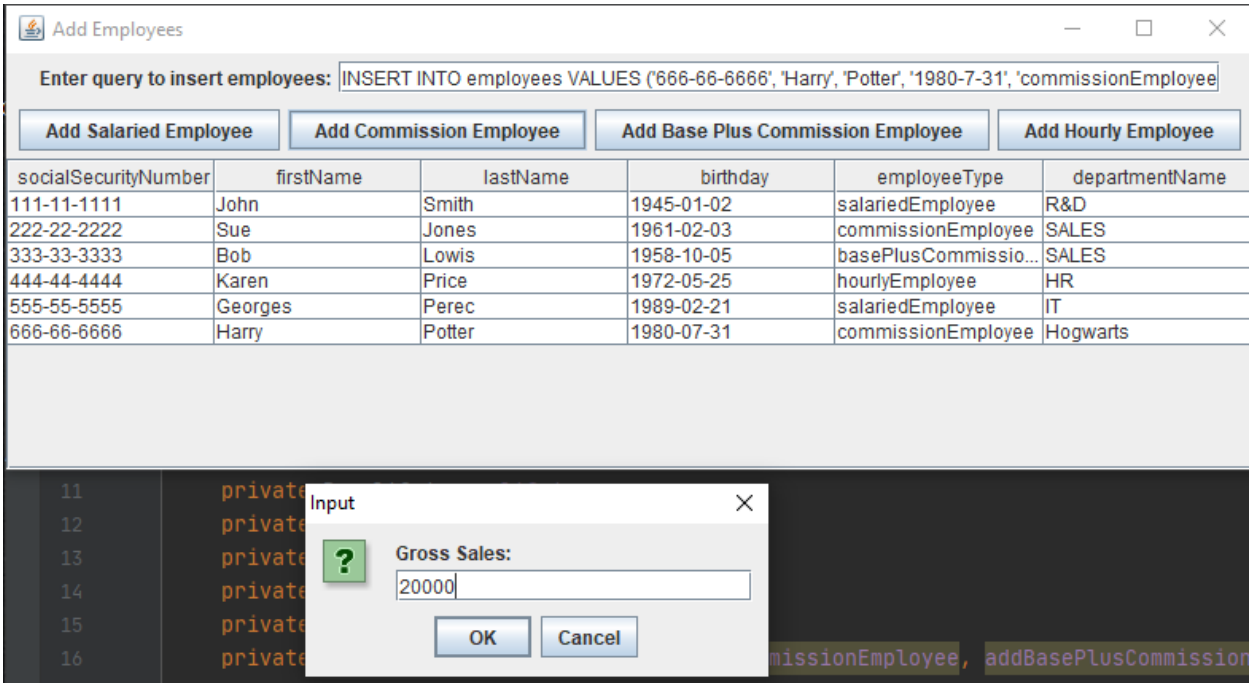
…and click enter, and the original employees table appears, showing the new employees has been added:



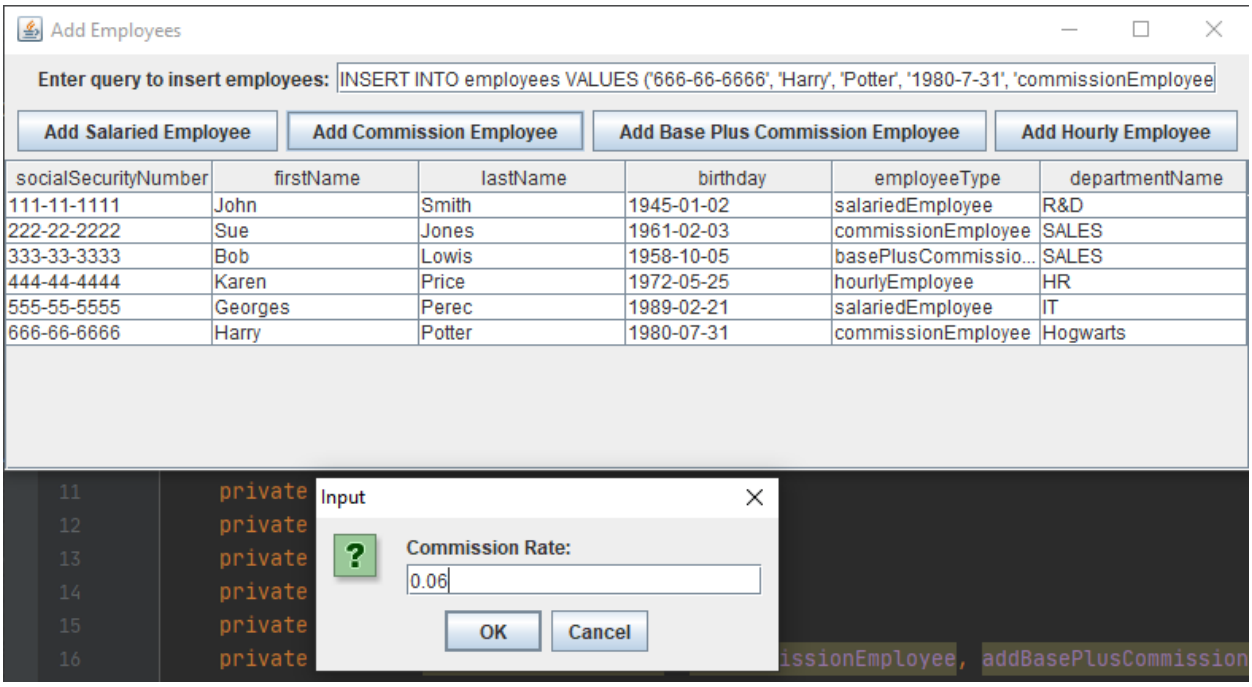
Now, we also need to add this employee to the commissionEmployees table, which we do by clicking the addCommissionEmployee button. As before, a JOptionPane appears:



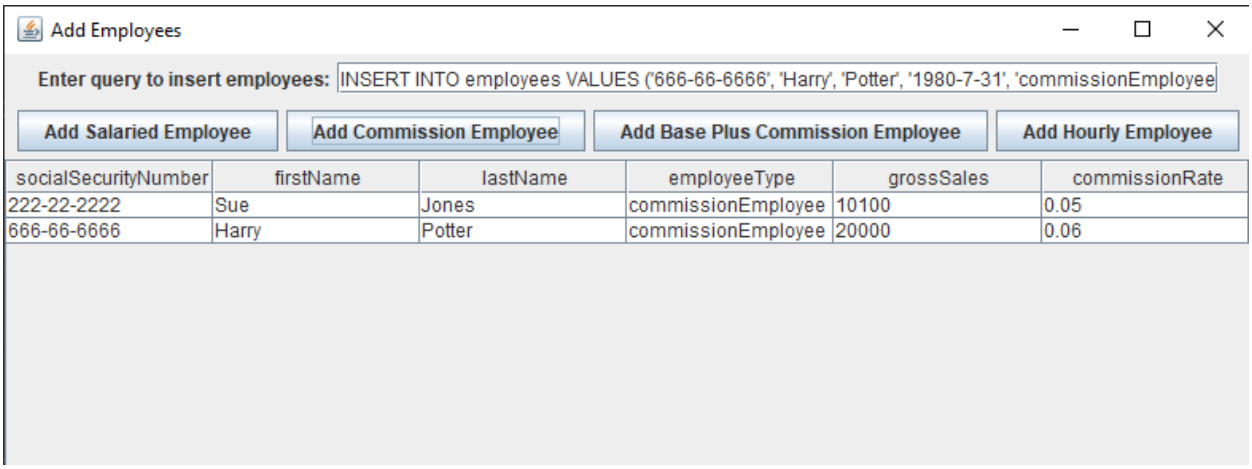
After typing the socialSecurityNumber of the employee, we click ok, and another JOptionPane appears with an InputDialog asking we input grossSales:



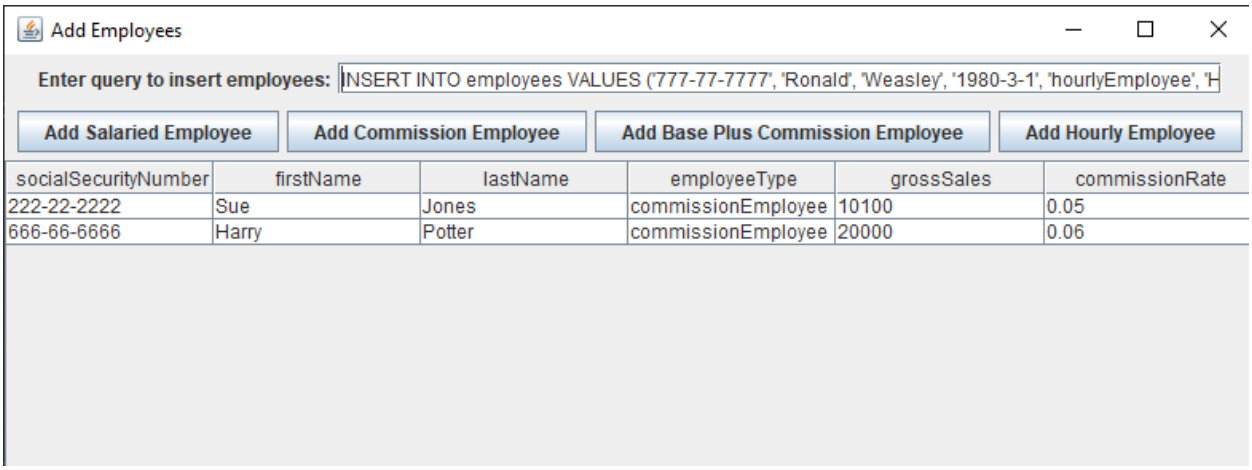
Lastly, a JOptionPane appears with an InputDialog requesting commissionRate:



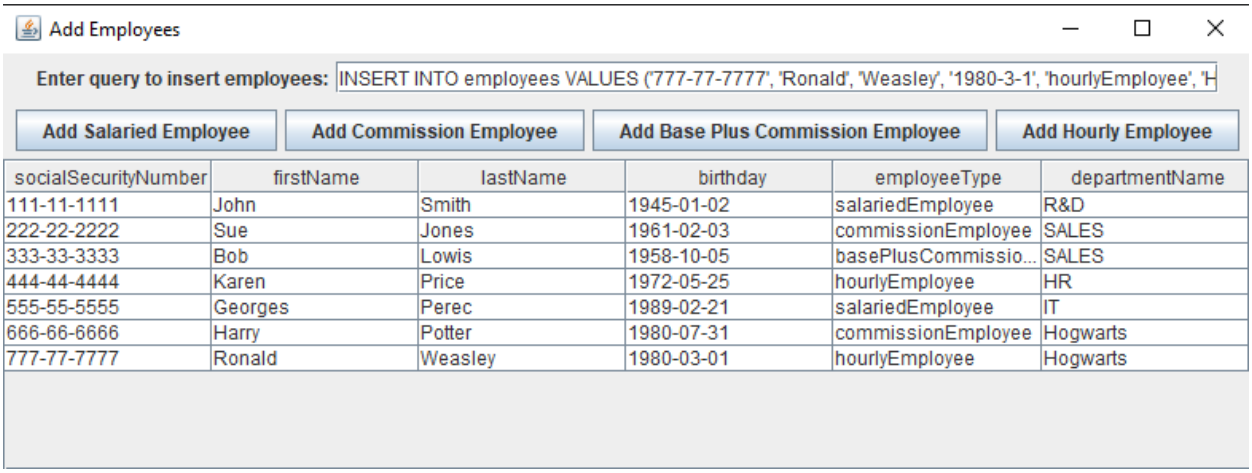
After we click ok, the app displays the commissionEmployees table, complete with the new employee we just added:



Now let us add an hourlyEmployee. We type an appropriate query in the query text field:

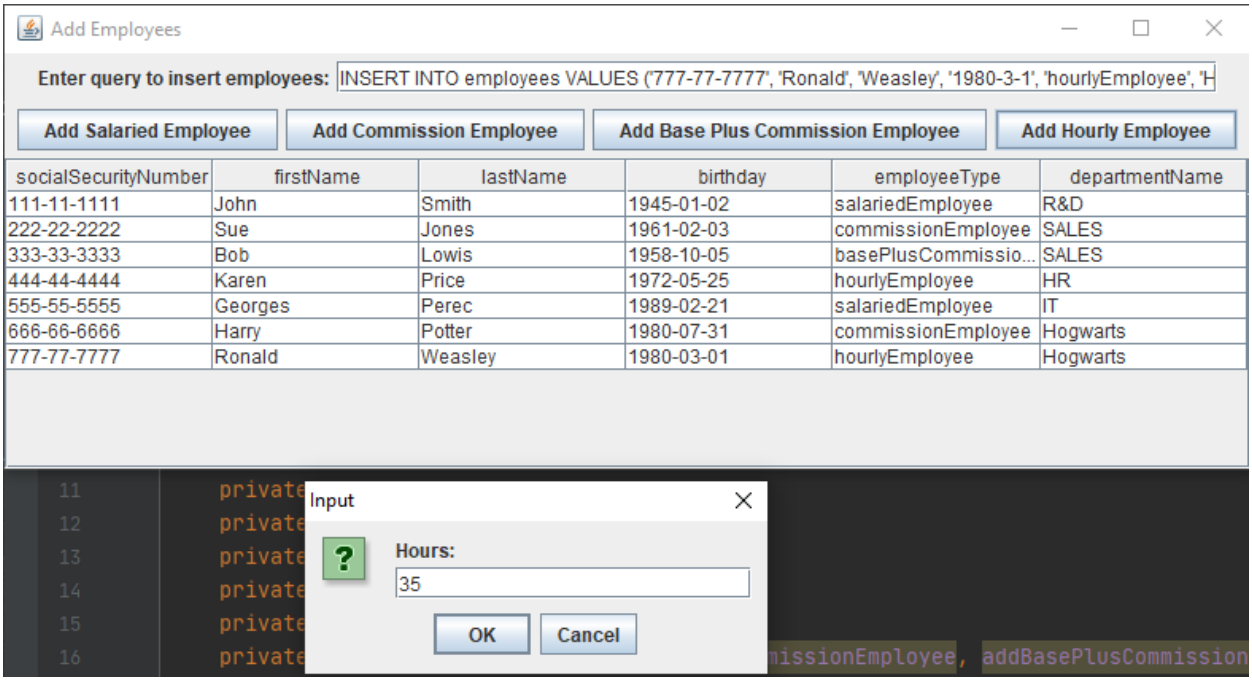


Hit Enter and we see the employees table again, with the new employee added:

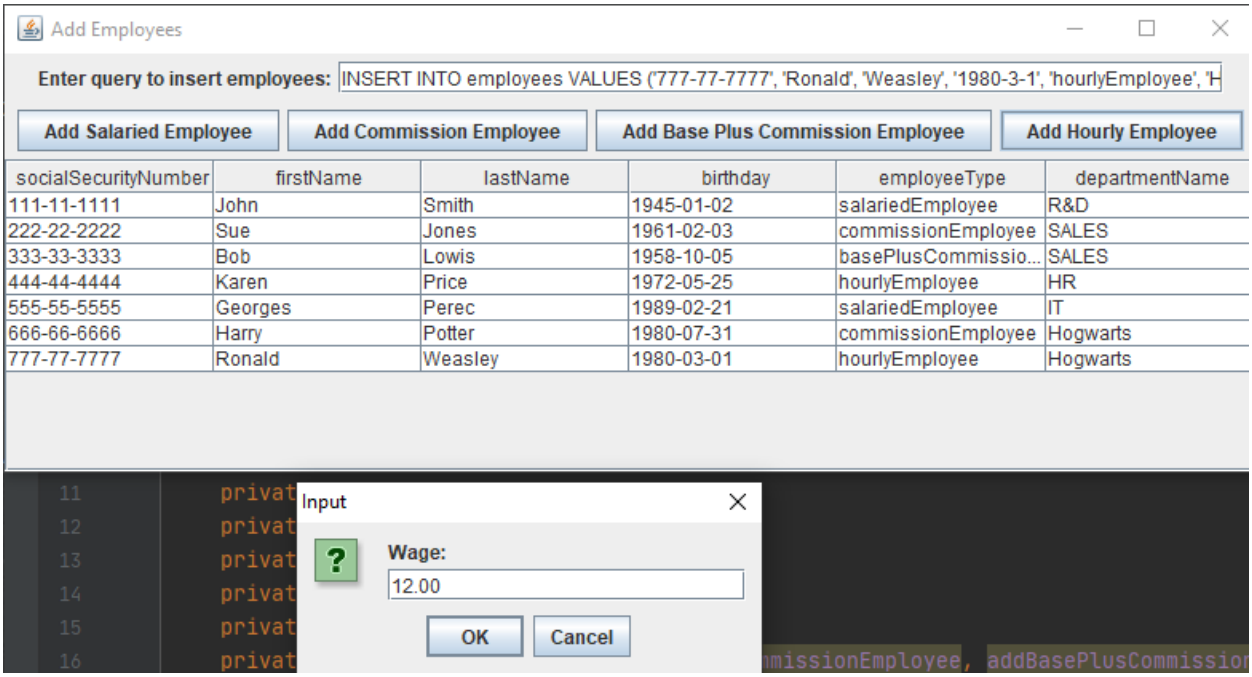


Now, we need to add this employee to the hourlyEmployees table, so we click the addHourlyEmployee button, and a JOptionPane appears with InputDialog requesting

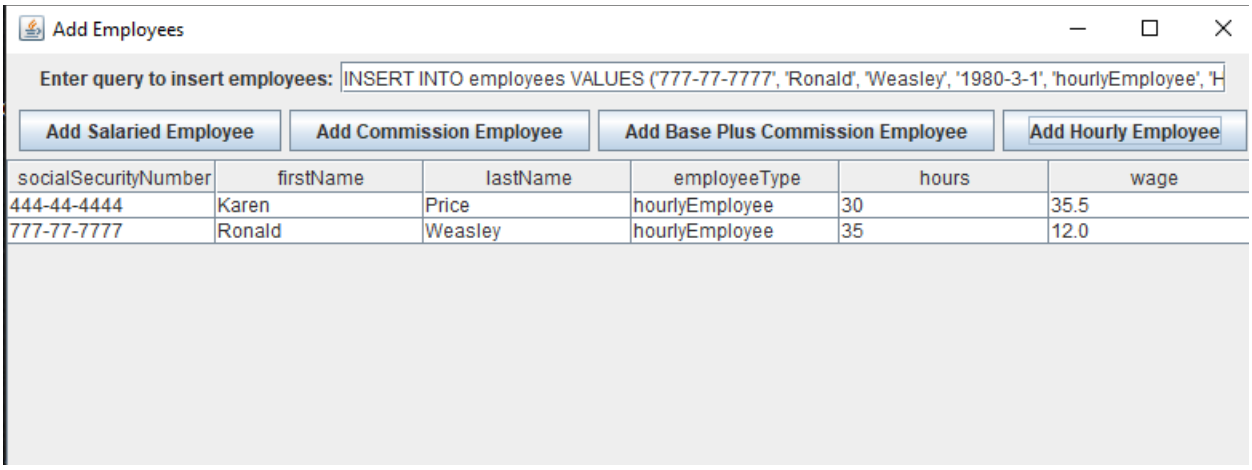
socialSecurityNumber (we skip the image since it's the same as the others). Then a JOptionPane appears requesting the number of hours worked:



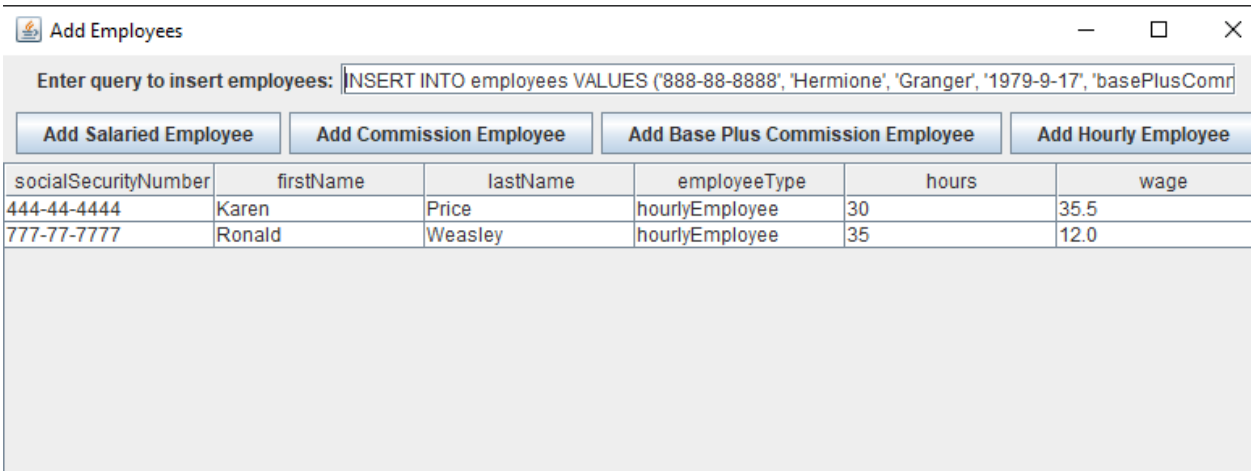
We click OK, and a final JOptionPane appears with InputDialog requesting the wage:



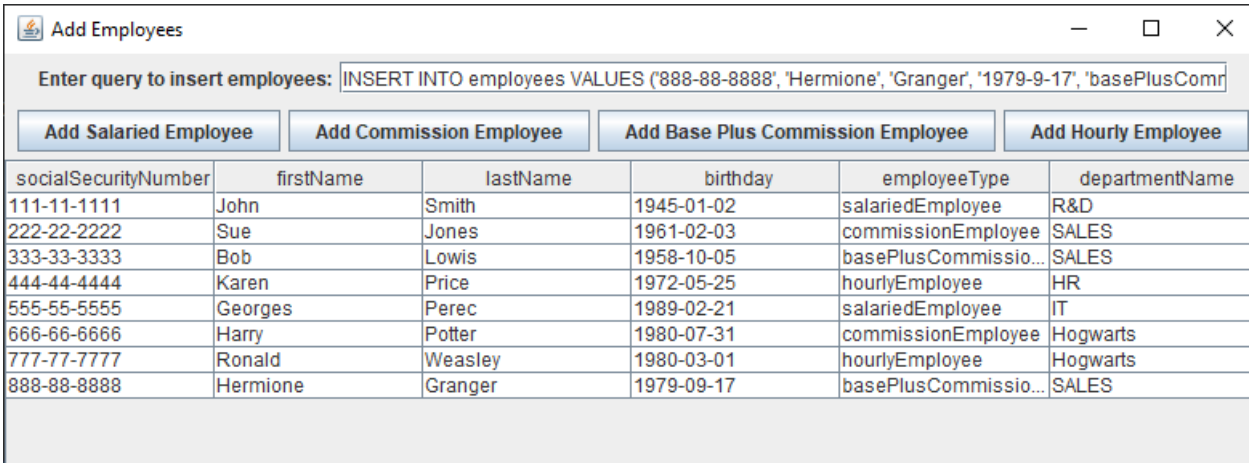
We click OK, and the app displays the hourlyEmployees table with the employee added:



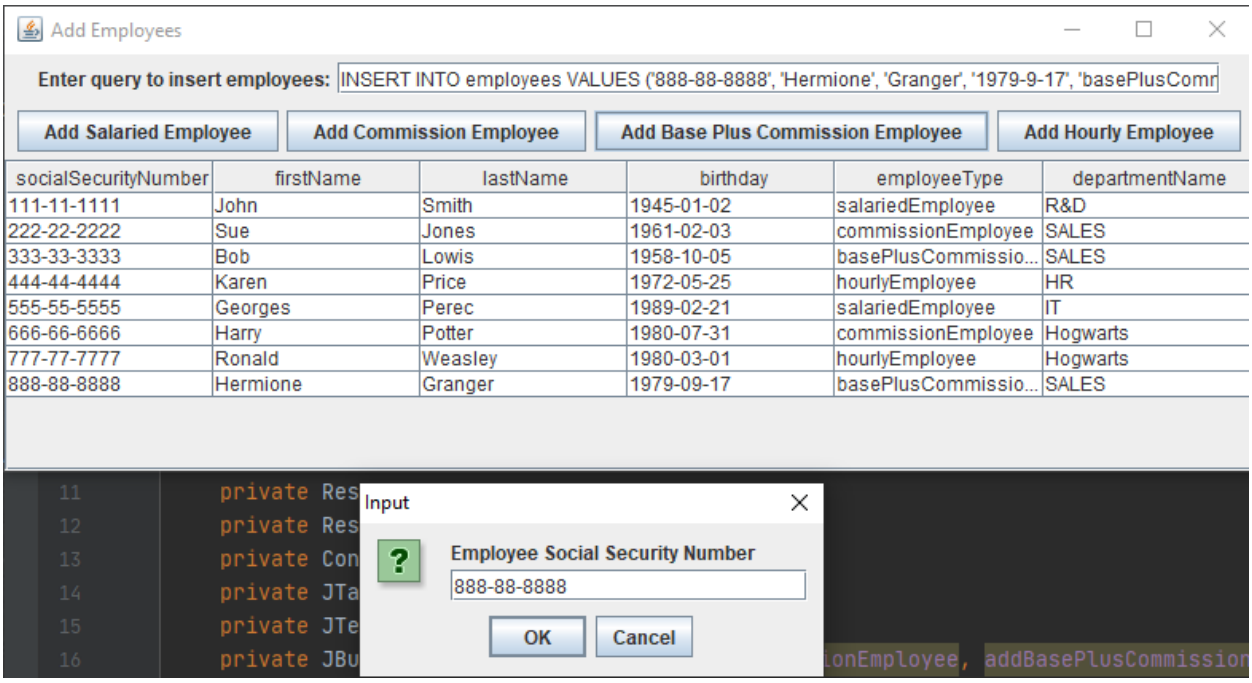
Now, lastly, why don't we try adding a basePlusCommissionEmployee. We type the appropriate query in the input JTextField…



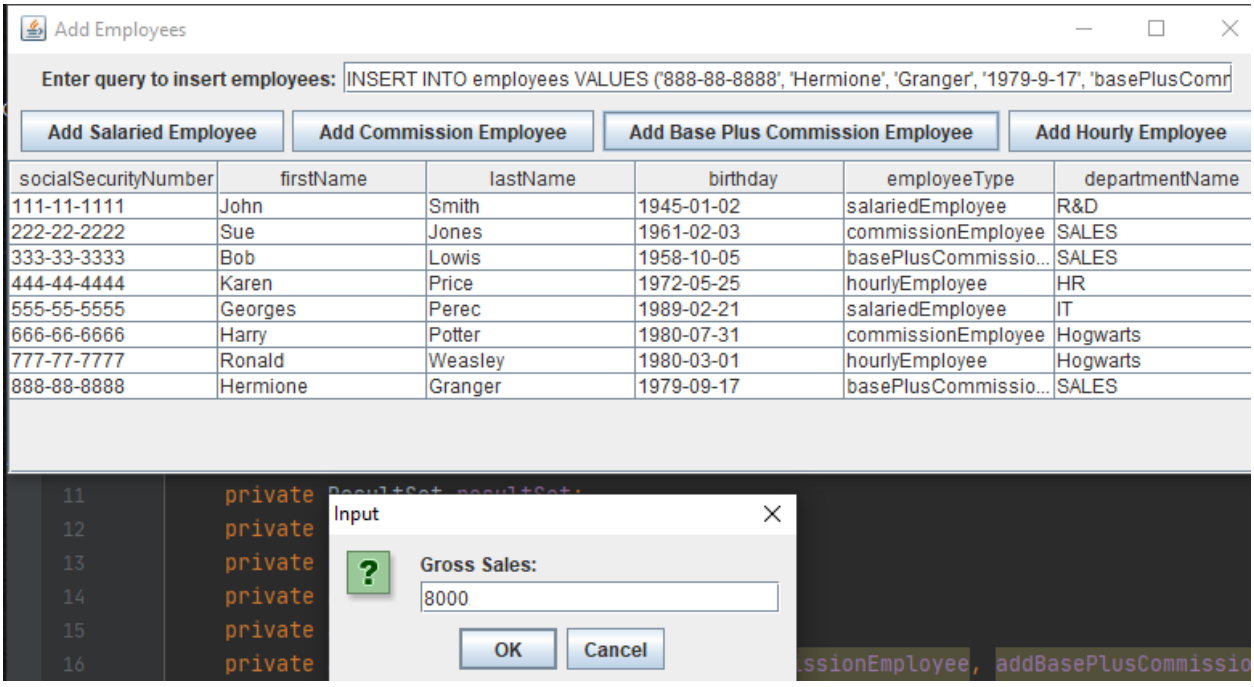
…and hit Enter, and the app displays the employees table with the new employee added:



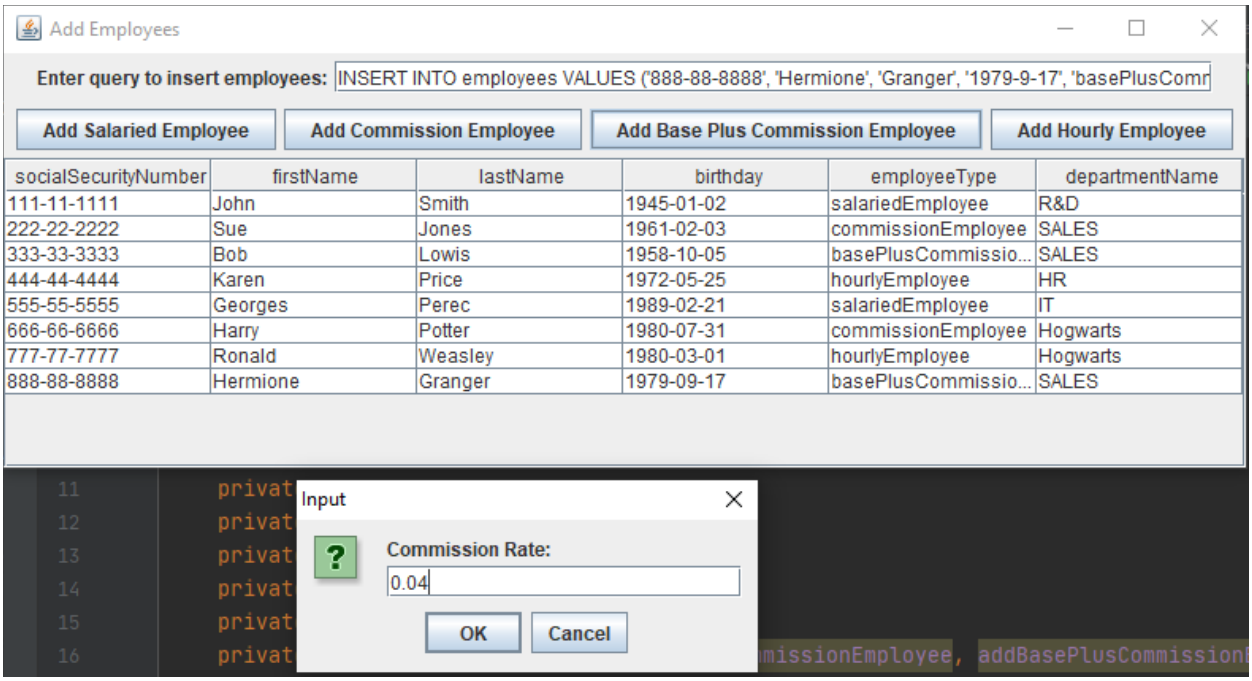
Now, we need to add the new employee to the basePlusCommissionEmployees table, so the click the addBasePlusCommissionEmployee button and the familiar JOptionPane appears with prompt for socialSecurityNumber:



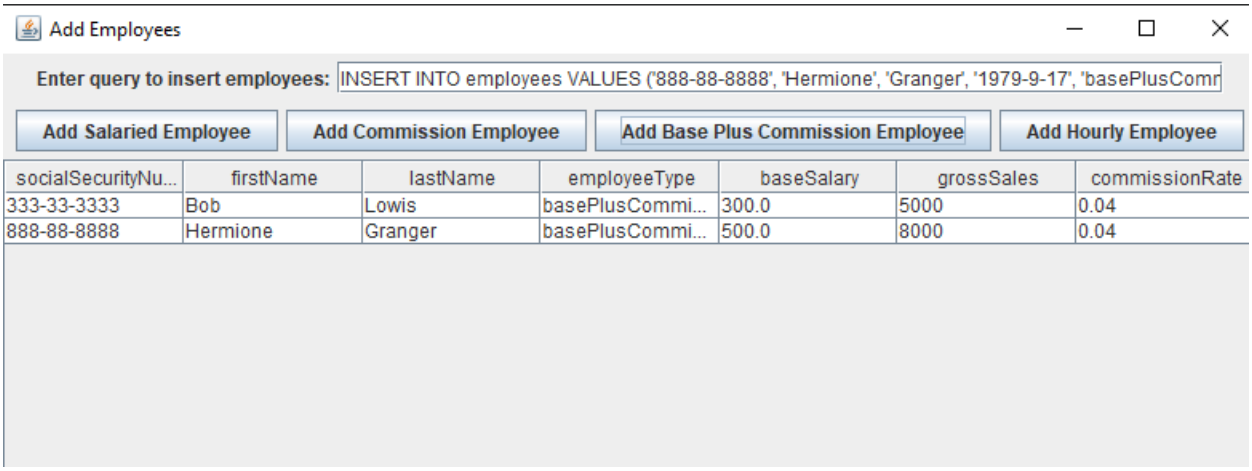
We click OK, and a JOptionPane appears with an InputDialog prompt for grossSales:



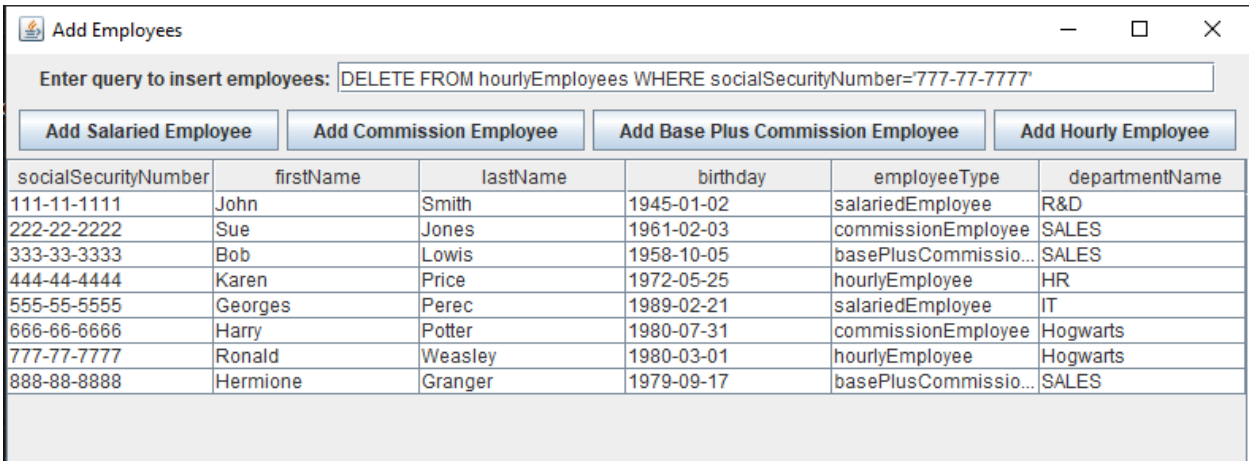
We click OK, and a JOptionPane appears with an InputDialog for commissionRate:



We click OK, and lastly a JOptionPane appears with an InputDialog prompt requesting baseSalary. We forgot to take a screen capture, but we input 500.00 and the app shows the basePlusCommissionEmployees table with the employee added:

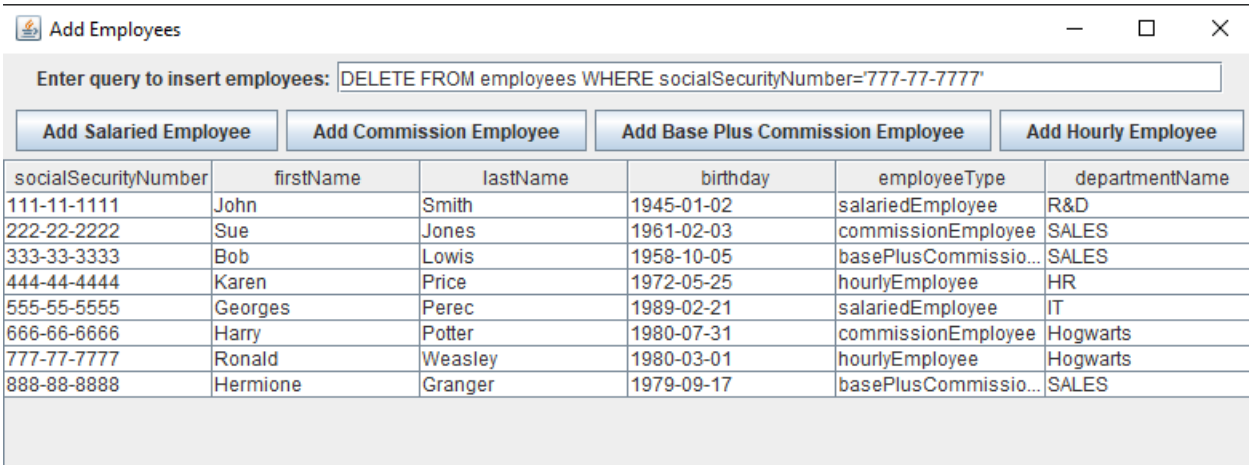


Now, other SQL queries will also work in the query JTextField. Suppose Ronald Weasley got fired and we want to delete him from our database. Before we can remove him from the parent table (employees), we have to remove him any child tables, so our first query must remove him from the hourlyEmployees table:



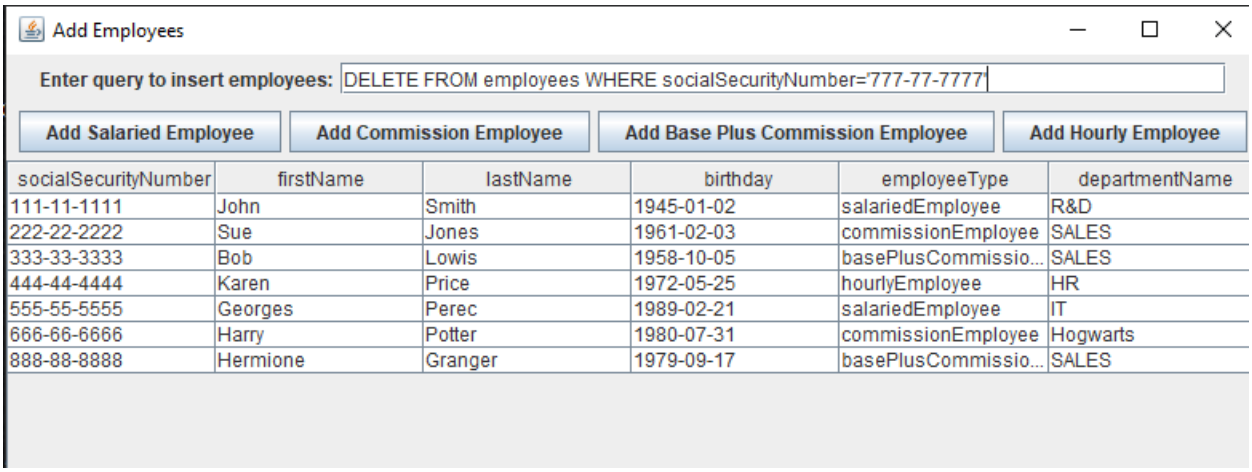
socialSecurityNumber	firstName	lastName	birthday	employeeType	departmentName
111-11-1111	John	Smith	1945-01-02	salariedEmployee	R&D
222-22-2222	Sue	Jones	1961-02-03	commissionEmployee	SALES
333-33-3333	Bob	Lowis	1958-10-05	basePlusCommissio...	SALES
444-44-4444	Karen	Price	1972-05-25	hourlyEmployee	HR
555-55-5555	Georges	Perec	1989-02-21	salariedEmployee	IT
666-66-6666	Harry	Potter	1980-07-31	commissionEmployee	Hogwarts
777-77-7777	Ronald	Weasley	1980-03-01	hourlyEmployee	Hogwarts
888-88-8888	Hermione	Granger	1979-09-17	basePlusCommissio...	SALES

We hit Enter, and then we type a similar query removing him from the employees table:



socialSecurityNumber	firstName	lastName	birthday	employeeType	departmentName
111-11-1111	John	Smith	1945-01-02	salariedEmployee	R&D
222-22-2222	Sue	Jones	1961-02-03	commissionEmployee	SALES
333-33-3333	Bob	Lowis	1958-10-05	basePlusCommissio...	SALES
444-44-4444	Karen	Price	1972-05-25	hourlyEmployee	HR
555-55-5555	Georges	Perec	1989-02-21	salariedEmployee	IT
666-66-6666	Harry	Potter	1980-07-31	commissionEmployee	Hogwarts
777-77-7777	Ronald	Weasley	1980-03-01	hourlyEmployee	Hogwarts
888-88-8888	Hermione	Granger	1979-09-17	basePlusCommissio...	SALES

We hit enter, and we see the employee disappears:



socialSecurityNumber	firstName	lastName	birthday	employeeType	departmentName
111-11-1111	John	Smith	1945-01-02	salariedEmployee	R&D
222-22-2222	Sue	Jones	1961-02-03	commissionEmployee	SALES
333-33-3333	Bob	Lowis	1958-10-05	basePlusCommissio...	SALES
444-44-4444	Karen	Price	1972-05-25	hourlyEmployee	HR
555-55-5555	Georges	Perec	1989-02-21	salariedEmployee	IT
666-66-6666	Harry	Potter	1980-07-31	commissionEmployee	Hogwarts
888-88-8888	Hermione	Granger	1979-09-17	basePlusCommissio...	SALES

So the app seems to work as intended. Now, let us take a look at the code. It is too long to just copy and paste it all and examine it all at once, so let us look at it by sections:

```
// 23.4 EmployeesSQL.java
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class EmployeesSQL extends JFrame {
    private Connection connection;
    private Statement statement;
    private ResultSet resultSet;
    private ResultSetMetaData rsMetaData;
    private Container container;
    private JTable table;
    private JTextField input;
    private JButton addSalariedEmployee, addCommissionEmployee,
addBasePlusCommissionEmployee, addHourlyEmployee;
```

Now, we import the typical awt and swing packages, since we will use various graphics objects in our app. Since we will use SQL, we obviously must import java.sql. Now, we will use a JFrame as the backbone of our GUI, so we need to include extends JFrame in our class declaration. We also declare all the more complex objects we are going to use later. We will take about these when we get to them in the program. Now, the first thing in the code is the constructor method for our class, which does much of the heavy lifting, since our main method…

```
public static void main( String[] args ) {
    final EmployeesSQL application = new EmployeesSQL();
    application.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    application.shutDown();
                    System.exit(0);
                }
            }
    );
}
```

…simply creates an EmployeesSQL object, adds a window listener to the object, which listens for the action of the user closing the window, and shuts down the application if the user closes the window (exiting with code 0, i.e. successful). Before we get to the constructor, let us briefly look at the shutDown method:

```
public void shutDown() {
    try {
        connection.close();
    }
    catch (SQLException sqlException) {
        System.err.println( "Unable to disconnect" );
        sqlException.printStackTrace();
    }
}
```

As discussed earlier, shutDown closes the connection we create to the database in the EmployeesSQL constructor method, or, if the connection fails to disconnect, throws an SQL error, prints the StackTrace, and returns to the main method.

Simple enough, now let us look at the constructor method, which handles most of the program operation:

```
public EmployeesSQL() {

        super("Add Employees");

//This specifies the location of the employees database on our local machine
        String url = "jdbc:mysql://localhost/employees";

// we must start by connecting to the database
        try {
            connection = DriverManager.getConnection(url, "deitel",
"deitel");
        }
        catch (SQLException sqlException) {
            System.err.println("Unable to connect");
            sqlException.printStackTrace();
            System.exit(1); // terminate program
        }
```

First, we call the constructor of the parent JFrame class to create a new JFrame and name it “Add Employees”. As mentioned earlier in the section where we discuss setup, we must connect to the database stored locally on our machine. For MySQL, this location is jdbc:mysql://localhost/[database name], we assign this location (in our specific case, “jdbc:mysql://localhost/employees) to the String url, which we then use to establish a connection to the data base with the line connection. = DriverManager.getConnection(url, ‘deitel’, ‘deitel’), where deitel is the user we gave access to the MySQL database and ‘deitel’ is the password for this account. Of course, to avoid accidental crashing, it is best to envelop all of this in a try…catch statement, where, in case of an SQLException, the program doesn’t crash but instead will display “Unable to connect”, print the StackTrace, and exit with code 1 (error). This will let us know where the problem occurred.

Now, the next part of the constructor builds the GUI for the app:

```
// set up GUI
        JPanel topPanel = new JPanel();
        topPanel.setLayout(new FlowLayout());
        topPanel.add(new JLabel("Enter query to insert employees:"));

        input = new JTextField(50);
        topPanel.add(input);
        input.addActionListener(
                new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        addEmployee(input.getText());
                    }
```

```
                }
        );
```

We first create a JPanel called ‘topPanel’ to hold the label and JTextField we use for the little query area in the top of our app. We give it a FlowLayout to arrange our next two items logically, from left to right. We first add a JLabel with instructions “Enter query to insert employees:”, and then create a 50-character wide JTextField called ‘input’ and add it to the JPanel. We then add an ActionListener to the JTextField, which, if the user hits enter while the text field is active, calls the addEmployee method with whatever is in the text field as an argument. Let us take a look at this method:

```
private void addEmployee(String query) {
    try {
        statement = connection.createStatement();
        statement.executeUpdate(query);
        getTable();
    }
    catch (SQLException sqlException) {
        sqlException.printStackTrace();
    }
}
```

So this method creates a statement, takes the query string from the text field and attempts to execute the string as an SQL query. If the query is invalid or not well-formatted, an SQLException is thrown and the StackTrace is printed (this is useful to avoid program crashing in such a case). Lastly, the getTable method is called, which displays the employees table in the main window. We will look at the details of getTable here in a moment, but for now let us get back to the EmployeesSQL constructor.

The next section of the constructor sets up the four buttons that add employees to the various child tables:

```
// set up four buttons that allow user to add employees of different types
        JPanel centerPanel = new JPanel();
        centerPanel.setLayout(new FlowLayout());

        addSalariedEmployee = new JButton("Add Salaried Employee");
        addSalariedEmployee.addActionListener(new ButtonHandler());

        addCommissionEmployee = new JButton("Add Commission Employee");
        addCommissionEmployee.addActionListener(new ButtonHandler());

        addBasePlusCommissionEmployee = new JButton("Add Base Plus Commission
Employee");
        addBasePlusCommissionEmployee.addActionListener(new ButtonHandler());

        addHourlyEmployee = new JButton("Add Hourly Employee");
        addHourlyEmployee.addActionListener(new ButtonHandler());
```

We create a JPanel names ‘centerPanel’ to assemble the four buttons, and set the JPanel’s Layout to be a FlowLayout, which means the buttons will simply be added left to right. Recall that we

declared all these buttons at the beginning of the class, so we don't have to redeclare them here. First we create the addSalariedEmployee button with the text "Add Salaried Employee" and add an actionListener to it which we will define further later, then we create a addCommissionEmployee button with the text "Add Commission Employee" and add an actionListener to it which we will define further later, then we create an addBasePlusCommissionEmployee button with the text "Add Base Plus Commission Employee" and add an actionListener to it which we will define further later, and lastly we create an addHourlyEmployee button with the text "Add Hourly Employee" and add an actionListener to it which we will define further later.

Next, we add these four buttons in order to our centerPanel:

```
// add four buttons to centerPanel
        centerPanel.add(addSalariedEmployee);
        centerPanel.add(addCommissionEmployee);
        centerPanel.add(addBasePlusCommissionEmployee);
        centerPanel.add(addHourlyEmployee);
```

Then we create a JPanel names 'inputPanel' to hold the two JPanels we just created. We set its layout to BorderLayout, and we add the topPanel on top (BorderLayout.NORTH) and the centerPanel in the center (BoderLayout.Center).

Now, if you recall, the bottom part of the app GUI is a table displaying the employee records. We create a JTable with 4 rows and 4 columns, and create a container with a BorderLayout to hold the inputPane from earlier and the table we just create, then add the inputPane on top (BorderLayout.NORTH) and the table on bottom (BorderLayout.CENTER):

```
table = new JTable(4,4);

container = getContentPane();
container.setLayout(new BorderLayout());
container.add(inputPanel, BorderLayout.NORTH);
container.add(table, BorderLayout.CENTER);
```

Note that this table is redefined and resized depending on the needs of the table we need to display, so its current dimensions will not be the final dimension. Now, the end of the constructor method is as follows:

```
    getTable();

    setSize(800,300);
    setVisible(true);
} // end constructor method of EmployeesSQL
```

As mentioned, getTable basically updates the JTable, and we will describe this method momentarily. SetSize sets the size of our JFrame to be 800x300 pixels, and setVisible(true) displays the JFrame with all its graphical elements. Now, let us take a look at getTable:

```
private void getTable() {
    try {
```

```
        statement = connection.createStatement();
        resultSet = statement.executeQuery("SELECT * FROM employees");
        displayResultSet(resultSet);
    }
    catch (SQLException sqlException) {
        sqlException.printStackTrace();
    }
}
```

We first create a statement connected with our database, then we execute the query `SELECT * FROM employees`, which selects all the records in the employees database and returns them as a resultSet, which is a type of object that is a table of data representing the appropriate rows from a database, with a provided iterator to iterate through each row and metadata (i.e. column names, etc) associated with each data entry. This is, of course, enveloped in a try...catch statement so, in case of an SQL error, we can throw an exception and print the StackTrace rather than have the program crash. The last line under the try block is displayResultSet(resultSet), which is a method to process and display our result set in the table part of our app. Let us look at this method:

```
private void displayResultSet(ResultSet rs) throws SQLException {
// position to first record
        boolean moreRecords = rs.next();

// if there are no records, display a message
        if (!moreRecords) {
            JOptionPane.showMessageDialog(this,"ResultSet contained no
records");
            return;
        }
        Vector columnHeads = new Vector();
        Vector rows = new Vector();

        try {
// get column heads
            ResultSetMetaData rsmd = rs.getMetaData();

            for (int i = 1; i <= rsmd.getColumnCount(); ++i)
                columnHeads.addElement(rsmd.getColumnName(i));

// get row data
            do {
                rows.addElement(getNextRow(rs, rsmd));
            } while (rs.next());

// display table with ResultSet contents
            table = new JTable(rows, columnHeads);
            JScrollPane scroller = new JScrollPane(table);
            container.remove(1);
            container.add(scroller, BorderLayout.CENTER);
            container.validate();
        } // end try

        catch (SQLException sqlException) {
            sqlException.printStackTrace();
```

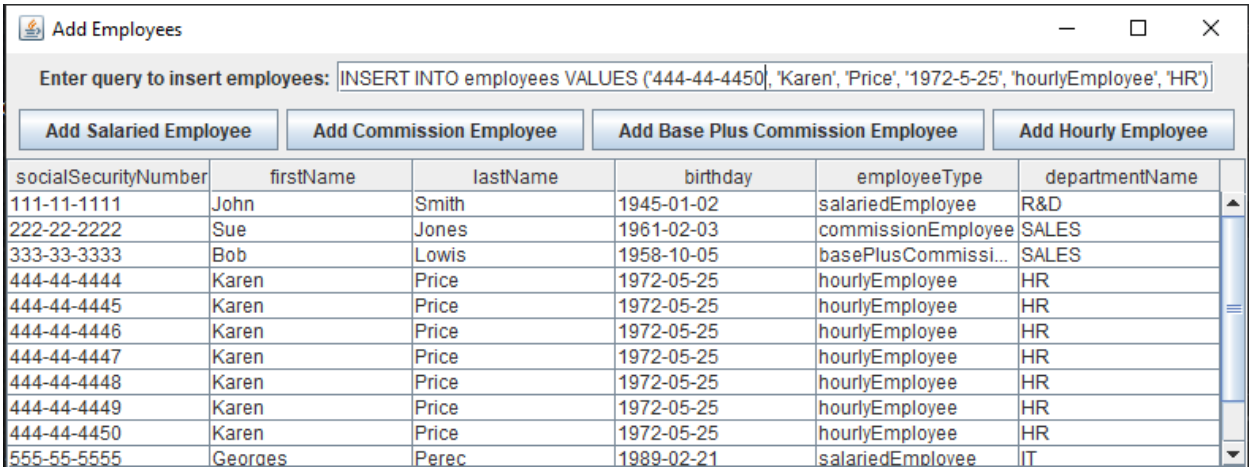
```
        }
    } // end method displayResultSet
```

Initially, we test to see if there are any records in the result set by checking is rs.next returns true or false. If it returns false, then !moreRecords will be true and we branch into the first if statement, displaying a JOptionPane message stating "ResultSet contained no records" and we return. If there is at least one record, then we skip the if statement and create Vectors to hold column head info and the different rows (we include import java.util.* at the beginning of the file to use the Vector class).

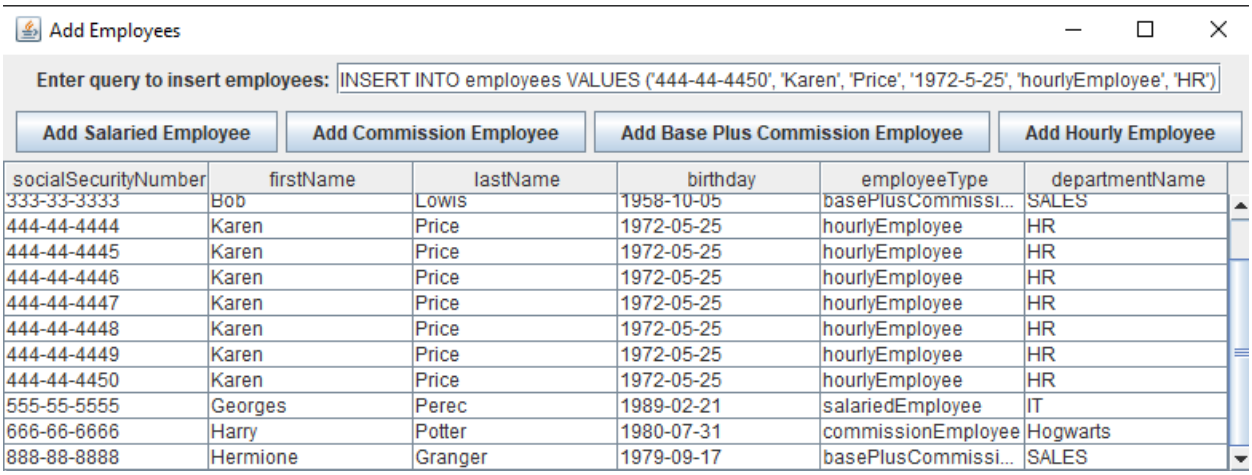
Now, if there is at least one record, we need to get our data from the ResultSet. We use a try…catch statement to catch SQLException errors just in case there are any issues with the ResultSet (in which case, we print the StackTrace and return). Now, first, we get the column heads by looking at the metadata in the ResultSet. We call getMetaData on the result set to get a ResultSetMetaData object, which is essentially an object with a number of useful get methods to get the information we need about ResultSet (types, labels, properties, number of columns, etc.). We use a for loop to cycle through the columns and add each column name to our columnHeads vector.

We then use the rare do…while loop to add rows to our rows Vector. Since we know there is at least one row, we add that row (with associated metadata) to our rows Vector, check to see if there are any more rows in the ResultSet, and another row if there is, check for any more rows, and so on, until rs.next() returns false. Now, the process of adding rows to our rows Vector uses a method called getNextRow, which we will describe in detail in a moment, but for now just know that it casts each element in a row of the ResultSet as the appropriate type before returning it as a vector.

Finally, we create a new JTable with our rows and columnHeads Vectors (which, due to the Vector, Vector constructor for JTable, will automatically give it to correct dimensions), then feed the table into the JScrollPane constructor to make a JScrollPane called scroller, which is helpful if we add many employee records to our database so we can scroll through them like this:



socialSecurityNumber	firstName	lastName	birthday	employeeType	departmentName
111-11-1111	John	Smith	1945-01-02	salariedEmployee	R&D
222-22-2222	Sue	Jones	1961-02-03	commissionEmployee	SALES
333-33-3333	Bob	Lowis	1958-10-05	basePlusCommissi...	SALES
444-44-4444	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4445	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4446	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4447	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4448	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4449	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4450	Karen	Price	1972-05-25	hourlyEmployee	HR
555-55-5555	Georges	Perec	1989-02-21	salariedEmployee	IT



socialSecurityNumber	firstName	lastName	birthday	employeeType	departmentName
333-33-3333	Bob	Lowis	1958-10-05	basePlusCommissi...	SALES
444-44-4444	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4445	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4446	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4447	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4448	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4449	Karen	Price	1972-05-25	hourlyEmployee	HR
444-44-4450	Karen	Price	1972-05-25	hourlyEmployee	HR
555-55-5555	Georges	Perec	1989-02-21	salariedEmployee	IT
666-66-6666	Harry	Potter	1980-07-31	commissionEmployee	Hogwarts
888-88-8888	Hermione	Granger	1979-09-17	basePlusCommissi...	SALES

We then remove the table that was previously in the bottom location (index 1), and add our JScrollPane to its spot in the Layout (BorderLayout.CENTER). We then call container.validate() which makes the container lay out its subcomponents again, which is necessary since we modified the table.

Now, that is the end of the displayResultSet method, but we skipped over the details of the getNextRow method, so let us take a look at it now:

```
private Vector getNextRow(ResultSet rs, ResultSetMetaData rsmd) throws
SQLException {
    Vector currentRow = new Vector();

    for (int i = 1; i <= rsmd.getColumnCount(); ++i)
        switch(rsmd.getColumnType(i)) {
            case Types.VARCHAR:
            case Types.LONGVARCHAR:
                currentRow.addElement(rs.getString(i));
                break;
            case Types.INTEGER:
                currentRow.addElement(new Long(rs.getLong(i)));
                break;
            case Types.DOUBLE:
                currentRow.addElement(new Float(rs.getDouble(i)));
                break;
            case Types.DATE:
                currentRow.addElement(rs.getDate(i));
                break;
            default:
                System.out.println("Type was: " + rsmd.getColumnTypeName(i));
        }
    return currentRow;
} // end method getNextRow
```

This method accepts a ResultSet and ResultSetMetaData object as arguments, and returns a vector containing the data in the current row in the ResultSet (in the calling method, the ResultSet has an iterator which is cycling through the different rows of the ResultSet). Now, we start by creating a currentRow Vector, then for each column in the row, we check the metadata to

get the column type, and for Varchar and Longvarchar we simply retrieve the element from the ResultSet as a String and add it to the currentRow. If the column type is Integer, we retrieve the element from the ResultSet as a Long, cast it as a Long, and add it to the currentRow. If the column type is Double, we retrieve it as a Double, cast it as a Float, and add it to the currentRow. Lastly, if the column type is a Dat, we retrieve it as a Date and add it to the currentRow. If there is a type mismatch or some other type is provided, then we retrieve the intended type in the column metadata and print out the necessary type to the console for the user to read. In essence, we are making sure each element in a resultSet row corresponds to the type dictated by the column metadata, adding it to a Vector, and returning this Vector corresponding to the row in the ResultSet. The calling method displayResultSet does this for every row, so that the rows Vector eventual contains a Vector with the row data for each row.

Ok, this covers the basic process of displaying the employees table and adding employees, but what about adding employees to other tables with the 4 JButtons we created? At the end of our code, we have a ButtonHandler designed to handle this process:

```
// inner class ButtonHandler handles button event
private class ButtonHandler implements ActionListener {

    public void actionPerformed(ActionEvent event) {
        String socialSecurityNumber = JOptionPane.showInputDialog("Employee
Social Security Number");
        String insertQuery = "", displayQuery = "";
```

First we declare a single ButtonHandler class that will handle ActionEvents from our 4 JButtons. If a JButton is clicked, no matter which button it is, we display a JOptionPane with InputDialog prompt “Enter Social Security Number”, and store the result as socialSecurityNumber. We then create blank Strings insertQuery and displayQuert, which will be modified depending on which button was click. First, the addSalariedEmployee button:

```
// add salaried employee to table salariedEmployee
            if (event.getSource() == addSalariedEmployee) {
                double weeklySalary =
Double.parseDouble(JOptionPane.showInputDialog("Weekly Salary:"));
                insertQuery = "INSERT INTO salariedEmployees VALUES ( '" +
                        socialSecurityNumber + "', '" + weeklySalary + "',
'0' )";
                displayQuery = "SELECT employees.socialSecurityNumber, " +
                        "employees.firstName, employees.lastName, " +
                        "employees.employeeType,
salariedEmployees.weeklySalary" +
                        " FROM employees, salariedEmployees WHERE " +
                        "employees.socialSecurityNumber = " +
                        "salariedEmployees.socialSecurityNumber";
            }
```

The ActionEvent event is triggered any time a button is clicked, so to figure out which of the 4 buttons it came from, we use the getSource method. If the source is the addSalariedEmployee method, we display a JOptionPane with InputDialog prompt for the user to input the weeklySalary, and parce the user input as a Double and assign it to the variable weeklySalary.

We then change insertQuery to read `"INSERT INTO salariedEmployees VALUES ( '" + socialSecurityNumber + "', '" + weeklySalary + "', '0' )"`, which will put the employee with the appropriate socialSecurityNumber into the salariedEmployees table with the indicated weeklySalary value and 0 as bonus (we didn't want to mess with bonuses, so we make them zero and don't display them throughout our application). Now we wish to display the salariedEmployees table, so we modify displayQuery to read `"SELECT employees.socialSecurityNumber, " + "employees.firstName, employees.lastName, " + "employees.employeeType, salariedEmployees.weeklySalary" + " FROM employees, salariedEmployees WHERE " + "employees.socialSecurityNumber = " + "salariedEmployees.socialSecurityNumber"` This displays the columns socialSecurityNumber, firstName, lastName, and employeeType from the employees table, along with weeklySalary from the salariedEmployees table, where the socialSecurityNumber input by the user is used to retrieve this information from each table and link the entries from those two tables.

What do we do with these two modified query strings? Let us skip the sections for the other buttons and get to the bottom of the ButtonHandler class:

```
            try {
                statement = connection.createStatement();
                statement.executeUpdate(insertQuery);

// display the employee info
                statement = connection.createStatement();
                resultSet = statement.executeQuery(displayQuery);
                displayResultSet(resultSet);
            }
            catch (SQLException exception) {
                exception.printStackTrace();
            }
        } // end method actionPerformed
    } // end inner class ButtonHandler
```

We envelop everything in a try…catch just in case there are SQL errors, we can catch them, print the StackTrace, and continue operating without the program crashing. Now, in the try block we first create a statement using our connection to the employees database. Then we execute the insertQuery String as an SQL command, which, as discussed earlier, puts the employee with the appropriate socialSecurityNumber into the salariedEmployees table with the indicated weeklySalary value and 0 as bonus. Then we create a another statement using our connection to our employees database. Then we execute our displayQuery which reutnrs a resultSet containing the data from the requested tables in the requested columns, etc. We then use displayResultSet to display this information.

Now, let us go back up to the code for the other types of button. For instance, if the ActionEvent came from the addCommissionEmployee button, then event.getSource() will branch us into the following block of code:

```
// add commission employee to table commissionEmployee
            else if (event.getSource() == addCommissionEmployee) {
```

```
                int grossSales =
Integer.parseInt(JOptionPane.showInputDialog("Gross Sales:"));
                double commissionRate =
Double.parseDouble(JOptionPane.showInputDialog("Commission Rate:"));
                insertQuery = "INSERT INTO commissionEmployees VALUES ( '" +
                        socialSecurityNumber + "', '" + grossSales + "', '" +
                        commissionRate + "', '0' )";
                displayQuery = "SELECT employees.socialSecurityNumber, " +
                        "employees.firstName, employees.lastName, " +
                        "employees.employeeType,
commissionEmployees.grossSales," +
                        " commissionEmployees.commissionRate FROM employees,
" +
                        "commissionEmployees WHERE
employees.socialSecurityNumber="
                        + "commissionEmployees.socialSecurityNumber";
            }
```

We use a JOptionPane to get the user to input grossSales, which we parse as an Integer and store with variable grossSales. We use another JOptionPane to get the user to input commissionRate, which we parse as a Double and store with the variable commissionRate. In this case, we modify the insertQuery String to read `"INSERT INTO commissionEmployees VALUES ( '" + socialSecurityNumber + "', '" + grossSales + "', '" + commissionRate + "', '0' )"`, i.e. when this query is executed we insert the employee with the user-defined socialSecurityNumber into the commissionEmployees data base with grossSales and commissionRate as input by the user via the JOptionPanes. Then we modify the displayQuery in a similar way to the salariedEmployees case, except that, rather than adding a column from salariedEmployees, we tack on columns grossSales and commissionRate from the commissionEmployees table.

The basePlusCommissionEmployee JButton case is exactly the same as commissionEmployee, except that there is an additional JOptionPane with InputDialog for baseSalary where the user input is parsed as a double and stored as variable baseSalary,. The insertQuery String that is eventually executed will include an additional argument for baseSalary, and the displayQuery will tack on an additional column for baseSalary (and the data will come from the employees and basePlusCommissionEmployees tables rather than the employees and commissionEmployees ables):

```
// add base plus commission employee to table basePlusCommissionEmployee
            else if (event.getSource() == addBasePlusCommissionEmployee) {
                int grossSales =
Integer.parseInt(JOptionPane.showInputDialog("Gross Sales:"));
                double commissionRate =
Double.parseDouble(JOptionPane.showInputDialog("Commission Rate:"));
                double baseSalary =
Double.parseDouble(JOptionPane.showInputDialog("Base Salary:"));
                insertQuery = "INSERT INTO basePlusCommissionEmployees " +
                        "VALUES ( '" + socialSecurityNumber + "', '" +
grossSales +
                        "', '" + commissionRate + "', '" + baseSalary + "',
'0' )";
```

```
                displayQuery = "SELECT employees.socialSecurityNumber, " +
                        "employees.firstName, employees.lastName, employees."
+
                        "employeeType,
basePlusCommissionEmployees.baseSalary, " +
                        "basePlusCommissionEmployees.grossSales, basePlus" +
                        "CommissionEmployees.commissionRate FROM employees, "
+
                        "basePlusCommissionEmployees WHERE " +
                        "employees.socialSecurityNumber = " +
                        "basePlusCommissionEmployees.socialSecurityNumber";
            }
```

Lastly, we have an "else" case, which will be for the hourlyEmployee JButton, since it is the only one that we did not handle explicitly:

```
// add hourly employee to table hourlyEmployee
            else {
                int hours = Integer.parseInt(
                        JOptionPane.showInputDialog( "Hours:" ) );
                double wage = Double.parseDouble(
                        JOptionPane.showInputDialog( "Wage:" ) );
                insertQuery = "INSERT INTO hourlyEmployees VALUES ( '" +
                        socialSecurityNumber + "', '" + hours + "', '" + wage
+
                        "', '0' )";
                displayQuery = "SELECT employees.socialSecurityNumber, " +
                        "employees.firstName, employees.lastName, " +
                        "employees.employeeType, hourlyEmployees.hours, " +
                        "hourlyEmployees.wage FROM employees, hourlyEmployees
" +
                        "WHERE employees.socialSecurityNumber = " +
                        "hourlyEmployees.socialSecurityNumber";
            }
```

We use JOptionPanes to allow users to input int hours (we parse user input as Integer and store as int) and wage (we parse user input as Double and store as double). We then modify insertQuery to `read "INSERT INTO hourlyEmployees VALUES ( '" + socialSecurityNumber + "', '" + hours + "', '" + wage + "', '0' )"`, which when executed later will insert an employee with the user input socialSecurityNumber into the hourlyEmployees table with hours and wages as defined by the user input to the JOptionPanes. We then modify the displayQuery String in a manner similar to the other cases, where we include columns socialSecurityNumber, firstName, lastName, and employeeType from the employees table, and columns hours and wage from the hourlyEmployees table, which, when executed and the resulting ResultSet passed into the displayResultSet method, will display the columns just described, linking information from the employees and hourlyEmployees tables by using the socialSecurityNumber key.

This completes our description of the operation and architecture of our employee database application.