

# Creating databases and populating tables with Java desktop applications

Tutorial

# What we need at home

- JRE
- Eclipse
- JDBC drivers:
  - Postgres v. 9.1
  - SQLite
- SQLite database
- ssh tunnel to Postgres db server

# What we need in the lab

- JRE
- Eclipse
- JDBC drivers:
  - Postgres v. 9.1
  - SQLite
- SQLite database

All is already  
installed

Step-by-step installation  
instructions

# JRE-Java Runtime Environment (If not already installed – check JRE, JDK)

- The Java Runtime Environment (JRE) released by Oracle is a software distribution containing a stand-alone Java VM and Java standard libraries
- Install the latest version of JRE for your operating system from <http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html>
- Java 7 was used to develop this demo
- Ensure that the full directory path of the **JRE bin directory** is in your **PATH** environment variable so that you can run the Java application launcher from any directory.

# Eclipse

- Install Development Environment which will make your life easier
- Suggested IDE: Eclipse
- Download **Eclipse for Java Developers** from <https://eclipse.org/downloads/>
- Unzip into directory of your choice
- Locate an executable called *eclipse* in this directory and launch Eclipse
- If it does not launch, check that your Java bin has been added to the PATH

# JDBC

- JDBC: API that allows java to communicate with a database server using SQL commands.
  - To use it do: `import java.sql.*`
- Most important:
  - **Connection**
  - **Statement, PreparedStatement**
  - **ResultSet**
- They are **interfaces** instead of classes.
  - Because the point of JDBC is to hide the specifics of accessing a particular database.
  - Implementation of the underlying classes is done in the **vendor provided driver** and associated classes.

# PostgreSQL

- You can download and install the PostgreSQL from:  
<https://www.postgresql.org/>
- Alternatively, the PostgreSQL server version 9.1 is installed in the lab
- You can access it through localhost if you open an ssh tunnel
- REMEMBER: if you close the tunnel, you cannot access the db server through localhost anymore, you need to open the tunnel again



# Opening SSH tunnel to cdf db server

Depending on your ssh client execute:

**Username** is your cdf login

```
ssh username@dbsrv1.cdf.toronto.edu -L 5432:localhost:5432
```

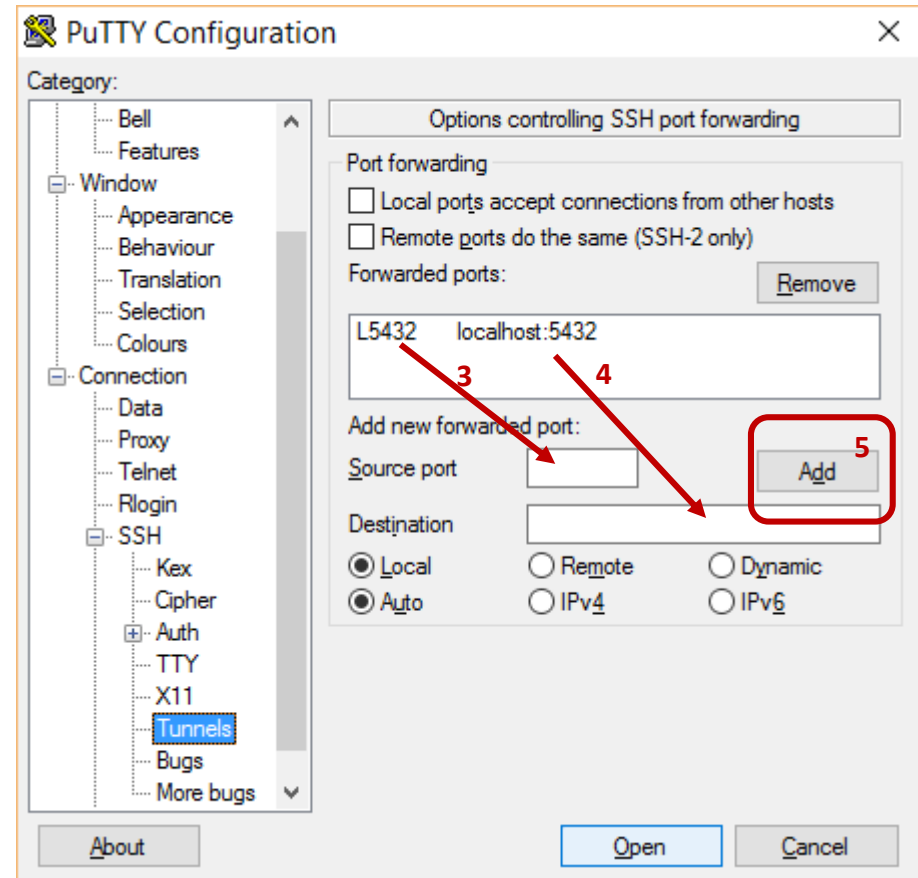
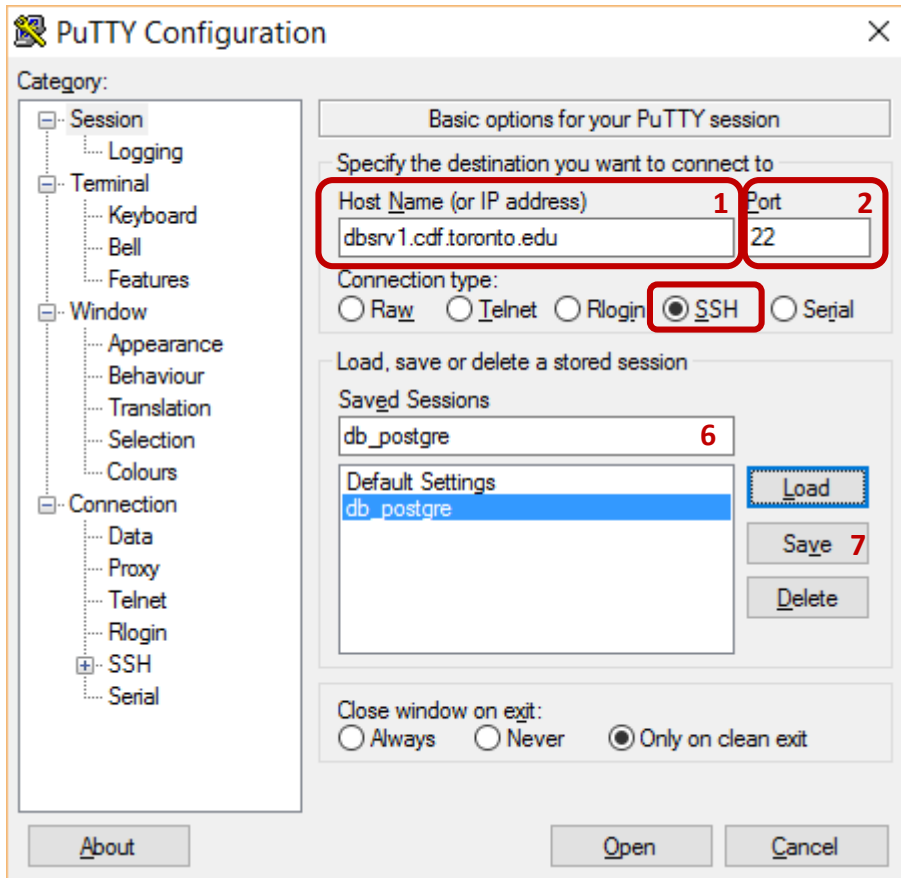
```
ssh2 username@dbsrv1.cdf.toronto.edu -L 5432:localhost:5432
```

```
putty mgbarsky@dbsrv1.cdf.toronto.edu -L 5432:localhost:5432
```

Enter your password

Now a **secure communication tunnel** has been opened between your db on server and your localhost:5432, and you can address your database as localhost:5432/csc343h-**username**

# Saved tunnel on windows (Putty)



# JDBC driver for Postgres

- Download the JDBC driver for version Postgres 9.1 (running on cdf database server)
- Download from here:  
<https://jdbc.postgresql.org/download.html#archived>
- The driver comes in a form of a .jar file, which you need to add to your project
- Under Eclipse directory create folder lib and save Postgres JDBC driver there: **postgresql-9.1-903.jdbc4.jar**

# SQLite3

- Fast, small-footprint, installation-free database, well suited for data analysis.

<https://www.sqlite.org/whentouse.html>

- Just download sqlite3 and start running queries.

<http://www.sqlite.org/download.html>

# JDBC driver for SQLite

- We will **develop** our assignment 1 with SQLite **first**
- Download the driver for SQLite:

<https://bitbucket.org/xerial/sqlite-jdbc/downloads>

- Another jar file into lib directory

[sqlite-jdbc-3.8.11.2.jar](#)

# Pizza lovers

demo application

# Part 1. Creating database

# Relations and schema

- The relations for this demo represent information about customers who eat pizza. The basic information about each person is recorded, as well as what types of pizza and where did this person eat

- Schema:

**Person**(name: string, age: int, gender: string);

**Frequents**(name: string, pizzeria: string);

**Eats**(name: string, pizza: string);

**Serves**(pizzeria: string, pizza: string, price: float);



# Creating tables in Postgres

```
create table Person(name VARCHAR(50), age int, gender  
CHAR(7));
```

```
create table Frequents(name VARCHAR(50), pizzeria  
VARCHAR(50));
```

```
create table Eats(name VARCHAR(50), pizza VARCHAR(70));
```

```
create table Serves(pizzeria VARCHAR(70), pizza  
VARCHAR(70), price NUMERIC(8,2));
```

# Creating database in Postgres

- Script for both table creation and data generation is in file `pizza_pg.sql`
- Run it as usual
- This will also create a new schema: *pizza*
- You can **change your search\_path permanently** by executing the following command:

```
ALTER ROLE uname SET SEARCH_PATH = 'pizza','movie','public';
```

# Creating tables in SQLite

```
create table Person(name text, age int, gender text);
```

```
create table Frequents(name text, pizzeria text);
```

```
create table Eats(name text, pizza text);
```

```
create table Serves(pizzeria text, pizza text, price decimal);
```

**Note how each DBMS has its own data types, but our JDBC application will access them through the common interface**

# Creating database in SQLite

- Script for both table creation and data generation is in file `pizza_sqlite.sql`
- Launch `sqlite`

**SQLite version 3.13.0 2016-05-18 10:57:30**

**sqlite> .open pizzerias**

Creates database named pizzerias

**sqlite> .read pizza\_sqlite.sql**

Runs sql script in file `pizza_sqlite.sql`

**sqlite> SELECT name FROM sqlite\_master WHERE type='table';**

**Person**

**Frequents**

**Eats**

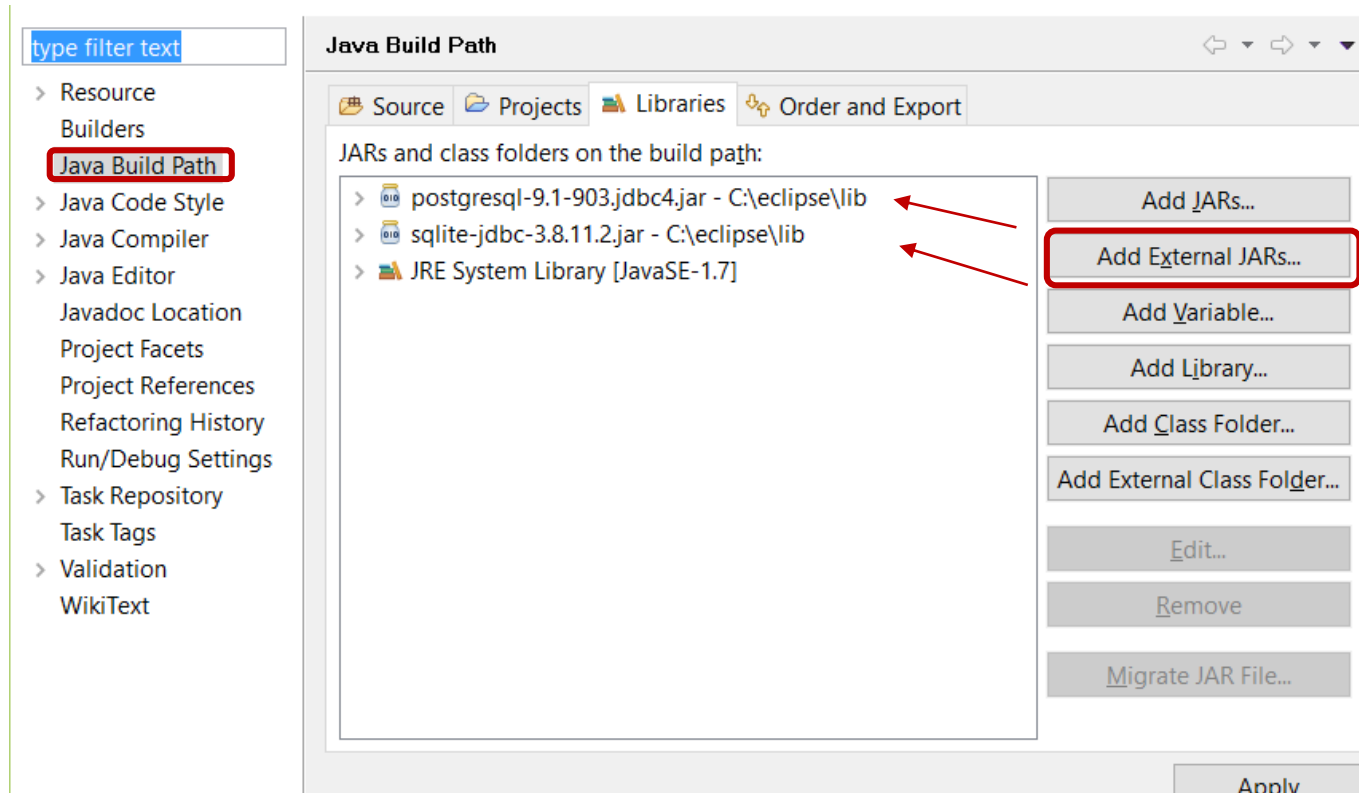
**Serves**

To see all the tables

# Part 2. Connecting to database from Java

# Java project and adding driver libraries

- In Eclipse create new project of type Java project
- Project->Properties->



# Defining connections in properties files

```
driver=org.sqlite.JDBC  
url=jdbc:sqlite:pizzerias
```

In file:  
properties\_sqlite.txt

```
driver=org.postgresql.Driver  
url=jdbc:postgresql://localhost:5432/csc343h-uname  
user=unname  
password=pwd
```

In file:  
properties\_postgre.txt

A single Java application will get the properties file name as a parameter, and work with the corresponding database

[Full code in file DBConnection.java](#)

# Loading driver class

- The syntax for loading a JDBC driver is:

**Class.forName (“drivername”)**

Causes class *drivername* to be dynamically loaded in runtime

- We load the driver depending on the property specified in the properties file:

```
String driver = props.getProperty("driver");
try {
    Class.forName(driver);
}
catch (ClassNotFoundException ce){
    System.out.println("JDBC Driver not found");
    return null;
}
```

[Full code in file DBConnection.java](#)



# Connection

```
String url = props.getProperty("url");  
String user = props.getProperty("user", "");  
String password = props.getProperty("password", "");
```

```
// load the JDBC driver using the current class loader and db URL
```

```
try {
```

```
Class.forName(driver);
```

```
return DriverManager.getConnection(url, user, password);
```

```
}
```

```
catch (ClassNotFoundException ce){
```

```
... return null;
```

```
}
```

```
catch (SQLException ex) {
```

```
... return null;      Full code in file DBConnection.java
```

# The first thing - test connection

```
public static void main (String [] args) throws IOException, SQLException {
    Properties props = new Properties();
    FileInputStream in = new FileInputStream(args[0]);
    props.load(in);
    in.close();

    Connection conn = getConnection (props);
    if (conn == null) {
        System.out.println("DB connection error");
    }
    else {
        System.out.println("Yes! connection works");
        conn.close();
    }
}
}
```

[Full code in file DBConnection.java \(main\)](#)

# Presenting DB errors to the user

```
public class SQLError {  
    public static void print(SQLException ex) {  
        while (ex != null) {  
            System.err.println("SQLState: " + ex.getSQLState());  
            System.err.println("Error Code: " + ex.getErrorCode());  
            System.err.println("Message: " + ex.getMessage());  
            Throwable t = ex.getCause();  
            while (t != null) {  
                System.out.println("Cause: " + t);  
                t = t.getCause();  
            }  
            ex = ex.getNextException();  
        }  
    }  
}
```

Full code in file [SQLError.java](#)

# Statement, Result Set

```
Statement stmt = null;
```

```
String query = "select * FROM " + tblName;
```

```
stmt = conn.createStatement();
```

```
ResultSet rs = stmt.executeQuery(query);
```

```
int cols = rs.getMetaData().getColumnCount();
```

```
while (rs.next()) {
```

```
for (int i=0; i< cols; i++)
```

```
System.out.print (rs.getObject(i+1) + "\t");
```

```
System.out.print("\n");
```

```
}
```

Columns  
start from 1

Printing generic table

Full code in file [PrintTable.java](#)

# Inserting data into a table

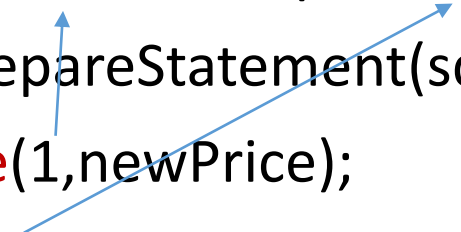
- Statement statement = null;
- String insertTableSQL = "INSERT INTO Serves (pizzeria, pizza, price)" + "  
VALUES ('Pizza Hut','corn',10.5)";

```
try {  
    statement = conn.createStatement();  
    // execute insert SQL statement  
    int qResult = statement.executeUpdate(insertTableSQL);  
    System.out.println("Rows affected="+qResult);  
} catch (SQLException e) {  
    SQLError.show(e);  
}  
} finally { if (statement != null) { statement.close();} }  
Full code in file AddStatement.java
```

# Updating. Prepared Statement

```
PreparedStatement updateStmt = null;
String sql = "UPDATE Serves "+
             "SET price=? WHERE pizza=?";
updateStmt = conn.prepareStatement(sql);
updateStmt.setDouble(1,newPrice);
updateStmt.setString(2, pizzaName);

updateStmt.execute();
```

A diagram consisting of two blue arrows. The first arrow starts from the first question mark '?' in the SQL string "SET price=? WHERE pizza=?" and points to the "1" parameter in the "setDouble(1,newPrice)" method call. The second arrow starts from the second question mark '?' in the same SQL string and points to the "2" parameter in the "setString(2, pizzaName)" method call.

# Deleting

```
PreparedStatement deleteStmt = null;
```

```
String sql = "DELETE FROM Serves "+  
            "WHERE pizzeria=? AND pizza=?";
```

```
deleteStmt = conn.prepareStatement(sql);
```

```
deleteStmt.setString(1,pizzeria);
```

```
deleteStmt.setString(2, pizza);
```

```
deleteStmt.execute();
```

# Person class: fields

```
private String name;
```

```
private int age;
```

```
private String gender;
```



# Person class: db methods

```
public void addToDatabase (Connection conn) throws SQLException  
public void updateInDatabase (Connection conn) throws SQLException  
public void deleteFromDatabase (Connection conn) throws SQLException
```

```
private String insertSQL = "INSERT INTO Person (name, age, gender)"  
                           + " VALUES (?, ?, ?)";
```

```
private String updateSQL = "UPDATE Person SET age=?, gender=? "  
                           + "WHERE name=?";
```

```
private String deleteSQL = "DELETE FROM Person WHERE name=?";
```

```
PreparedStatement stmt = null;
```

[Full code in file Person.java](#)

# setXXX, getXXX

```
private String insertSQL = "INSERT INTO Person "  
    +" (name, age, gender)"  
    + " VALUES (?, ?, ?)";
```

```
stmt = conn.prepareStatement(insertSQL);
```

```
stmt.setString(1, this.name);
```

Adds single quotes

```
stmt.setInt(2, this.age);
```

```
stmt.setString(3, this.gender);
```

```
stmt.execute();
```

# Testing Person functionality with text UI

```
java.sql.Connection conn = DBConnection.getConnection (props);
```

```
// create a scanner so we can read the command-line input
```

```
Scanner scanner = new Scanner(System.in);
```

```
while (true){
```

```
    // prompt for the user's action
```

```
    System.out.print("What would you like to do next? ");
```

```
    int choice = scanner.nextInt();
```

```
}
```

Full code in file [PersonTest.java](#)

# Serves class

```
private String pizzeria;
```

```
private String pizza;
```

```
private double price;
```

```
public void addPizzaToDatabase (Connection conn) throws SQLException
```

```
public void updatePriceInDatabase (Connection conn) throws  
SQLException
```

```
public void deleteFromDatabase (Connection conn) throws SQLException
```

[Full code in file Serves.java](#)

# Persistent Objects and Hibernate

- Open-source persistence framework
- Does object-relational mapping with XML
- Implemented using Java.reflection

```
<hibernate-mapping>
  <class name="Employee" table="EMPLOYEE">
    <meta attribute="class-description">
      This class contains the employee detail.
    </meta>
    <id name="id" type="int" column="id">
      <generator class="native"/>
    </id>
    <property name="firstName" column="first_name"
      type="string"/>
    <property name="lastName" column="last_name"
      type="string"/>
    <property name="salary" column="salary" type="int"/>
  </class>
</hibernate-mapping>
```

```
tx = session.beginTransaction();  
Employee employee = new Employee(fname, lname, salary);  
employeeID = (Integer) session.save(employee);  
tx.commit();
```

Not required for A 1. If you want, you can do A 1 with Hibernate,  
just to learn it and to get 1 point bonus

# Recording data at Point of Sale (POS)

- When the sale occurs, we insert data into 2 different tables: Frequents and Eats.
- This should be an atomic operation

# Sale class. Transactions

```
conn.setAutoCommit(false);
```

```
stmt1 = conn.prepareStatement(insertSQL1);
```

```
stmt1.setString(1, this.name);
```

```
stmt1.setString(2, this.pizzeria);
```

```
stmt2 = conn.prepareStatement(insertSQL2);
```

```
stmt2.setString(1, this.name);
```

```
stmt2.setString(2, this.pizza);
```

```
stmt1.execute();
```

```
stmt2.execute();
```

```
conn.commit();
```

Full code in file POSapp.java

# GUI with swing

- The most difficult part: layout
- The demo also shows how to create and update JTable object

Full code in file POSapp.java

SWING is not required for A 1. IF YOU KNOW IT – do A1 with Swing, for 1 point bonus