Revised: 12/17/2019

Databases

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Database Management Systems
  - JDBC
  - SQL
  - Postgres
- Setting up Postgres
  - Creating a Database
- Connecting to a Database
- Creating a Table
  - Statement Objects
  - CREATE TABLE Statement
- Inserting Records
  - INSERT Statement
- Changing Records
  - UPDATE Statement
- Retrieving Records
  - SELECT Statement
  - ResultSet Objects
- Deleting Records
  - DELETE Statement

Colors/Fonts

Local Variable Names	Brown
Primitive data types	Fuchsia
Literals	Blue
Keywords	Orange
Object names	Green
Operators/Punctuation	Black
Field Names	Lt Blue
Method Names	Purple
Parameter Names	Gold
Comments	Gray
Package Names	Pink

Source Code	**Consolas**
Output	`Courier New`

Database Management Systems

- When an application only needs to store a small amount of data, text files and binary files are fine.
- But when you are dealing with huge amounts of data, searching, inserting, and deleting information from text and binary files is very inefficient.
- A Database Management System (DBMS) is software that is designed to store, retrieve, and manipulate large amounts of data.
- Some common Database Management Systems: Oracle, MS SQL, PostgreSQL, MySQL, and MariaDB

Database Management Systems

- The DBMS handles all interaction with the data it maintains.
  - Java programs (or any language that can interact with the particular DBMS) communicate with the DBMS who then carries out the instructions.

- The advantage is applications do not need to know *how* the data is structured and stored; they only need to know how to interact with the DBMS.

Java Database Connectivity (JDBC)

- The JDBC API allows Java applications to interact with a DMBS.

- Databases provide JDBC drivers that Java classes use to connect to and interact with the DBMS.

Structured Query Language (SQL)

- SQL (pronounced "*S-Q-L*" or "*Sequel*") is a widely used and *mostly* standard language for working with database management systems.
  - Not really a programming language; You can't create a program using SQL.
  - Some variations between different *dialects* of SQL.
    - For example, the syntax for Oracle SQL is a little different from MySQL or MS SQL.
  - If you know one dialect, learning others is easy.

- When applications interact with a DBMS to retrieve or manipulate data, it is typically done with an SQL statement or query.
  - The application creates the query and passes it to the DBMS to be executed.

- While this isn't a database course, we will become familiar with some basic SQL keywords and queries.
  - Concepts found in any dialect of SQL.

PostgreSQL

- PostgreSQL (pronounced *“post-gress”* or *“post-gress-Q-L”* and often just called Postgres) is an open source DBMS.
  - Initially released in 1996.

- This is what we will be using for this course.

PostgreSQL

- Complete the steps in the PostgreSQL Server Setup Guide
- Be sure to download the PostgreSQL JDBC Driver (jar file) posted on Canvas.
    - The jar is added to an IntelliJ project the same way you added the JUnit jars/libraries.
        - Follow the steps in the PostgreSQL Driver Setup Guide to add the library to the IntelliJ project.
    - This is needed for Java programs to connect to Postgres databases.

Managing Postgres

- After installing Postgres, you should see a **PostgreSQL 11** folder in your start menu.
  - Inside that folder, click the **pgAdmin 4** menu item.
  - If you are on an Apple computer, use the Spotlight feature to search for pgAdmin.

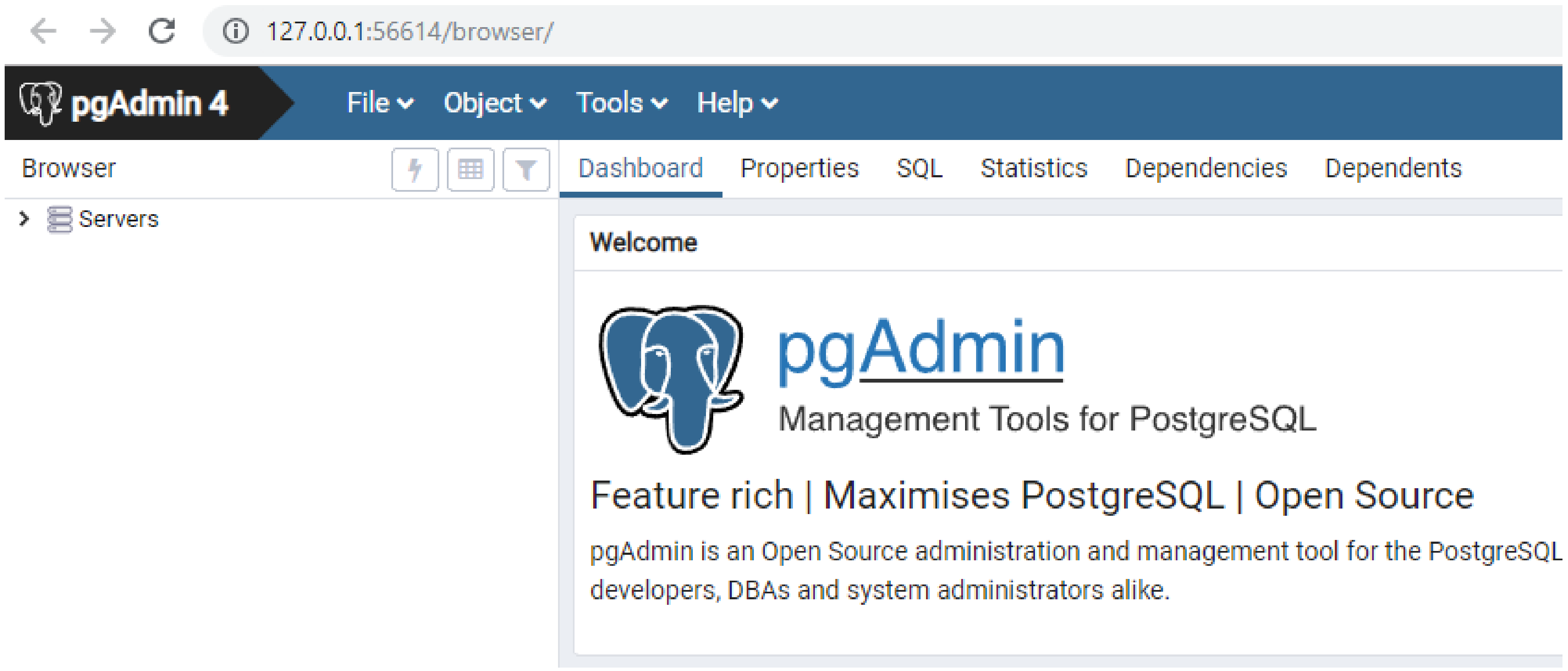
- pgAdmin is a web-based interface for managing the database server.

Postgres Server

- Postgres might not start automatically when you turn on your computer.

- Opening pgAdmin will start the server.

Setting up Postgres

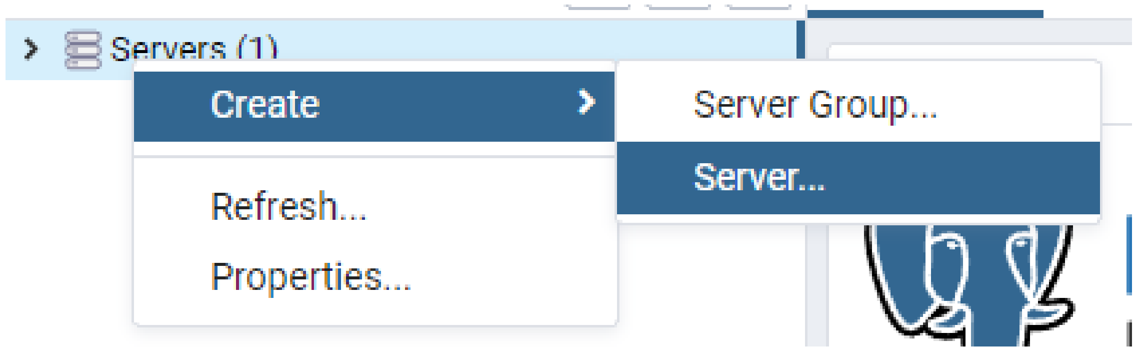


Feature rich | Maximises PostgreSQL | Open Source

pgAdmin is an Open Source administration and management tool for the PostgreSQL developers, DBAs and system administrators alike.

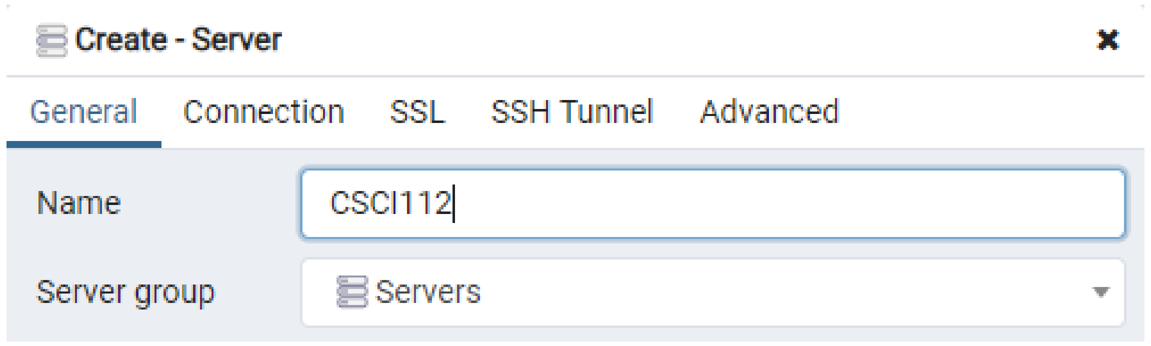
Creating a Database

- Right-click **Servers** on the left and choose **Create** > **Server…**



Creating a Database

- On the General tab, name the server CSCI112

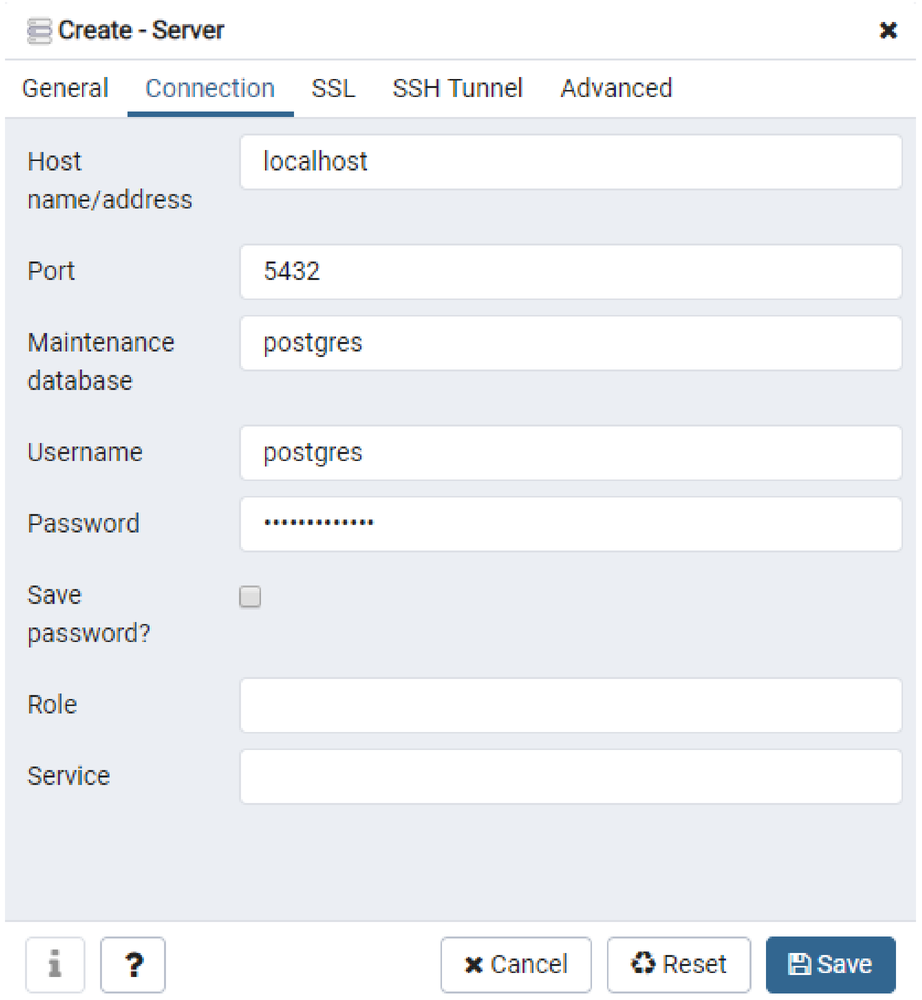


Name: CSCI112

Server group: Servers

Creating a Database

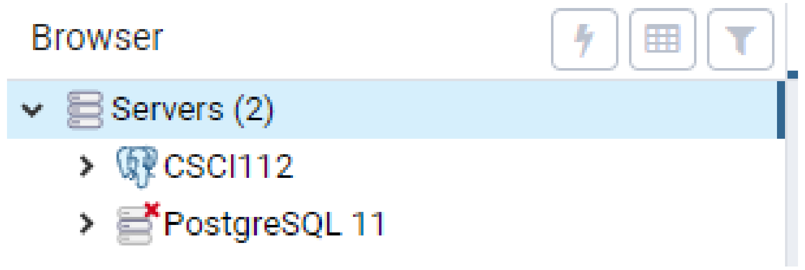
- On the Connection tab, ensure your settings match what is shown to the right.
- Enter the administrator password you set when installing Postgres.
    - CSCI112ONLINE if you followed my suggestion in the setup document.
- Click **Save**.



Host name/address	localhost
Port	5432
Maintenance database	postgres
Username	postgres
Password	•••••••••••••
Save password?	☐
Role	
Service	

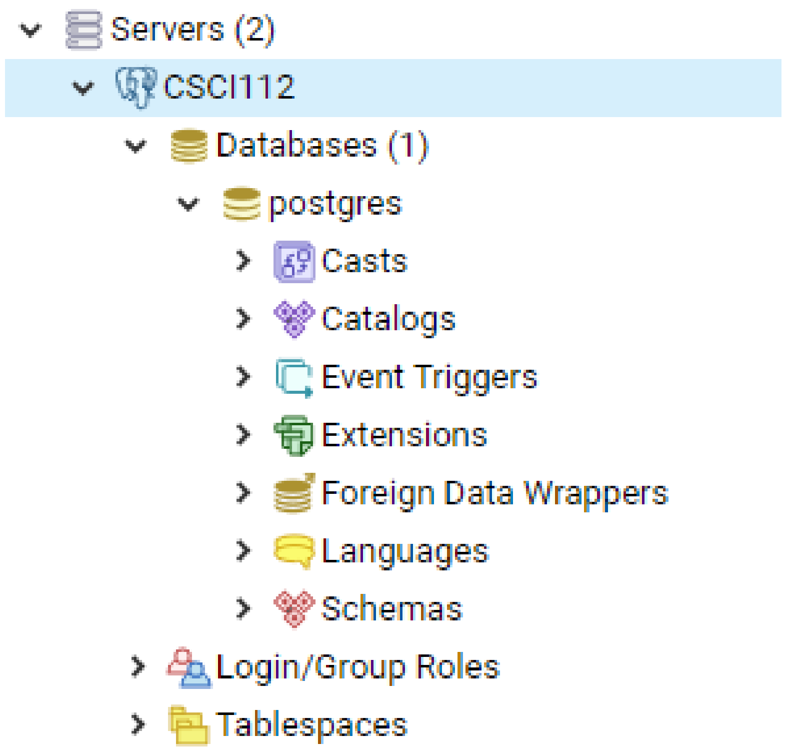
Creating a Database

- You’ll see your new server listed on the left.
  - The **PostgreSQL 11** server was created by default during installation.



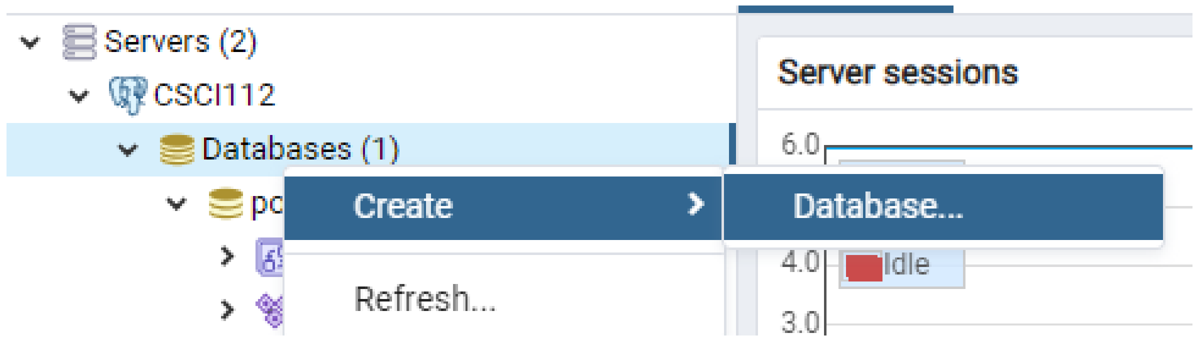
Creating a Database

- Now with a database server setup, we can add a database to the server.
- Expand the CSCI112 Server.
- Expand the Databases group.
- You'll see a postgres database that already exists.
  - This is a maintenance database that is necessary for the server but we will not utilize it in this course.



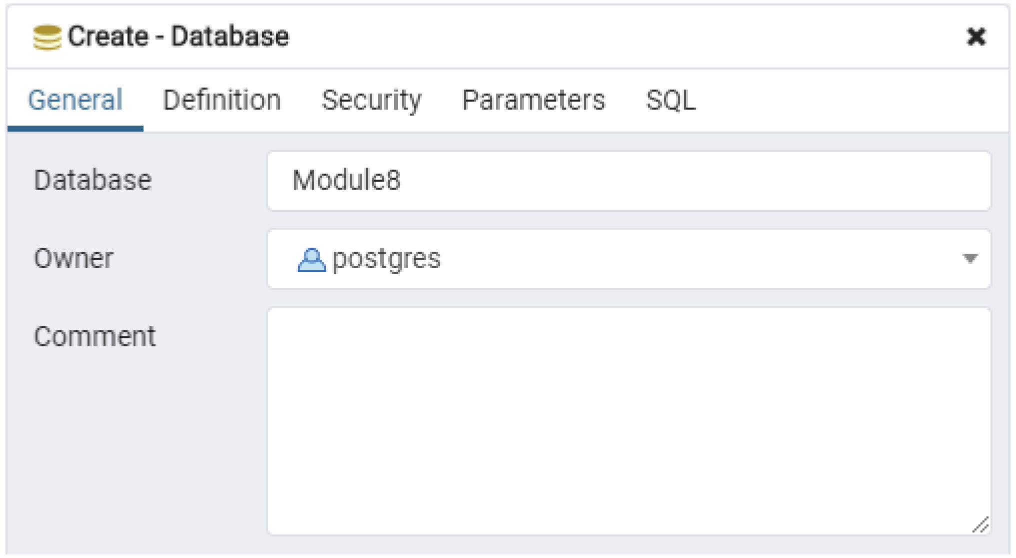
Creating a Database

- Right-click the Databases group and select **Create** > **Database…**



Creating a Database

- Name the database **Module8** (all one word)
- Click **Save**.



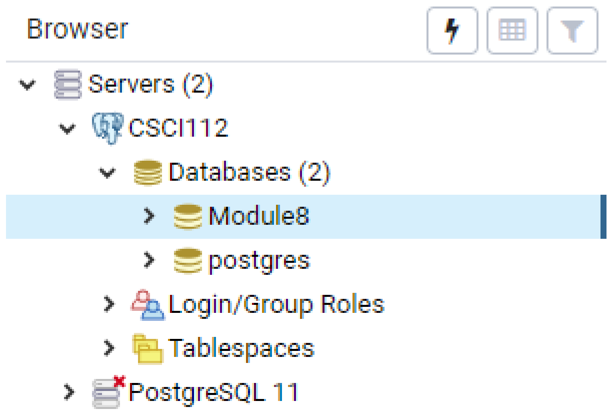
Database: Module8

Owner: postgres

Comment:

Creating a Database

- The Module8 database should now be listed in the Databases group.



Connecting to a Database

- The Connection object is used to create a single connection to a database.
  - You can have multiple instances of Connection objects connected to the same database (or other databases), but each object only has one connection.

- Must be imported:

```
import java.sql.Connection;
```

Connecting to a Database

- Connection objects are rarely instantiated using its constructor.
  - Aside from the Connection object’s default constructor, there are no others.

- To get an instance of a Connection object, we’ll use the DriverManager object.

Connecting to a Database

- The DriverManager object manages JDBC drivers.

```
import java.sql.DriverManager;
```

- The DriverManager object contains only static methods, so you do not need to instantiate it.

Connecting to a Database

- To get an instance of a Connection object, use the DriverManager's getConnection method.
  - The "URL" is a string that indicates the database we wish to connect to. (See next slide)
  - The "Username" is the account to connect to the database with.
    - Use `"postgres"`
  - The "Password" is the account's password.
    - Use `"CSCI112ONLINE"` if you followed my suggestion (or use the password you chose during the Postgres installation).

```
Connection conn = DriverManager.getConnection(URL, Username, Password);
```

Connecting to a Database

- The format for the URL is

  <u>protocol</u>:<u>subprotocol</u>://<u>database address</u>

  - When using JDBC, the protocol will always be “jdbc”
  - The subprotocol is the type of DBMS we will be connecting to. For Postgres, the subprotocol is “postgresql”
  - The database address is the address of the database you are connecting to.
    - Usually an IP address or domain name like mydatabase.com/dbname
    - Since we’re using a local server, this will always be **localhost/Module8**
    - You could also use **127.0.0.1/Module8**

Connecting to a Database

- For our work, we will always use this for getting the connection:

```
Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost/Module8",
                                               "postgres",
                                               "CSCI112ONLINE");
```

Connecting to a Database

```
import java.sql.SQLException;
```

- If the getConnection method can't (for whatever reason) establish a connection, it will throw a SQLException.

```
Connection conn = null;
try {
    Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost/abcd",
                                                  "postgres",
                                                  "CSCI112ONLINE");
    System.out.println("Connected to database.");
}
catch(SQLException e) {
    System.out.println("Error connecting to database: " + e.getMessage());
}
```

```
Error connecting to database: Database 'abcd' not found.
```

Connecting to a Database

```
Connection conn = null;
try {
    Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost/Module8",
                                                 "postgres",
                                                 "CSCI112ONLINE");
    System.out.println("Connected to database.");
}
catch(SQLException e) {
    System.out.println("Error connecting to database: " + e.getMessage());
}
```

```
Connected to database.
```

Closing the Connection

- When our program is done with the database connection, call its close method.
    - Typically done in a finally clause.

```
conn.close();
```

Closing the Connection

```
Connection conn = null;
try {
    Connection conn = DriverManager.getConnection("jdbc:postgresql://localhost/Module8",
                                                   "postgres",
                                                   "CSCI112ONLINE");
    System.out.println("Connected to database.");
}
catch(SQLException e) {
    System.out.println("Error connecting to database: " + e.getMessage());
}
finally {
    try {
        conn.close();
    }
    catch(Exception e) {
        //Connection wasn't opened; Do nothing.
    }
}
```

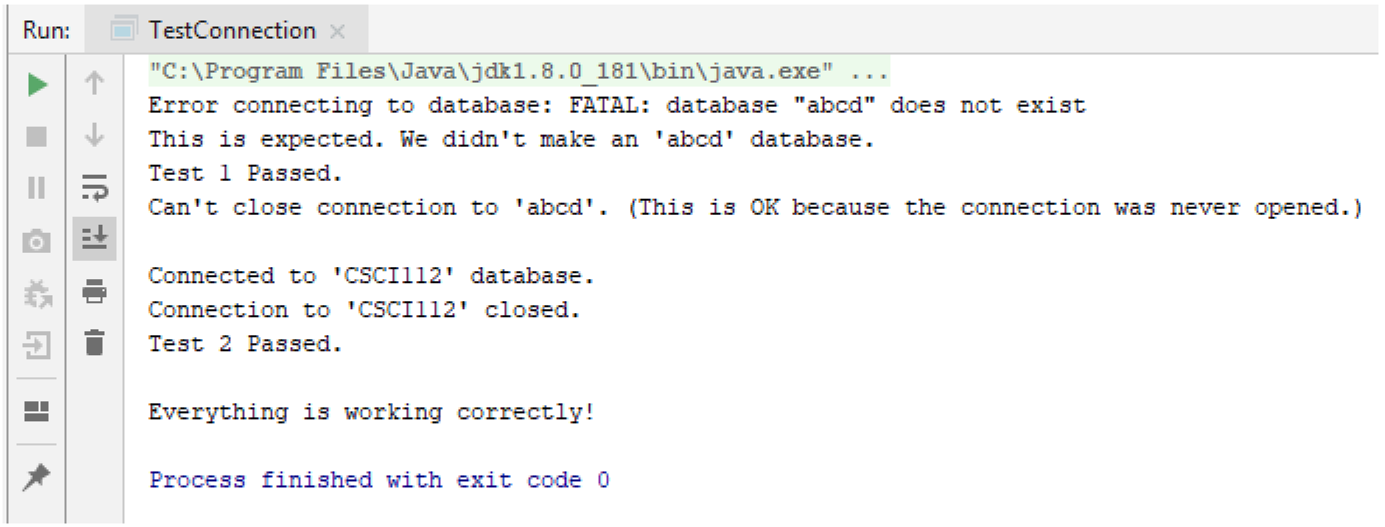
Connecting to a Database

- Follow the steps in the Postgres Driver Setup document posted in Module 8.
  - You'll add the jar library just as you did for the JUnit jar libraries.

- This library is needed for any programs in Module 8 that try to connect to a Postgres database.

Connecting to a Database

- Find the TestConnection.java program in the Module 8 SampleCode folder.
  - Update the password variable if you used something other than "CSCI112ONLINE"
  - Run the program to test that everything is set up correctly.



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Error connecting to database: FATAL: database "abcd" does not exist
This is expected. We didn't make an 'abcd' database.
Test 1 Passed.
Can't close connection to 'abcd'. (This is OK because the connection was never opened.)

Connected to 'CSCI112' database.
Connection to 'CSCI112' closed.
Test 2 Passed.

Everything is working correctly!

Process finished with exit code 0
```

Connecting to a Database

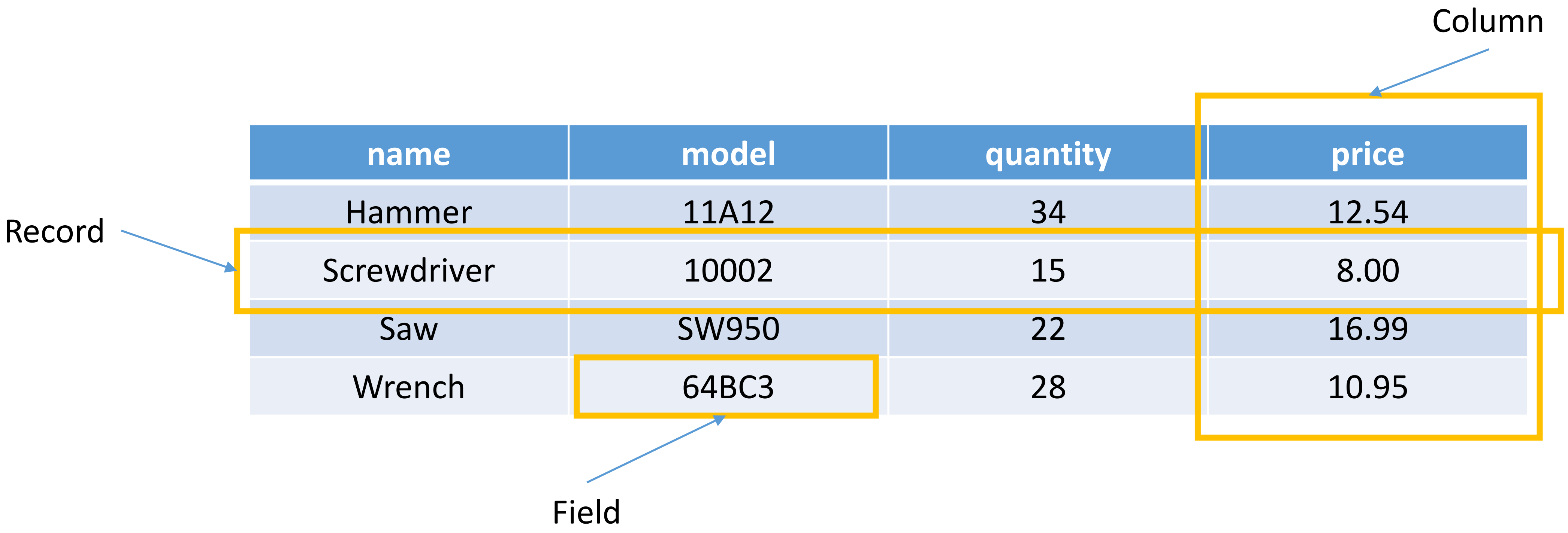
- **Do not proceed until TestConnection's tests pass**.

- Things to check if it's not working:
  - Check the DBMS is running
    - Start pgAdmin to start the server
  - Check that you added the Postgres Driver correctly
    - Review Postgres Driver Setup document.
  - Check the URL (ie. database name and protocols are spelled correctly.)
  - Check that the username and password variables are set correctly.

Creating a Table

- Relational databases like Postgres (and most other DBMS’s) organize their data into **tables**.
- Tables are similar to a spreadsheet
  - Data is organized into rows and columns
- Each column in a table is named and has a data type.
- Each row is referred to as a **record**
- Each “cell” in the table is referred to as a **field**

Creating a Table



name	model	quantity	price
Hammer	11A12	34	12.54
Screwdriver	10002	15	8.00
Saw	SW950	22	16.99
Wrench	64BC3	28	10.95

Data Types

- Each column in a table must have a type specified.
- The common types are:
  - smallint – 16-bit integers (shorts)
  - int – 32-bit integers (ints)
  - bigint – 64-bit integers (longs)
  - real – 32-bit floating point numbers (floats)
  - double precision – 64-bit floating point numbers (doubles)
  - boolean – booleans
  - varchar(n) – Strings of n-length
  - text – Strings with no length restriction

For more on data types and descriptions:
https://www.postgresql.org/docs/11/datatype.html

Statements

- Statement objects allow us to execute SQL statements on the data in the database.

```
import java.sql.Statement;
```

- The Connection object’s createStatement method handles getting us an instance of a Statement object.
    - Of course, your Connection object must already have an open connection with a database.

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

Creating Tables

- The general format for a SQL statement to create a table is:

  CREATE TABLE *tableName* (*columnName* datatype, …)

```
"CREATE TABLE inventory (name varchar(15),
                         model varchar(5),
                         quantity int,
                         price double precision)"
```

Creating Tables

- The Statement object's execute method accepts a String argument.
  - The String is the SQL statement to execute.

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

stmt.execute("CREATE TABLE inventory (name varchar(15),
                                       model varchar(5),
                                       quantity int,
                                       price double precision)");
```

- This creates a table named inventory with four columns:
  - name (strings limited to 15 characters)
  - model (strings limited to 5 characters)
  - quantity (ints)
  - price (doubles)

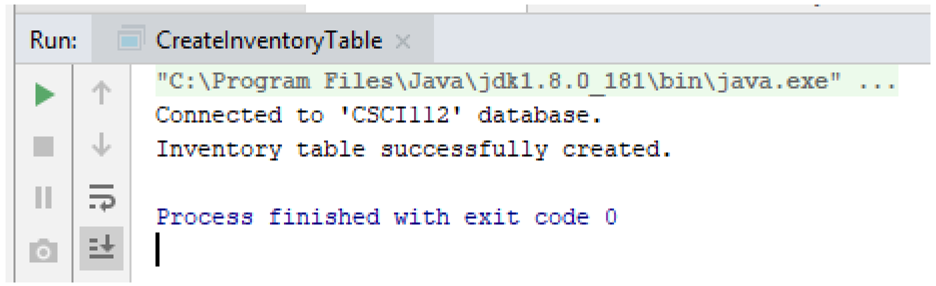
Creating Tables

- If a table named Inventory already existed, a SQLException will be thrown.

- Statement objects can be used more than once.
    - You don’t need a Statement object for every SQL statement.
    - You can use the same Statement object and pass it new Strings to execute.

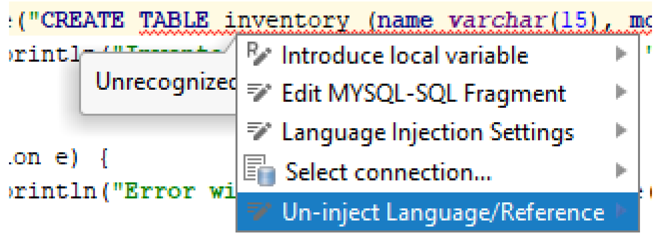
Creating Tables

- Review and run the CreateInventoryTable.java program to create this Inventory table in your Module8 database.



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Connected to 'CSCI112' database.
Inventory table successfully created.

Process finished with exit code 0
```

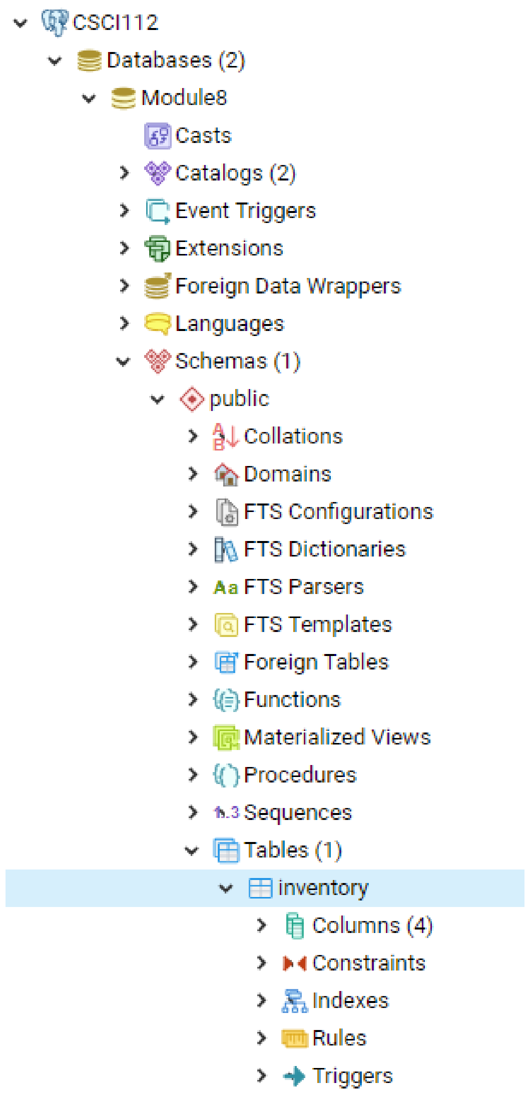
- If there is an error shown for the SQL statement, click red line under the SQL statement and press Alt Enter
  - Select Un-inject Language/Reference



```
:("CREATE TABLE inventory (name varchar(15), mc
println("Inventor
.on e) {
println("Error wi
```

Creating Tables

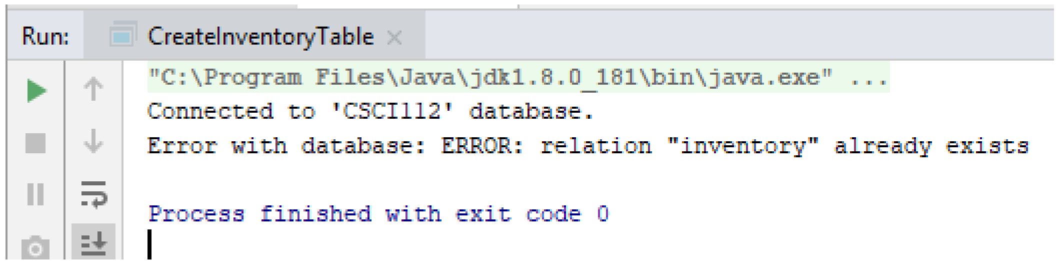
- Open pgAdmin and verify the inventory table exists in the database.
  - (See right)



```
CSCI112
  Databases (2)
    Module8
      Casts
      Catalogs (2)
      Event Triggers
      Extensions
      Foreign Data Wrappers
      Languages
      Schemas (1)
        public
          Collations
          Domains
          FTS Configurations
          FTS Dictionaries
          FTS Parsers
          FTS Templates
          Foreign Tables
          Functions
          Materialized Views
          Procedures
          Sequences
          Tables (1)
            inventory
              Columns (4)
              Constraints
              Indexes
              Rules
              Triggers
```

Creating Tables

- Re-running the CreateInventoryTable.java program will cause an exception to be thrown since the table already exists.



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Connected to 'CSCI112' database.
Error with database: ERROR: relation "inventory" already exists

Process finished with exit code 0
```

Creating Tables

- More Information:
  - https://www.postgresql.org/docs/11/ddl-basics.html

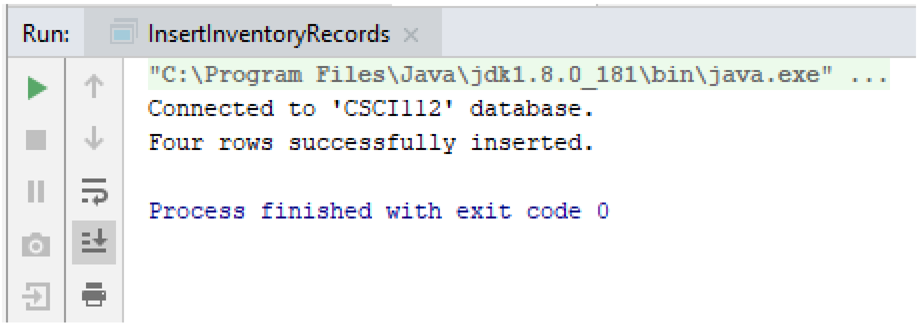
Inserting Records

- The general format for a SQL statement to insert a record is:
  - INSERT INTO tableName VALUES ('some string', 1234)
    - The values must be comma separated and in parentheses
    - There must be one value (of the correct type) for each column
    - Varchars and text should be in single quotes
    - Numeric values do not need quotes
- When inserting, we need to use the Statement object’s **executeUpdate** method.

```
stmt.executeUpdate("INSERT INTO inventory VALUES ('Hammer', '11A12', 34, 12.54)");
```

Inserting Records

- Review and run the InsertInventoryRecords.java program to insert four records into the Inventory table in your Module8 database.

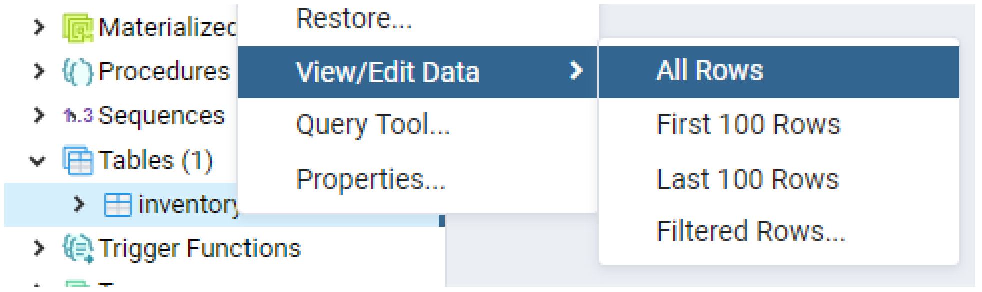


```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Connected to 'CSCI112' database.
Four rows successfully inserted.

Process finished with exit code 0
```

Viewing a table in pgAdmin

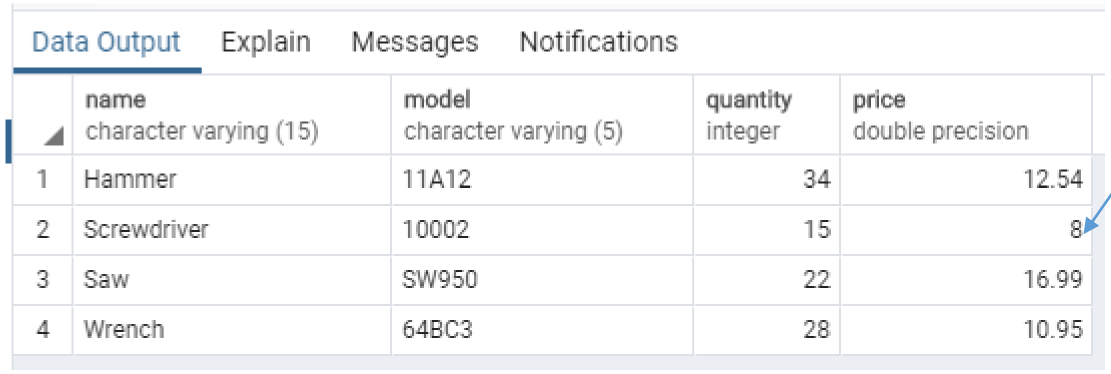
- Right click the inventory table in pgAdmin.

- Select **View/Edit Data** > **All Rows**



Viewing a table in pgAdmin

- You can manually edit the data in the table here.
  - Double-click the field to manually change the value.



	name character varying (15)	model character varying (5)	quantity integer	price double precision
1	Hammer	11A12	34	12.54
2	Screwdriver	10002	15	8
3	Saw	SW950	22	16.99
4	Wrench	64BC3	28	10.95

Doubles with no fractional amount will display as an integer. The type is still double.

- <u>You do not need to change any fields</u>. This is just to show you how you can view/modify the contents of the table through pgAdmin.

Inserting Records

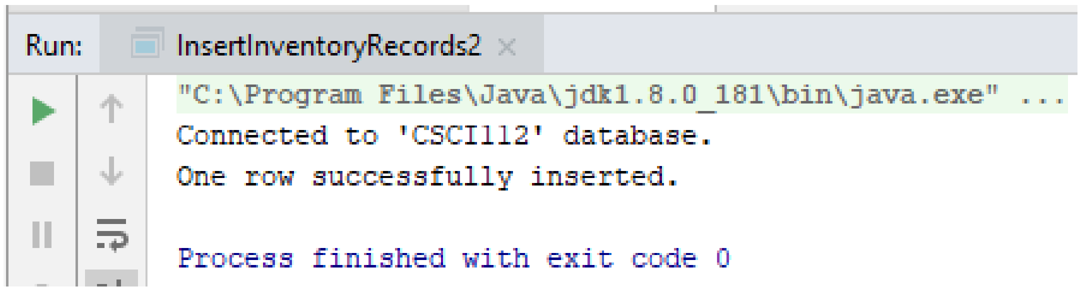
- When concatenating variables into a SQL statement, be sure any concatenated Strings are surrounded by single quotes.

```
String name = "Pliers";
String model = "XT51";
int quantity = 13;
double price = 11.99;

stmt.executeUpdate("INSERT INTO inventory VALUES ('" + name + "', '" + model + "', " +
quantity + ", " + price + ")");
```

INSERT INTO inventory VALUES ('**Pliers**', '**XT51**', **13**, **11.99**)

Inserting Records

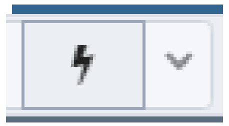
- Run the InsertInventoryRecords2.java program to insert the previously shown record into the Inventory table in your Module8 database.



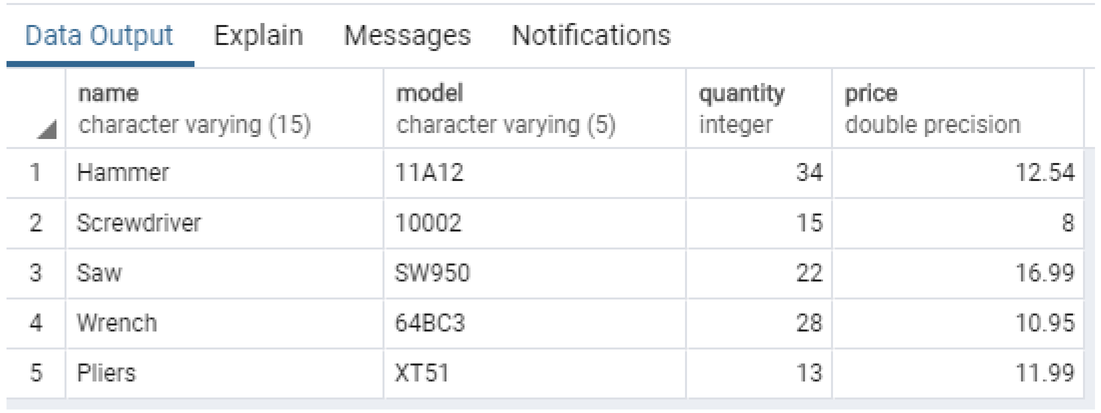
```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Connected to 'CSCI112' database.
One row successfully inserted.

Process finished with exit code 0
```

Inserting Records

- Back in pgAdmin, click the Execute/Refresh button.

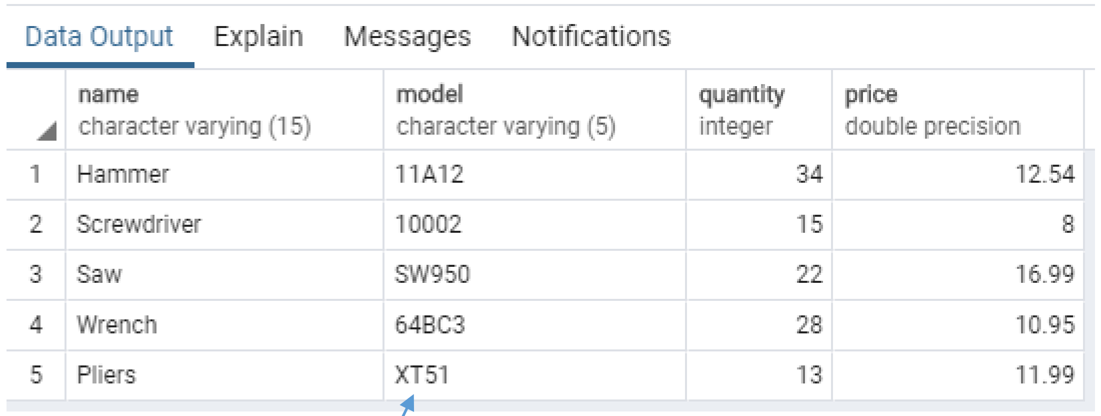


- The inventory table should now show all five records/rows.



	name character varying (15)	model character varying (5)	quantity integer	price double precision
1	Hammer	11A12	34	12.54
2	Screwdriver	10002	15	8
3	Saw	SW950	22	16.99
4	Wrench	64BC3	28	10.95
5	Pliers	XT51	13	11.99

Inserting Records



	name character varying (15)	model character varying (5)	quantity integer	price double precision
1	Hammer	11A12	34	12.54
2	Screwdriver	10002	15	8
3	Saw	SW950	22	16.99
4	Wrench	64BC3	28	10.95
5	Pliers	XT51	13	11.99

It’s OK this is only 4 characters.
The max for this column is 5.

Inserting Records

- More Information:
  - https://www.postgresql.org/docs/11/dml-insert.html

Changing Records

- An UPDATE statement is used to change fields in records.
  - Uses the SET and WHERE keywords

- Must specify:
  - The table to update.
  - The field(s) to update.
  - Condition that indicates which records to change.

Changing Records

```
UPDATE inventory SET price=13.50 WHERE name='Hammer'
```

- This would set the price field for only the records where the name field is 'Hammer' to 13.50

```
UPDATE inventory SET price=12.99, quantity=100 WHERE name='Wrench'
```

- This would set the price and quantity for only the records where the name field is 'Wrench' to 12.99 (price) and 100 (quantity)

Changing Records

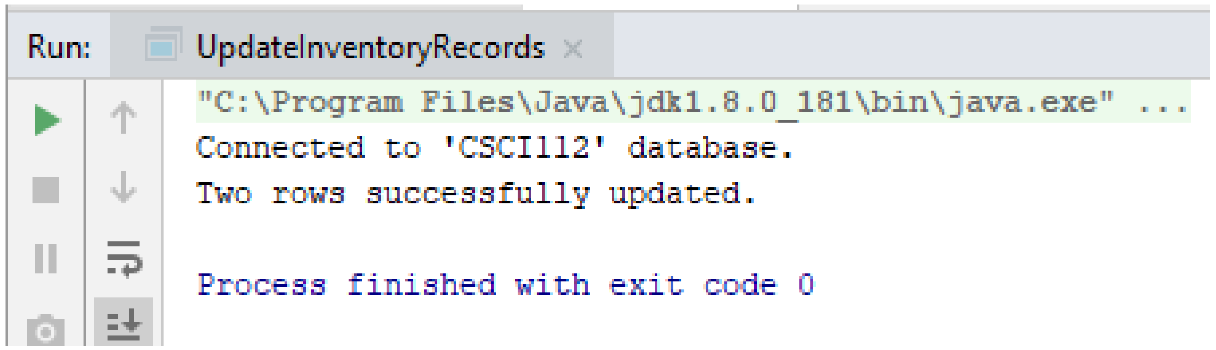
- If there are no matching rows to update, then nothing in the table will be changed.
  - It also won't throw an exception.

- Use the Statement object's executeUpdate method for UPDATE statements, just as you did for INSERT statements.

```
stmt.executeUpdate("UPDATE inventory SET price=13.50 WHERE name='Hammer'");
```

Changing Records

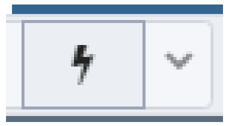
- Review and run the UpdateInventoryRecords.java program to update the previously shown records in the Inventory table in your Module8 database.



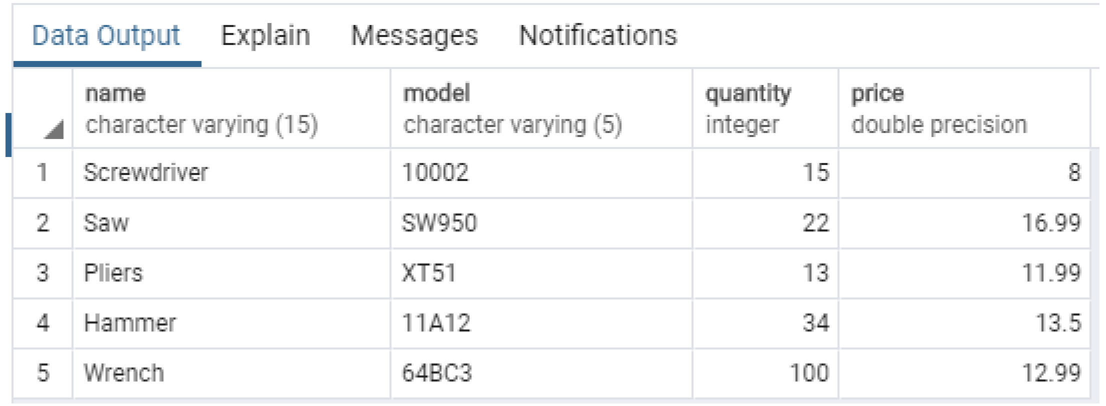
```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Connected to 'CSCI112' database.
Two rows successfully updated.

Process finished with exit code 0
```

Changing Records

- Back in pgAdmin, click the Execute/Refresh button.



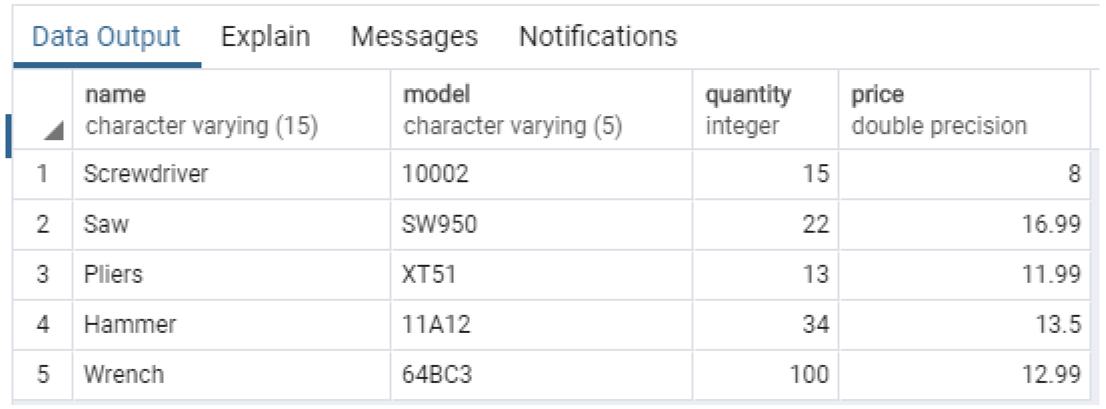
- The inventory table should now show the updated records/rows.



	name character varying (15)	model character varying (5)	quantity integer	price double precision
1	Screwdriver	10002	15	8
2	Saw	SW950	22	16.99
3	Pliers	XT51	13	11.99
4	Hammer	11A12	34	13.5
5	Wrench	64BC3	100	12.99

Changing Records

- The order of rows doesn't matter in a relational database.
- You'll see that Hammer (which was listed as Row 1 earlier) is now Row 4.
  - Never assume any particular row will always be in the same place.
  - This is why everything in tables are treated in terms of individual "records" instead of persistent "rows"



	name character varying (15)	model character varying (5)	quantity integer	price double precision
1	Screwdriver	10002	15	8
2	Saw	SW950	22	16.99
3	Pliers	XT51	13	11.99
4	Hammer	11A12	34	13.5
5	Wrench	64BC3	100	12.99

Changing Records

- More Information:
  - https://www.postgresql.org/docs/11/dml-update.html

Retrieving Records

- A SELECT statement is used to **query** (retrieve records from) a database table.
  - (Optionally) Uses the WHERE keyword
- Must specify:
  - The table to query.
  - The field(s) to retrieve.
  - Optional condition indicating which records to retrieve.

Retrieving Records

```
SELECT * FROM inventory
```

- This query would retrieve ALL records (and ALL columns for every record) from the inventory table
  - The * in SQL means “all columns”

Screwdriver	10002	15	8.00
Saw	SW950	22	16.99
Pliers	XT51	13	11.99
Hammer	11A12	34	13.5
Wrench	64BC3	100	12.99

Retrieving Records

```
SELECT name FROM inventory
```

- This query would retrieve only the name column from all records in the inventory table

Screwdriver
Saw
Pliers
Hammer
Wrench

Retrieving Records

```
SELECT name, price FROM inventory
```

- This query would retrieve <u>only the name and price columns</u> from all records in the inventory table

Screwdriver	8.00
Saw	16.99
Pliers	11.99
Hammer	13.5
Wrench	12.99

Retrieving Records

```
SELECT quantity, name FROM inventory
```

- This query would retrieve only the quantity and name columns from all records in the inventory table
  - Columns do not need to be selected in the same order as they appear in the table

15	Screwdriver
22	Saw
13	Pliers
34	Hammer
100	Wrench

Retrieving Records

```
SELECT * FROM inventory WHERE price < 12
```

- This query would retrieve any records (and all columns for every record) from the inventory table where the price field is less than 12

Screwdriver	10002	15	8.00
Pliers	XT51	13	11.99

Retrieving Records

```
SELECT name FROM inventory WHERE quantity > 30
```

- This query would retrieve only the name column from any records in the inventory table where the quantity field is greater than 30

Hammer
Wrench

Retrieving Records

```
SELECT * FROM inventory WHERE price < 12 AND quantity <= 14
```

- This query would retrieve any records (and all columns for every record) from the inventory table where the price field is less than 12 and the quantity field is less than or equal to 14

Pliers	XT51	13	11.99

Retrieving Records

```
SELECT * FROM inventory WHERE quantity > 200
```

- This query would retrieve any records (and all columns for every record) from the inventory table where the quantity field is greater than 200
  - There are no such records in the table, so 0 records would be selected.

Retrieving Records

```
SELECT * FROM inventory WHERE name = 'Saw'
```

- This query would retrieve any records (and all columns for every record) from the inventory table where the name field is equal to Saw
  - Case sensitive
  - =, not ==

Saw	SW950	22	16.99

Retrieving Records

```
SELECT * FROM inventory WHERE quantity = 15
```

- This query would retrieve any records (and all columns for every record) from the inventory table where the quantity field is equal to 15

Screwdriver	10002	15	8.00

Retrieving Records

```
SELECT * FROM inventory WHERE price <> 11.99
```

- This query would retrieve any records (and all columns for every record) from the inventory table where the price field is <u>not equal</u> to 11.99
  - Could also use !=

Screwdriver	10002	15	8.00
Saw	SW950	22	16.99
Hammer	11A12	34	13.5
Wrench	64BC3	100	12.99

Result Sets

- To retrieve records from a table, use the Statement object’s executeQuery method.

```
import java.sql.ResultSet;
```

- When you execute a SELECT statement using the executeQuery method, it returns a ResultSet object.
  - The ResultSet contains the information selected from the table.

```
ResultSet rs = stmt.executeQuery("SELECT * from inventory");
```

Result Sets

- The ResultSet returned does not initially point to any record.
- The ResultSet’s next method advances to the next record, sequentially.
    - In this case, it advances to the first record in the ResultSet

```
ResultSet rs = stmt.executeQuery("SELECT * from inventory");
rs.next();
```

Result Sets

- ResultSet’ have various get methods for retrieving the values in a record.
  - These methods accept Strings, which are the name of the column.
  - There are no equivalent “setter” methods. We don’t update records in a table with a ResultSet.

- Common methods:
  - getInt("column_name")
  - getDouble("column_name")
  - getString("column_name")

https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html#method.summary

Result Sets

```java
ResultSet rs = stmt.executeQuery("SELECT * from inventory");
rs.next();

double price = rs.getDouble("price");
String name = rs.getString("name");
String model = rs.getString("model");
int quantity = rs.getInt("quantity");
```

- Order doesn’t matter when “getting” the fields from the ResultSet
- An SQLException will be thrown if:
  - The ResultSet is empty (no records where selected)
  - The specified column name String doesn’t exist/wasn’t selected in the query.

Result Sets

- Be sure to close the ResultSet when finished by calling its close method.

```
rs.close();
```

Result Sets

```
ResultSet rs = stmt.executeQuery("SELECT * from inventory");
rs.next();

double price = rs.getDouble("price");
String name = rs.getString("name");
String model = rs.getString("model");
int quantity = rs.getInt("quantity");

System.out.println("Name:  " + name);
System.out.println("Model:  " + model);
System.out.println("Quantity:  " + quantity);
System.out.println("Price: $" + price);

rs.close();
```

This example demonstrates only retrieving and printing the first record in the ResultSet

Iterating through Result Sets

- The ResultSet’s next method returns a boolean when called.
  - Will return true if there was a record to advance to.
  - Will return false if there were no more records to advance to.

- Can be used as the condition of a while loop.

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

}
```

Iterating through Result Sets

```
ResultSet rs = stmt.executeQuery("SELECT * from inventory");

while(rs.next()) {
    double price = rs.getDouble("price");
    String name = rs.getString("name");
    String model = rs.getString("model");
    int quantity = rs.getInt("quantity");

    System.out.println("Name: " + name);
    System.out.println("Model: " + model);
    System.out.println("Quantity: " + quantity);
    System.out.println("Price: $" + price);
}

rs.close();
```

Testing if a Result Set is empty

- The ResultSet's isBeforeFirst method returns a boolean when called.
  - Will return true for a new ResultSet if there is at least one record.
    - The pointer is currently before the first row.
  - Will return false if there is no first record to advance to.

- Can be used as the condition of an if statement.

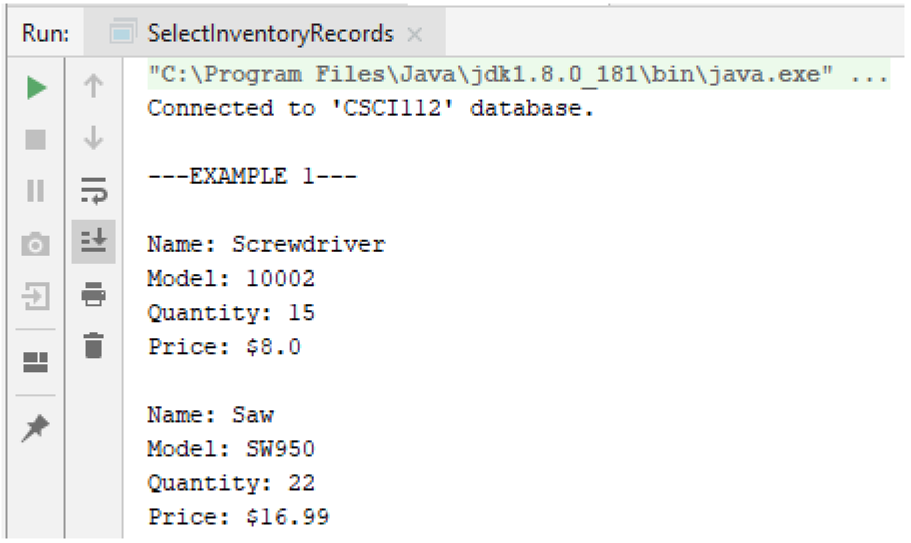
```
ResultSet rs = stmt.executeQuery("SELECT * from inventory");
if(rs.isBeforeFirst()) {
   //Ok to advance (next()) to the first record
}
```

Testing if a Result Set is empty

```
ResultSet rs = stmt.executeQuery("SELECT * from inventory");

if(rs.isBeforeFirst()) {
     while(rs.next()) {
          double price = rs.getDouble("price");
          String name = rs.getString("name");
          String model = rs.getString("model");
          int quantity = rs.getInt("quantity");

          System.out.println("Name: " + name);
          System.out.println("Model: " + model);
          System.out.println("Quantity: " + quantity);
          System.out.println("Price: $" + price);
     }
}
else {
     System.out.println("No records selected.");
}

rs.close();
```

Retrieving Records

- Review and run the SelectInventoryRecords.java program.
  - The program contains multiple methods that each demonstrate selecting records from the Inventory table in your Module8 database.



```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Connected to 'CSCI112' database.

---EXAMPLE 1---

Name: Screwdriver
Model: 10002
Quantity: 15
Price: $8.0

Name: Saw
Model: SW950
Quantity: 22
Price: $16.99
```

Retrieving Records

- More Information:
  - https://www.postgresql.org/docs/11/queries.html

Deleting Records

- A DELETE statement is used to remove records from a database table.
    - (Optionally) Uses the WHERE keyword
- Must specify:
    - The table to remove records from.
    - Optional condition indicating which records to delete.

Deleting Records

```
DELETE FROM inventory
```

- This statement would delete ALL records from the inventory table

```
DELETE FROM inventory WHERE name = 'Hammer'
```

- This statement would delete any records from the inventory table where the name field is equal to 'Hammer’

```
DELETE FROM inventory WHERE quantity < 5 AND price <= 10.00
```

- This statement would delete any records from the inventory table where the quantity field less than 5 and the price field is less than or equal to 10

Deleting Records

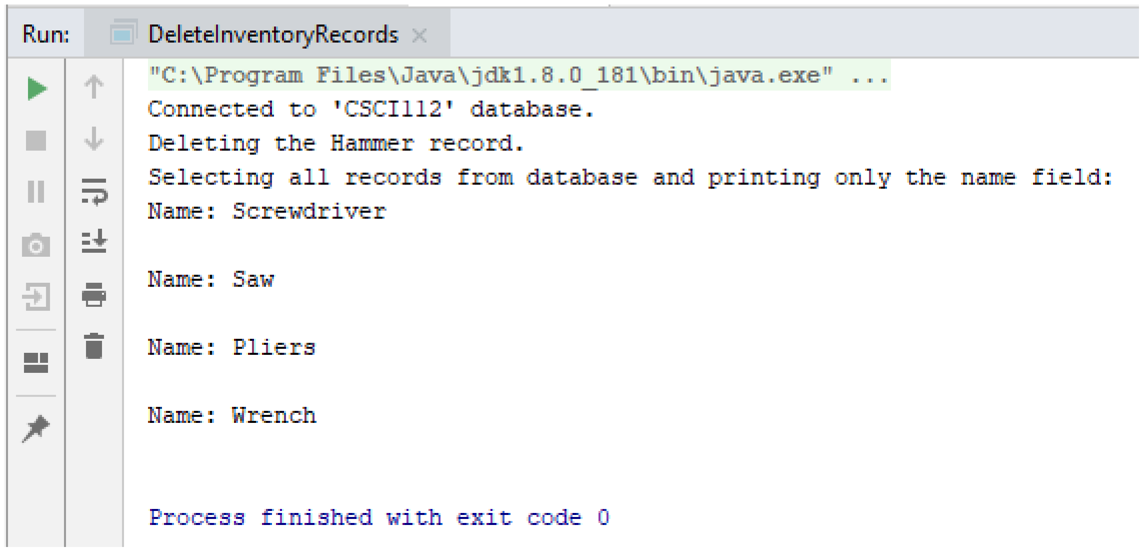
- If there are no matching rows to delete, then nothing in the table will be deleted.
  - It also won’t throw an exception.

- Use the Statement object’s executeUpdate method for DELETE statements, just as you did for INSERT and UPDATE statements.

```
stmt.executeUpdate("DELETE FROM inventory WHERE name = 'Hammer'");
```

Deleting Records

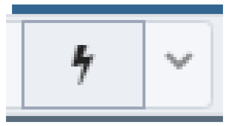
- Review and run the DeleteInventoryRecords.java program to delete a record from the Inventory table in your Module8 database.



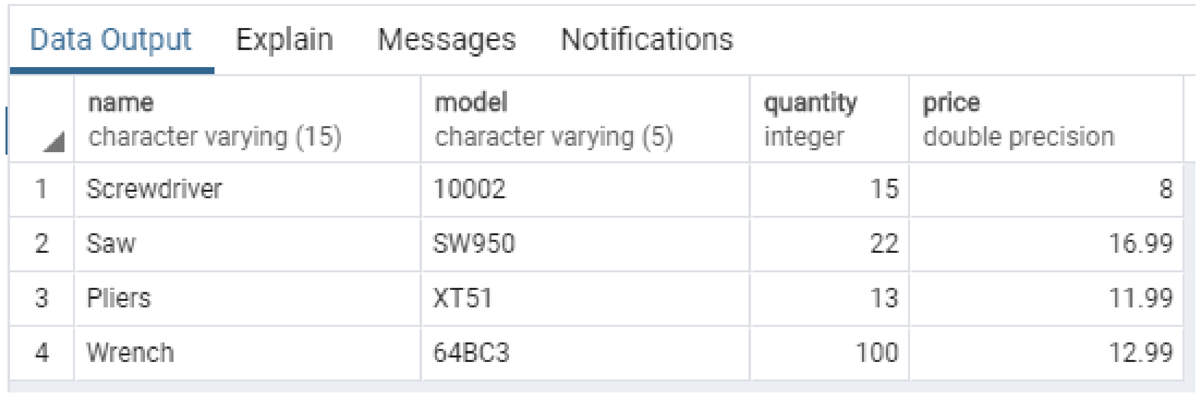
```
"C:\Program Files\Java\jdk1.8.0_181\bin\java.exe" ...
Connected to 'CSCI112' database.
Deleting the Hammer record.
Selecting all records from database and printing only the name field:
Name: Screwdriver

Name: Saw

Name: Pliers

Name: Wrench


Process finished with exit code 0
```

Deleting Records

- Back in pgAdmin, click the Execute/Refresh button.



- The inventory table should now only 4 records/rows.



Data Output | Explain | Messages | Notifications

	name character varying (15)	model character varying (5)	quantity integer	price double precision
1	Screwdriver	10002	15	8
2	Saw	SW950	22	16.99
3	Pliers	XT51	13	11.99
4	Wrench	64BC3	100	12.99

Deleting Records

- More Information:
  - https://www.postgresql.org/docs/11/dml-delete.html

- Full PostgreSQL Documentation Website:
  - https://www.postgresql.org/docs/11/index.html

Need/want to start this over?

- Run the DropInventoryTable.java program to delete the inventory table from the database.

- Re-run the CreateInventoryTable.java program to re-create the inventory table.