# Getting Columns in a `ResultSet` Object

| `ResultSet` Method | Description |
|---|---|
| *`double getDouble(int colNumber)`*<br>*`double getDouble(String colName)`* | Returns the `double` that is stored in the column specified by *`colNumber`* or *`colName`*. The column must hold data that is compatible with the `double` data type in Java. If an error occurs, the method throws an `SQLException`. |
| *`int getInt(int colNumber)`*<br>*`int getInt(String colName)`* | Returns the `int` that is stored in the column specified by *`colNumber`* or *`colName`*. The column must hold data that is compatible with the `int` data type in Java. If an error occurs, the method throws an `SQLException`. |
| *`String getString(int colNumber)`*<br>*`String getString(String colName)`* | Returns the string that is stored in the column specified by *`colNumber`* or *`colName`*. The column must hold data that is compatible with the `String` type in Java. If an error occurs, the method throws an `SQLException`. |

# Specifying Search Criteria with the `WHERE` clause

- **The `WHERE` clause can be used with the `SELECT` statement to specify a search criteria**

```
SELECT Columns FROM Table WHERE Criteria
```

  - *`Criteria`* is a conditional expression
- **Example:**

```
SELECT * FROM Coffee WHERE Price > 12.00
```

| Description | ProdNum | Price |
|---|---|---|
| Kona Medium | 18-001 | 18.45 |
| Kona Dark | 18-002 | 18.45 |

- **Only the rows that meet the search criteria are returned in the result set**
- **A *result set* is an object that contains the results of an SQL statement**

# SQL Relational Operators

- **Standard SQL supports the following relational operators:**

| Operator | Meaning |
|---|---|
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| = | Equal to |
| <> | Not equal to |

- **Notice a few SQL relational operators are different than in Java**
  - SQL equal to operator is =
  - SQL not equal to operator is <>

Example: CoffeeMinPrice.java

# String Comparisons in SQL

- **Example 1:**

```
SELECT * FROM Coffee WHERE Description = 'French Roast Dark'
```

  - In SQL, strings are enclosed in single quotes
- **Warning!**

```
SELECT * FROM Coffee WHERE Description = 'french roast dark'
```

  - String comparisons in SQL are case sensitive
- **Example 2:**

```
SELECT * FROM Coffee
   WHERE UPPER(Description) = 'FRENCH ROAST DARK'
```

  - The `UPPER()` or `LOWER()` functions convert the string to uppercase or lowercase and can help prevent case sensitive errors when comparing strings
- **Example 3:**

```
SELECT * FROM Coffee WHERE Description ='Joe''s Special Blend'
```

  - If a single quote (`'`) is part of a string, use two single quotes (`''`)

# Using the `LIKE` Operator

- **In SQL, the `LIKE` operator can be used to search for a substring**
- **Example 1:**

```
SELECT * FROM Coffee WHERE Description LIKE '%Decaf%'
```

  - The `%` symbol is used as a wildcard for multiple characters
- **Example 2:**

```
SELECT * FROM Coffee WHERE ProdNum LIKE '2_-00_'
```

  - The underscore `(_)` is a used as a wildcard for a single character
- **Example 3:**

```
SELECT * FROM Coffee
   WHERE Description NOT LIKE '%Decaf%'
```

  - The `NOT` operator is used to disqualify the search criteria

# Using AND and OR

- **The AND and OR operators can be used to specify multiple search criteria in a WHERE clause**
- **Example 1:**

```
SELECT * FROM Coffee
   WHERE Price > 10.00 AND Price < 14.00
```

  - The AND operator requires that *both* search criteria be true
- **Example 2:**

```
SELECT * FROM Coffee
   WHERE Description LIKE '%Dark%' OR ProdNum LIKE '16%'
```

  - The OR operator requires that *either* search criteria be true

# Sorting the Results of a `SELECT` Query

- **Use the `ORDER BY` clause to sort results according to a column value**
- **Example 1:**

```
SELECT * FROM Coffee ORDER BY Price
```

  - Sorted in ascending order (`ASC`) by default
- **Example 2:**

```
SELECT * FROM Coffee
WHERE Price > 9.95 ORDER BY Price DESC
```

  - Use the `DESC` operator to sort results in descending order

# Mathematical Functions

- **The `AVG` function**
  - calculates the average value in a particular column

```
SELECT AVG(Price) FROM Coffee
```

- **The `SUM` function**
  - calculates the sum of a column's values

```
SELECT SUM(Price) FROM Coffee
```

- **The `MIN` and `MAX` functions**
  - calculate the minimum and maximum values found in a column

```
SELECT MIN(Price) FROM Coffee

SELECT MAX(Price) FROM Coffee
```

- **The `COUNT` function**
  - can be used to determine the number of rows in a table

```
SELECT COUNT(*) FROM Coffee
```

# Inserting Rows

- **In SQL, the `INSERT` statement inserts a row into a table**

```
INSERT INTO TableName VALUES (Value1, Value2, ...)
```

  - *`TableName`* is the name of the database table
  - *`Value1, Value2, ...`* is a list of column values
- **Example:**

```
INSERT INTO Coffee
VALUES ('Honduran Dark', '22-001', 8.65)
```

  - Strings are enclosed in single quotes
  - Values appear in the same order as the columns in the table
- **Inserts a new row with the following column values:**

```
Description: Honduran Dark
ProdNum: 22-001
Price: 8.65
```

# Inserting Rows

- **If column order is uncertain, the following general format can be used**

```
INSERT INTO TableName
    (ColumnName1, ColumnName2, ...)
VALUES
    (Value1, Value2, ...)
```

  - *ColumnName1, ColumnName2, ...* is a list of column names
  - *Value1, Value2, ...* is a list of corresponding column values
- **Example:**

```
INSERT INTO Coffee
    (ProdNum, Price, Description)
VALUES
    ('22-001', 8.65, 'Honduran Dark')
```

- **Keep in mind that primary key values must be unique**
- **For example, a duplicate `ProdNum` is not allowed in the `Coffee` table**

# Inserting Rows with JDBC

- **To issue an `INSERT` statement, you must get a reference to a `Statement` object**
  - The `Statement` object has an `executeUpdate` method
  - Accepts a string containing the SQL `INSERT` statement as an argument
  - Returns an `int` value for the number of rows inserted
- **Example:**

```
String sqlStatement = "INSERT INTO Coffee " +
                      "(ProdNum, Price, Description)" +
                      " VALUES " +
                      "('22-001', 8.65, 'Honduran Dark')";

int rows = stmt.executeUpdate(sqlStatement);
```

- **`rows` should contain the value `1`, indicating that one row was inserted**

# Updating an Existing Row

- **In SQL, the `UPDATE` statement changes the contents of an existing row in a table**

```
UPDATE Table
   SET Column = Value
   WHERE Criteria
```

  - *`Table`* is a table name
  - *`Column`* is a column name
  - *`Value`* is the value to store in the column
  - *`Criteria`* is a conditional expression
- **Example:**

```
UPDATE Coffee
   SET Price = 9.95
   WHERE Description = 'Galapagos Organic Medium'
```

# Updating More Than One Row

- **It is possible to update more than one row**
- **Example:**

```
UPDATE Coffee
   SET Price = 12.95
   WHERE ProdNum LIKE '21%'
```

  - Updates the price of all rows where the product number begins with 21
- **Warning!**

```
UPDATE Coffee
   SET Price = 4.95
```

  - Because this statement does not have a `WHERE` clause, it will change the price for every row

# Updating Rows with JDBC

- **To issue an `UPDATE` statement, you must get a reference to a `Statement` object**
  - The `Statement` object has an `executeUpdate` method
  - Accepts a string containing the SQL `UPDATE` statement as an argument
  - Returns an `int` value for the number of rows affected
- **Example:**

```
String sqlStatement = "UPDATE Coffee " +
                      "SET Price = 9.95" +
                      " WHERE " +
                      "Description = 'Brazilian Decaf'";

int rows = stmt.executeUpdate(sqlStatement);
```

- **`rows` indicates the number of rows that were changed**

# Deleting Rows with the `DELETE` Statement

- **In SQL, the `DELETE` statement deletes one or more rows in a table**

  `DELETE FROM` *`Table`* `WHERE` *`Criteria`*

  - *`Table`* is the table name
  - *`Criteria`* is a conditional expression
- **Example 1:**

  ```
  DELETE FROM Coffee WHERE ProdNum = '20-001'
  ```

  - Deletes a single row in the `Coffee` table where the product number is 20-001
- **Example 2:**

  ```
  DELETE FROM Coffee WHERE Description LIKE 'Sumatra%'
  ```

  - Deletes all rows in the `Coffee` table where the description begins with Sumatra
- **Warning!**

  ```
  DELETE FROM Coffee
  ```

  - Because this statement does not have a `WHERE` clause, it will delete every row in the `Coffee` table

# Deleting Rows with JDBC

- **To issue a `DELETE` statement, you must get a reference to a `Statement` object**
  - The `Statement` object has an `executeUpdate` method
  - Accepts a string containing the SQL `DELETE` statement as an argument
  - Returns an `int` value for the number of rows that were deleted
- **Example:**

```
String sqlStatement = "DELETE FROM Coffee " +
                      "WHERE ProdNum = '20-001'";
int rows = stmt.executeUpdate(sqlStatement);
```

- **`rows` indicates the number of rows that were deleted**

# Creating Tables with the `CREATE TABLE` Statement

- **In SQL, the `CREATE TABLE` statement adds a new table to the database**

```
CREATE TABLE TableName
    (ColumnName1 DataType1,
     ColumnName2 DataType2, ...)
```

  - *`TableName`* is the name of the table
  - *`ColumnName1`* is the name of the first column
  - *`DataType1`* is the SQL data type for the first column
  - *`ColumnName2`* is the name of the second column
  - *`DataType2`* is the SQL data type for the second column
- **Example:**

```
CREATE TABLE Customer
   ( Name CHAR(25), Address CHAR(25),
     City CHAR(12), State CHAR(2), Zip CHAR(5) )
```

  - Creates a new table named `Customer` with the columns `Name`, `Address`, `City`, `State`, and `Zip`

# Creating Tables with the `CREATE TABLE` Statement

- **The `PRIMARY KEY` qualifier is used to specify a column as the primary key**
- **The `NOT NULL` qualifier is used to specify that the column must contain a value for every row**
  - Qualifiers should be listed *after* the column's data type

```
CREATE TABLE Customer
    ( CustomerNumber CHAR(10) NOT NULL PRIMARY KEY
      Name CHAR(25), Address CHAR(25),
      City CHAR(12), State CHAR(2), Zip CHAR(5) )
```

  - Creates a new table named `Customer` with the columns `CustomerNumber`, which is the primary key, `Name`, `Address`, `City`, `State`, and `Zip`

# Removing a Table with the `DROP TABLE` Statement

- **In SQL, the `DROP TABLE` statement deletes an existing table from the database**

```
DROP TABLE TableName
```

  - *`TableName`* is the name of the table you wish to delete
- **Example:**

```
DROP TABLE Customer
```

  - Deletes the `Customer` table from the `CoffeeDB` database
  - Useful if you make a mistake creating a table
  - Simply delete the table and recreate

# Scrollable Result Sets

- **By default, a `ResultSet` object is created with a read-only concurrency level and the cursor is limited to forward movement**
- **A *scrollable result set* can be created with the overloaded version the `Connection` object's `createStatement` method**

```
conn.createStatement(type, concur);
```

  - *`type`* is a constant for the scrolling type
  - `concur` is a constant for the concurrency level
- **Example:**

```
Statement stmt =
   conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                        ResultSet.CONCUR_READ_ONLY);
```

  - Creates a scrollable result set that is read-only and insensitive to database changes

# The `ResultSet` Scrolling Types

- **`ResultSet.TYPE_FORWARD_ONLY`**
  - Default scrolling type
  - Cursor moves forward only
- **`ResultSet.TYPE_SCROLL_INSENSITIVE`**
  - Cursor moves both forward and backward
  - Changes made to the database do not appear
- **`ResultSet.TYPE_SCROLL_SENSITIVE`**
  - Cursor moves both forward and backward
  - Changes made to the database appear as soon as they are made

# The `ResultSet` Concurrency Levels

- **`ResultSet.CONCUR_READ_ONLY`**
  - Default concurrency level
  - Read-only version of data from the database
  - Cannot change database by altering result set
- **`ResultSet.CONCUR_UPDATEABLE`**
  - Result set is updateable
  - Changes can be made to the result set and saved to the database
  - Uses methods that allow changes to be made to the database without issuing SQL statements

# `ResultSet` Navigation Methods

- **`first()`**
  - Moves the cursor to the first row
- **`last()`**
  - Moves the cursor to the last row
- **`next()`**
  - Moves the cursor to the next row
- **`previous()`**
  - Moves the cursor to the previous row

# `ResultSet` Navigation Methods

- **`relative(`*`rows`*`)`**
  - Moves the cursor the number specified by the *`rows`* argument relative to the current row
    - A positive *`rows`* value will move the cursor forward
    - A negative *`rows`* value will move the cursor backward
- **`absolute(`*`rows`*`)`**
  - Moves the cursor to the row number specified by the *`rows`* argument
    - A `rows` value of 1 will move the cursor to the first row
    - A `rows` value of 2 will move cursor to the second row
    - And so on until the last row

# Determining the Number of Rows in a Result Set

- **`ResultSet` navigation methods can be used to determine the number of rows in a result set**
- **Example:**

```
resultSet.last()                    // Move to the last row
int numRows = resultSet.getRow();   // Get the current row number
resultSet.first();                  // Move back to the first row
```

  - Move cursor to last row
  - Get the last row's number and store the value
  - Move back to the first row

# Result Set Metadata

- **Metadata refers to data that describes other data**
- **A `ResultSet` object has metadata that describes a result set**
- **Can be used to determine many things about a result set**
  - Number of columns
  - Column names
  - Column data types
  - And much more
- **Useful for submitting SQL queries in applications**

# Result Set Metadata

- `ResultSetMetaData` is an interface in the `java.sql` package
- **The `getMetaData` method of a `ResultSet` object returns a reference to a `ResultSetMetaData` object.**

```
ResultSetMetaData meta = resultSet.getMetaData();
```

- Creates a `ResultSetMetaData` object reference variable named `meta`

# A Few `ResultSetMetaData` Methods

| Method | Description |
|---|---|
| `int getColumnCount()` | Returns the number of columns in the result set. |
| `String getColumnName(int col)` | Returns the name of the column specified by the integer *`col`*. The first column is column 1. |
| `String getColumnTypeName(int col)` | Returns the name of the data type of the column specified by the integer *`col`*. The first column is column 1. The data type name returned is the database-specific SQL data type. |
| `int getColumnDisplaySize(int col)` | Returns the display width, in characters, of the column specified by the integer *`col`*. The first column is column 1. |
| `String getTableName(int col)` | Returns the name of the table associated with the column specified by the integer *`col`*. The first column is column 1. |

# The `JTable` Class

- **The `JTable` class is a Swing component that displays data in a two-dimensional table**

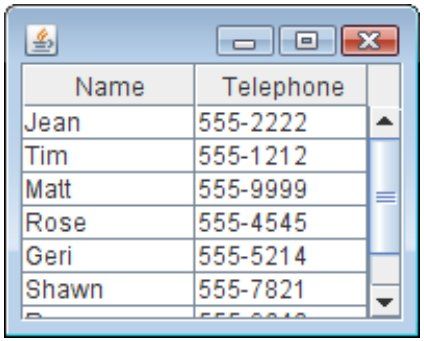
  `Jtable(Object[][] rowData, Object[] colNames)`

  - *`rowData`* is a two-dimensional array of objects
    - Contains data that will be displayed in the table
    - Each row becomes a row of data in the table
    - Each column becomes a column of data in the table
    - `JTable` calls `toString` method of each object to get values
  - `colNames` is a one-dimensional array of objects
    - Contains the column names to display
    - `JTable` calls `toString` method of each object to get value

# Setting Up a Simple `JTable` Component

- **Example:**

```
String[] colNames = {"Name", "Telephone" };
String[][] rowData = {{ "Jean",  "555-2222" },
                       { "Tim",   "555-1212" },
                       { "Matt",  "555-9999" },
                       { "Rose",  "555-4545" },
                       { "Geri",  "555-5214" },
                       { "Shawn", "555-7821" },
                       { "Renee", "555-9640" },
                       { "Joe",   "555-8657" } };
```



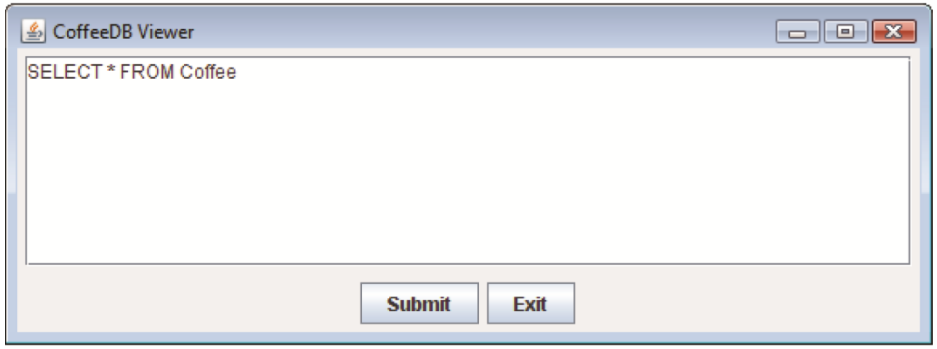

```
JTable myTable = new JTable(rowData, colNames);
JScrollPane scrollPane = new JScrollPane(JTable);
```

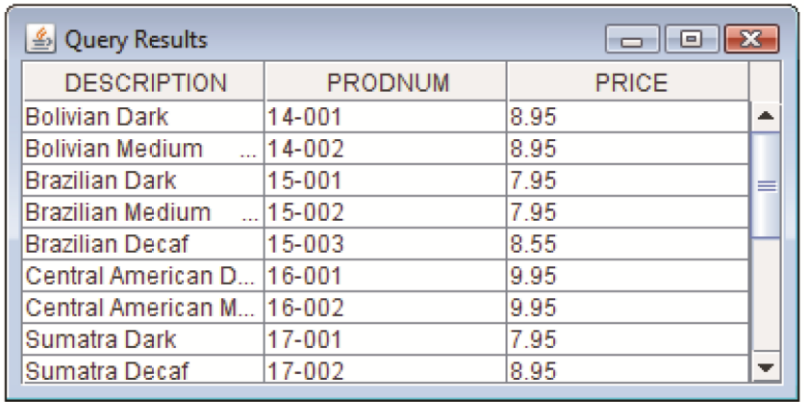
- The figure shows an example of how the table appears in a frame

# Displaying Query Results in a `JTable`

This window appears first

The user enters a `SELECT` statement and clicks the Submit button




| DESCRIPTION | PRODNUM | PRICE |
|---|---|---|
| Bolivian Dark | 14-001 | 8.95 |
| Bolivian Medium ... | 14-002 | 8.95 |
| Brazilian Dark | 15-001 | 7.95 |
| Brazilian Medium ... | 15-002 | 7.95 |
| Brazilian Decaf | 15-003 | 8.55 |
| Central American D... | 16-001 | 9.95 |
| Central American M... | 16-002 | 9.95 |
| Sumatra Dark | 17-001 | 7.95 |
| Sumatra Decaf | 17-002 | 8.95 |

This window appears next

It displays the results in a `Jtable` component

- **Example: TableFormatter.java, CoffeeDBQuery.java, CoffeeDBViewer.java**

# Joining Data from Multiple Tables

- **In SQL, you must use qualified column names in a `SELECT` statement if the tables have columns with the same name**
- **A *qualified column name* takes the following form:**

*`TableName.ColumnName`*

- **Example:**

```
SELECT
   Customer.CustomerNumber, Customer.Name,
   UnpaidOrder.OrderDate, UnpaidOrder.Cost,
   Coffee.Description
FROM
   Customer, UnpaidOrder, Coffee
WHERE
   UnpaidOrder.CustomerNumber = Customer.CustomerNumber
   AND
   UnpaidOrder.ProdNum = Coffee.ProdNum
```

- The search criteria tell the DBMS how to link the rows in the tables

# Transactions

- **An operation that requires multiple database updates is known as a *transaction*.**
- **For a transaction to be complete**
  - All of the steps involved in the transaction must be performed.
- **If any single step within a transaction fails**
  - None of the steps in the transaction should be performed.
- **When you write transaction-processing code, there are two concepts you must understand:**
  - Commit
  - Rollback
- **The term *commit* refers to making a permanent change to a database**
- **The term *rollback* refers to undoing changes to a database**

# JDBC Auto Commit Mode

- **By default, the JDBC `Connection` class operates in auto commit mode.**
- **In *auto commit* mode**
  - All updates that are made to the database are made permanent as soon as they are executed.
- **When auto commit mode is turned off**
  - Changes do not become permanent until a commit command is executed
  - A rollback command can be used to undo changes

# JDBC Transaction Methods

- **To turn auto commit mode off**
  - Call the `Connection` class's `setAutoCommit` method
  - Pass the argument `false`

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

- **To execute a commit command**
  - Call the `Connection` class's `commit` method

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

- **To execute a rollback command**
  - Call the `Connection` class's

```
conn.rollback();
```

# JDBC Transaction Example

```
conn.setAutoCommit(false);
// Attempt the transaction
try
{
    // Update the inventory records.
    stmt.executeUpdate(updateStatement);
    // Add the order to the UnpaidOrder table.
    stmt.executeUpdate(insertStatement);
    // Commit all these updates.
    conn.commit();
}
catch (SQLException ex)
{
    // Roll back the changes.
    conn.rollback();
}
```

The commit method is called in the try block

The rollback method is called in the catch block

# Stored Procedures

- **Many commercial database systems allow you to create SQL statements and store them in the DBMS itself**
- **These SQL statements are called *stored procedures***
  - Can be executed by other applications using the DBMS
  - Ideal for SQL statements that are used often in a variety of applications
  - Usually execute faster than SQL statements that are submitted from applications outside the DBMS
- **Each DBMS has its own syntax for creating a stored procedure in SQL**
- **To execute a stored procedure, you must create a `CallableStatement` object**
- **`CallableStatement` is an interface in the `java.sql` package**
- **To create a `CallableStatement` object, you call the `Connection` class's `prepareCall` statement**