Chapter 16 discusses the following main topics:

- Introduction to Database Management Systems
- Tables, Rows, and Columns
- Introduction to the SQL \texttt{SELECT} Statement
- Inserting Rows
- Updating and Deleting Existing Rows
- Creating and Deleting Tables
- Creating a New Database with JDBC
Chapter Topics (2)

– Scrollable Result Sets
– Result Set Metadata
– Displaying Query Results in a JTable
– Relational Data
– Advanced Topics
Storing data in traditional text or binary files has its limits

- well suited for applications that store only a small amount of data
- not practical for applications that must store a large amount of data
- simple operations become cumbersome and inefficient as data increases
Introduction to Database Management Systems

• A *database management system (DBMS)* is software that is specifically designed to work with large amounts of data in an efficient and organized manner
  – Data is stored using the database management system
  – Applications written in Java or other languages communicate with the DBMS rather than manipulate the data directly
  – DBMS carries out instructions and sends the results back to the application
A Java Application Interacts with a DBMS, Which Manipulates Data

The Application sends a command to the DBMS

The DBMS executes the command on the Data

The DBMS sends the result back to the Application

The Application displays the result to the user
JDBC Provides Connectivity to the DBMS

- JDBC stands for *Java database connectivity*
- It is the technology that makes communication possible between the Java application and DBMS
- The Java API contains numerous JDBC classes that allow your Java applications to interact with a DBMS
SQL Sends Commands to the DBMS

- SQL stands for *structured query language*
- A standard language for working with database management systems
- Not used as a general programming language
- Consists of several key words, used to construct statements known as *queries*
- Statements or queries are strings passed from the application to the DBMS using API method calls
- Serve as instructions for the DBMS to carry out operations on its data
JDBC Needs a DBMS

• To use JDBC to work with a database you will need a DBMS
  – Java DB
  – Oracle
  – Microsoft SQL Server
  – DB2
  – MySQL
• The examples in this chapter were created with Java DB
JDBC Classes

• Java comes with a standard set of JDBC classes
  – java.sql and javax.sql

• Using JDBC in a Java application requires the following steps
  1. Get a connection to the database
  2. Pass a string containing an SQL statement to the DBMS
  3. If the SQL statement has results to send back, they will be sent back as a result set
  4. When finished working with the database, close the connection
Getting a Database Connection

- The static `DriverManager.getConnection` method is used to get a connection to the database
  - General format of the simplest version:
    ```java
    DriverManager.getConnection(DatabaseURL);
    ```
  - General format if a user name and a password are required:
    ```java
    DriverManager.getConnection(DatabaseURL, Username, Password);
    ```
  
  - `Username` is a string containing a valid username
  - `Password` is a string containing a password
  - `DatabaseURL` lists the protocol used to access the database
Getting a Database Connection

- **DatabaseURL** is a string known as a database URL
  - URL stands for uniform resource locator
- A simple database URL has the following general format:
  \[
  \text{protocol}:\text{subprotocol}:\text{databaseName}
  \]
  - **protocol** is the database protocol
    - value is **jdbc** when using JDBC
  - **subprotocol** varies depending on the type of DBMS
    - value is **derby** when using Java DB
  - **databaseName** is the name of the database
- Using Java DB, the URL for the **CoffeeDB** database is:
  \[
  \text{jdbc:derby:CoffeeDB}
  \]
Getting a Database Connection

• The `DriverManager.getConnection` method
  – Searches for and loads a compatible JDBC driver for the database specified by the URL
  – Returns a reference to a `Connection` object
    • Should be saved in a variable, so it can be used later
  – Throws an `SQLException` if it fails to load a compatible JDBC driver

    Final String DB_URL = "jdbc:derby:CoffeeDB";
    Connection conn = DriverManager.getConnection(DB_URL);

• Example: `TestConnection.java`
Tables, Rows, and Columns

- A database management system stores data in a database
- A database is organized into one or more tables
- Each table holds a collection of related data, organized into rows and columns
- A row is a complete set of information about a single item, divided into columns
- Each column is an individual piece of information about the item
Database Organization

- Database
- Table
- Column
- Row
Each row contains data for a single item.

<table>
<thead>
<tr>
<th>Description</th>
<th>ProdNum</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bolivian Dark</td>
<td>14-001</td>
<td>8.95</td>
</tr>
<tr>
<td>Bolivian Medium</td>
<td>14-002</td>
<td>8.95</td>
</tr>
<tr>
<td>Brazilian Dark</td>
<td>15-001</td>
<td>7.95</td>
</tr>
<tr>
<td>Brazilian Medium</td>
<td>15-002</td>
<td>7.95</td>
</tr>
<tr>
<td>Brazilian Decaf</td>
<td>15-003</td>
<td>8.55</td>
</tr>
<tr>
<td>Central American Dark</td>
<td>16-001</td>
<td>9.95</td>
</tr>
<tr>
<td>Central American Medium</td>
<td>16-002</td>
<td>9.95</td>
</tr>
<tr>
<td>Sumatra Dark</td>
<td>17-001</td>
<td>7.95</td>
</tr>
<tr>
<td>Sumatra Decaf</td>
<td>17-002</td>
<td>8.95</td>
</tr>
<tr>
<td>Sumatra Medium</td>
<td>17-003</td>
<td>7.95</td>
</tr>
<tr>
<td>Sumatra Organic Dark</td>
<td>17-004</td>
<td>11.95</td>
</tr>
<tr>
<td>Kona Medium</td>
<td>18-001</td>
<td>18.45</td>
</tr>
<tr>
<td>Kona Dark</td>
<td>18-002</td>
<td>18.45</td>
</tr>
<tr>
<td>French Roast Dark</td>
<td>19-001</td>
<td>9.65</td>
</tr>
<tr>
<td>Galapagos Medium</td>
<td>20-001</td>
<td>6.85</td>
</tr>
<tr>
<td>Guatemalan Dark</td>
<td>21-001</td>
<td>9.95</td>
</tr>
<tr>
<td>Guatemalan Decaf</td>
<td>21-002</td>
<td>10.45</td>
</tr>
<tr>
<td>Guatemalan Medium</td>
<td>21-003</td>
<td>9.95</td>
</tr>
</tbody>
</table>
Column Data Types

- Columns in a database are assigned an SQL data type
  - SQL data types are generally compatible with Java data types

<table>
<thead>
<tr>
<th>SQL Data Type</th>
<th>Description</th>
<th>Corresponding Java Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER or INT</td>
<td>An integer number</td>
<td>int</td>
</tr>
<tr>
<td>CHARACTER (n) or CHAR (n)</td>
<td>A fixed-length string with a length of n characters.</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR (n)</td>
<td>A variable-length string with a maximum length of n characters.</td>
<td>String</td>
</tr>
<tr>
<td>REAL</td>
<td>A single-precision floating point number</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>A double-precision floating point number</td>
<td>double</td>
</tr>
<tr>
<td>DECIMAL (t, d)</td>
<td>A decimal value with t total digits and d digits appearing after the decimal point.</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DATE</td>
<td>A date</td>
<td>java.sql.Date</td>
</tr>
</tbody>
</table>
The Coffee Table Column Data Types

• **Description column data type is CHAR(25)**
  – String with a fixed length of 25 characters
  – Compatible with the `String` type in Java

• **ProdNum column data type is CHAR(10)**
  – String with a fixed length of 10 characters
  – Compatible with the `String` type in Java

• **Price column data type is DOUBLE**
  – Double-precision floating-point number
  – Compatible with the `double` data type in Java
Primary Keys

• A primary key is a column that holds a unique value for each row in a database table.

• In the Coffee table, ProdNum is the primary key:
  – Each type of coffee has a unique product number.
  – Used to identify any coffee stored in the table.

• A primary key can be the combination of several columns in a table.
Introduction to the SQL SELECT Statement

• The SELECT statement is used to retrieve the rows in a table

\[
\text{SELECT Columns FROM Table}
\]

  – Columns is one or more column names
  – Table is a table name

• Example 1:

\[
\text{SELECT Description FROM Coffee}
\]

• Example 2:

\[
\text{SELECT Description, Price FROM Coffee}
\]

  – Multiple column names are separated with a comma

• Example 3:

\[
\text{SELECT * FROM Coffee}
\]

  – The * character can be used to retrieve all columns in the table
More About SQL Statements

- SQL statements are free form
  - tabs, new lines, and spaces between key words are ignored
- SQL key words and table names are case insensitive
- Example: *The following statements all work the same:*

```sql
SELECT * FROM Coffee

SELECT *
FROM Coffee

select * from coffee
```
Passing an SQL Statement to the DBMS

• Once you have established a connection, you must get a reference to a `Statement` object before you can issue SQL statements to the DBMS
  – A `Statement` object has an `executeQuery` method that returns a reference to a `ResultSet` object
  – A `ResultSet` object contains the results of the query

• Example:

```java
Connection conn = DriverManager.getConnection(DB_URL);
Statement stmt = conn.createStatement();
String sqlStatement = "SELECT Description FROM Coffee";
ResultSet result = stmt.executeQuery(sqlStatement);
```
Getting a Row from the ResultSet Object

- A ResultSet object has an internal cursor
  - Points to a specific row in the ResultSet
  - The row to which it points is the current row
  - Initially positioned just before the first row
  - Can be moved from row to row to examine all rows

<table>
<thead>
<tr>
<th>Row</th>
<th>Description</th>
<th>Code</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sumatra Organic Dark</td>
<td>17-004</td>
<td>11.95</td>
</tr>
<tr>
<td>2</td>
<td>Kona Medium</td>
<td>18-001</td>
<td>18.45</td>
</tr>
<tr>
<td>3</td>
<td>Kona Dark</td>
<td>18-002</td>
<td>18.45</td>
</tr>
<tr>
<td>4</td>
<td>Guatemalan Decaf</td>
<td>21-002</td>
<td>10.45</td>
</tr>
</tbody>
</table>
Getting a Row from the `ResultSet` Object

- A `ResultSet object`'s `next` method moves the cursor to the next row in the `ResultSet`

```java
result.next();
```
- moves to first row in a newly created `ResultSet`
- moves to the next row each time it is called

<table>
<thead>
<tr>
<th>Cursor → Row 1</th>
<th>Sumatra Organic Dark</th>
<th>17-004</th>
<th>11.95</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 2</td>
<td>Kona Medium</td>
<td>18-001</td>
<td>18.45</td>
</tr>
<tr>
<td>Row 3</td>
<td>Kona Dark</td>
<td>18-002</td>
<td>18.45</td>
</tr>
<tr>
<td>Row 4</td>
<td>Guatemalan Decaf</td>
<td>21-002</td>
<td>10.45</td>
</tr>
</tbody>
</table>
Getting a Row from the `ResultSet` Object

- A `ResultSet` object’s `next` method returns a Boolean value
  - `true` if successfully moved to the next row
  - `false` if there are no more rows
- A `while` loop can be used to move through all the rows of a newly created `ResultSet`

```java
while (result.next())
{
    // Process the current row.
}
```
Getting Columns in a ResultSet Object

- You use one of the ResultSet object’s “get” methods to retrieve the contents of a specific column in the current row.
- Can pass an argument for either the column number or the column name

```java
System.out.println(result.getString(1));
System.out.println(result.getString(1));
System.out.println(result.getString(1));
System.out.println(result.getString("Description"));
System.out.println(result.getString("ProdNum"));
System.out.println(result.getDouble("Price"));
```

Examples:  
- ShowCoffeeDescriptions.java
- ShowDescriptionsAndPrices.java
## Getting Columns in a ResultSet Object

<table>
<thead>
<tr>
<th>ResultSet Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>double getDouble(int colNumber)</code></td>
<td></td>
</tr>
<tr>
<td><code>double getDouble(String colName)</code></td>
<td>Returns the double that is stored in the column specified by <code>colNumber</code> or <code>colName</code>. The column must hold data that is compatible with the double data type in Java. If an error occurs, the method throws an SQLException.</td>
</tr>
<tr>
<td><code>int getInt(int colNumber)</code></td>
<td></td>
</tr>
<tr>
<td><code>int getInt(String colName)</code></td>
<td>Returns the int that is stored in the column specified by <code>colNumber</code> or <code>colName</code>. The column must hold data that is compatible with the int data type in Java. If an error occurs, the method throws an SQLException.</td>
</tr>
<tr>
<td><code>String getString(int colNumber)</code></td>
<td></td>
</tr>
<tr>
<td><code>String getString(String colName)</code></td>
<td>Returns the string that is stored in the column specified by <code>colNumber</code> or <code>colName</code>. The column must hold data that is compatible with the String type in Java. If an error occurs, the method throws an SQLException.</td>
</tr>
</tbody>
</table>
Specifying Search Criteria with the \textit{WHERE} clause

- The \textit{WHERE} clause can be used with the \texttt{SELECT} statement to specify a search criteria

\begin{verbatim}
SELECT Columns FROM Table WHERE Criteria
\end{verbatim}

- \texttt{Criteria} is a conditional expression

- Example:

\begin{verbatim}
SELECT * FROM Coffee WHERE Price > 12.00
\end{verbatim}

- Only the rows that meet the search criteria are returned in the result set

- A \textit{result set} is an object that contains the results of an SQL statement
SQL Relational Operators

- Standard SQL supports the following relational operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

- 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:**
  
  ```sql
  SELECT * FROM Coffee WHERE Description = 'French Roast Dark'
  ```
  
  - In SQL, strings are enclosed in single quotes

- **Warning!**
  
  ```sql
  SELECT * FROM Coffee WHERE Description = 'french roast dark'
  ```
  
  - String comparisons in SQL are case sensitive

- **Example 2:**
  
  ```sql
  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:**
  
  ```sql
  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:**
  ```sql
  SELECT * FROM Coffee WHERE Description LIKE '%%Decaf%%'
  ```
  - The `%` symbol is used as a wildcard for multiple characters
- **Example 2:**
  ```sql
  SELECT * FROM Coffee WHERE ProdNum LIKE '2_00_'
  ```
  - The underscore (`_`) is a used as a wildcard for a single character
- **Example 3:**
  ```sql
  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:**

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

  - The **AND** operator requires that both search criteria be true.

- **Example 2:**

  ```sql
  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:
  ```sql
  SELECT * FROM Coffee ORDER BY Price
  ```
  – Sorted in ascending order (ASC) by default
• Example 2:
  ```sql
  SELECT * FROM Coffee
  WHERE Price > 9.95 ORDER BY Price DESC
  ```
  – Use the `DESC` operator to sort results in descending order
Mathematical Functions

Example: CoffeeMath.java

- **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
  
  ```sql
  INSERT INTO TableName VALUES (Value1, Value2, ...)
  ```
  
  - *TableName* is the name of the database table
  - *Value1, Value2, ...* is a list of column values

- Example:
  ```sql
  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

Example: CoffeeInserter.java
Updating an Existing Row

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

  $$\text{UPDATE } \text{Table}$$
  $$\text{SET } \text{Column} = \text{Value}$$
  $$\text{WHERE } \text{Criteria}$$

  - **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:

  $$\text{UPDATE Coffee}$$
  $$\text{SET Price} = 9.95$$
  $$\text{WHERE Description} = '\text{Galapagos Organic Medium'}$$
Updating More Than One Row

- It is possible to update more than one row
- Example:
  
  ```sql
  UPDATE Coffee
  SET Price = 12.95
  WHERE ProdNum LIKE '21%'
  ```
  
  - Updates the price of all rows where the product number begins with 21
- Warning!
  
  ```sql
  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:

  ```java
  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

Example: [CoffeePriceUpdater.java](#)
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:

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

- rows indicates the number of rows that were deleted

**Example:** CoffeeDeleter.java
Creating Tables with the `CREATE TABLE` Statement

- In SQL, the `CREATE TABLE` statement adds a new table to the database
  
  ```sql
  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:
  
  ```sql
  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
• Example: [CreateCustomerTable.java](#)

```
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 \texttt{DROP \ TABLE} Statement

- In SQL, the \texttt{DROP \ TABLE} statement deletes an existing table from the database

\begin{center}
\texttt{DROP \ TABLE \ TableName}
\end{center}

- \texttt{TableName} is the name of the table you wish to delete

- Example:

\begin{center}
\texttt{DROP \ TABLE \ Customer}
\end{center}

- Deletes the \texttt{Customer} table from the \texttt{CoffeeDB} database
- Useful if you make a mistake creating a table
- Simply delete the table and recreate
Creating a New Database with Java DB

• The `create=true` attribute creates a new database when appended to the database URL
  
  "jdbc:derby:EntertainmentDB;create=true"

  – Creates an empty database named `EntertainmentDB`
  – The `CREATE TABLE` statement can be used to create tables
  – Java DB creates a folder with the name of the database on your system
  – Delete the database folder to delete the database

• Example: [BuildEntertainmentDB.java](BuildEntertainmentDB.java)
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 of the Connection object’s createStatement method:

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

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

- Example:

  ```java
  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**(\textit{rows})
  - Moves the cursor the number specified by the \textit{rows} argument relative to the current row
    - A positive \textit{rows} value will move the cursor forward
    - A negative \textit{rows} value will move the cursor backward

- **absolute**(\textit{rows})
  - Moves the cursor to the row number specified by the \textit{rows} argument
    - A \textit{rows} value of 1 will move the cursor to the first row
    - A \textit{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:**

  ```java
  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.
- Example: `MetaDataDemo.java`

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

- Creates a `ResultSetMetaData` object reference variable named `meta`
# A Few `ResultSetMetaData` Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int getColumnCount()</code></td>
<td>Returns the number of columns in the result set.</td>
</tr>
<tr>
<td><code>String getColumnName(int col)</code></td>
<td>Returns the name of the column specified by the integer <code>col</code>. The first column is column 1.</td>
</tr>
<tr>
<td><code>String getColumnTypeName(int col)</code></td>
<td>Returns the name of the data type of the column specified by the integer <code>col</code>. The first column is column 1. The data type name returned is the database-specific SQL data type.</td>
</tr>
<tr>
<td><code>int getColumnDisplaySize(int col)</code></td>
<td>Returns the display width, in characters, of the column specified by the integer <code>col</code>. The first column is column 1.</td>
</tr>
<tr>
<td><code>String getTableName(int col)</code></td>
<td>Returns the name of the table associated with the column specified by the integer <code>col</code>. The first column is column 1.</td>
</tr>
</tbody>
</table>
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:

```java
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

This window appears next

It displays the results in a JTable component

- Example: TableFormatter.java, CoffeeDBQuery.java, CoffeeDBViewer.java
Relational Data

• A foreign key is a column in one table that references a primary key in another table
  – Creates a relationship between the tables

• Example:

  UnpaidOrder table:

  CustomerNumber  CHAR(10)  Foreign Key
  ProdNum         CHAR(10)  Foreign Key
  OrderDate       CHAR(10)
  Quantity        DOUBLE
  Cost            DOUBLE

  – The CustomerNumber column references the Customer table
  – The ProdNum column references the Coffee table
  – This creates a relationship between the tables of the CoffeeDB database
Creating the **UnpaidOrder Table**

- The following SQL statement creates the `UnpaidOrder` table in the `CoffeeDB` database:

  ```sql
  CREATE TABLE UnpaidOrder
  ( CustomerNumber CHAR(10) NOT NULL
    REFERENCES Customer(CustomerNumber),
    ProdNum CHAR(10) NOT NULL
    REFERENCES Coffee(ProdNum),
    OrderDate CHAR(10),
    Quantity DOUBLE,
    Cost DOUBLE )
  ```

- The `REFERENCES` qualifier ensures *referential integrity* between tables
  - The `CustomerNumber` in the `UnpaidOrder` table must contain a valid `CustomerNumber` from the `Customer` table
  - The `ProdNum` in the `UnpaidOrder` table must contain a valid `ProdNum` from the `Coffee` table
Entity Relationship Diagrams

- An entity relationship diagram shows the relationships between tables.

- Primary keys are denoted with (PK).
- Lines drawn between tables show how they are related:
  - The ends of each line show either a 1 or an infinity symbol (∞).
    - The infinity symbol means *many* and number 1 means *one*.
  - A *one to many relationship* means that for each row in table A there can be many rows in table B that reference it.
  - A *many to one relationship* means that many rows in table A can reference a single row in table B.
CoffeeDB Relationships Left to Right

- One to many relationship between **Customer** and **UnpaidOrder**
  - One row in the **Customer** table may be referenced by many rows in the **UnpaidOrder** table

- Many to one relationship between the **UnpaidOrder** and **Coffee** tables
  - Many rows in the **UnpaidOrder** table may reference a single row in the **Coffee** table.
CoffeeDB Relationships Right to Left

- One to many relationship between Coffee and UnpaidOrder
  - One row in the Coffee table may be referenced by many rows in the UnpaidOrder table
- Many to one relationship between UnpaidOrder and Customer
  - Many rows in the UnpaidOrder table may reference a single row in the Customer table.
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:

  
  $\text{TableName}.\text{Column}\text{Name}$

- 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.
An Order Entry System

- The *Place Order* application uses a relational database (*CoffeeDB*)
- Requires the *Coffee*, *Customer*, and *UnpaidOrder* tables

Example: [CoffeeDBManager.java](#), [CustomerPanel.java](#), [CoffeePanel.java](#), [QtyDatePanel.java](#), [PlaceOrder.java](#)
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`
    
    ```java
    conn.setAutoCommit(false);
    ```

• To execute a commit command
  – Call the Connection class's `commit` method
    
    ```java
    conn.commit();
    ```

• To execute a rollback command
  – Call the Connection class's `rollback` method
    
    ```java
    conn.rollback();
    ```
```
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();
}
```
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