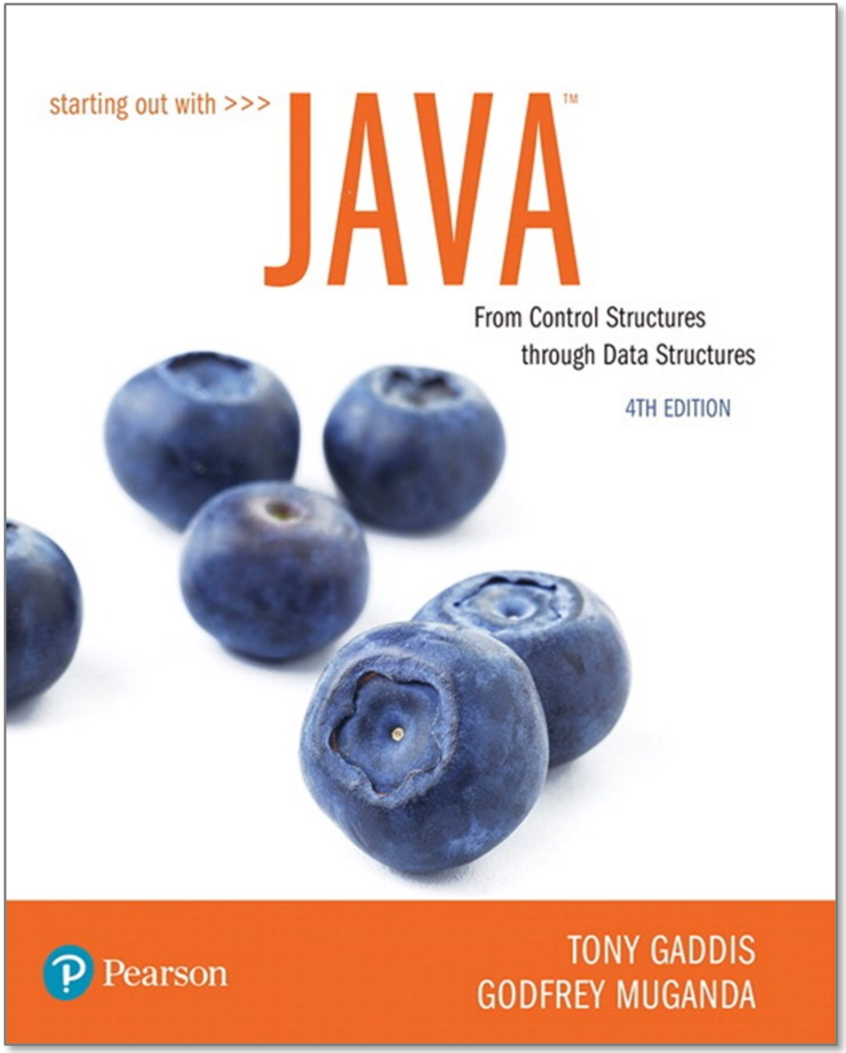
STARTING OUT WITH JAVA™

4th Edition



Chapter 22

Databases

Pearson

Chapter Topics (1 of 2)

Chapter 22 discusses the following main topics:

- Introduction to Database Management Systems
- Tables, Rows, and Columns
- Introduction to the SQL `SELECT` Statement
- Inserting Rows
- Updating and Deleting Existing Rows
- Creating and Deleting Tables
- Creating a New Database with JDBC

Chapter Topics (2 of 2)

- Scrollable Result Sets
- Result Set Metadata
- Relational Data
- Advanced Topics

Introduction to Database Management Systems (1 of 2)

- 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

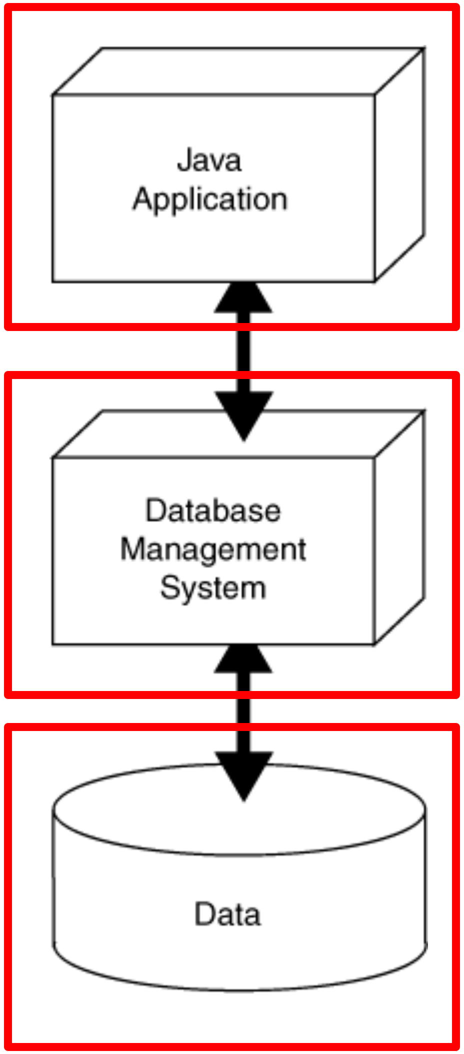
Introduction to Database Management Systems (2 of 2)

- 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

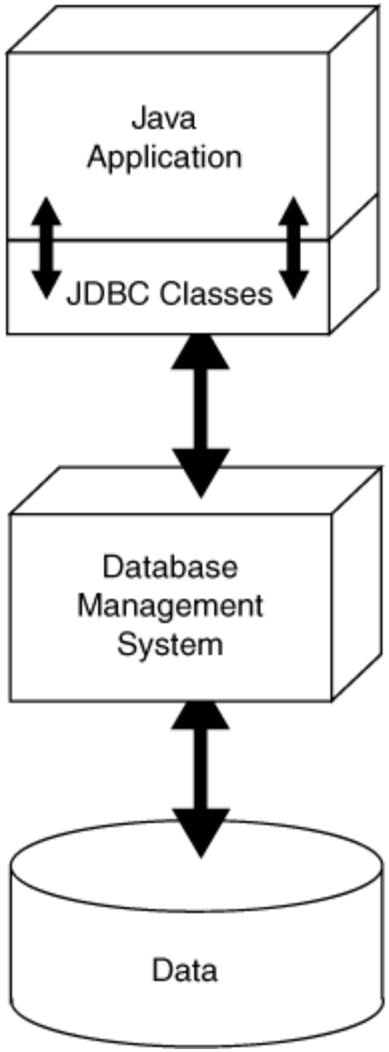


The Application displays the result to the user

The DBMS sends the result back to the Application

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

❑ Java DB is the Oracle release of the Apache Software Foundation's (ASF) open-source relational database project, Derby.

❑ Java DB (Derby) was included in JDK from Java 6

❑ From Java 9, it is no longer distributed with JDK

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

Pearson

Getting a Database Connection (1 of 3)

- The static `DriverManager.getConnection` method is used to get a connection to the database
  - General format of the simplest version:

    ```
    DriverManager.getConnection(DatabaseURL);
    ```

  - General format if a user name and a password are required:

    ```
    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 (2 of 3)

- *`DatabaseURL`* is a string known as a *database URL*
  - URL stands for uniform resource locator
- A simple database URL has the following general format:

  `protocol:subprotocol: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:

  `jdbc:derby:CoffeeDB`

Getting a Database Connection (3 of 3)

- 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

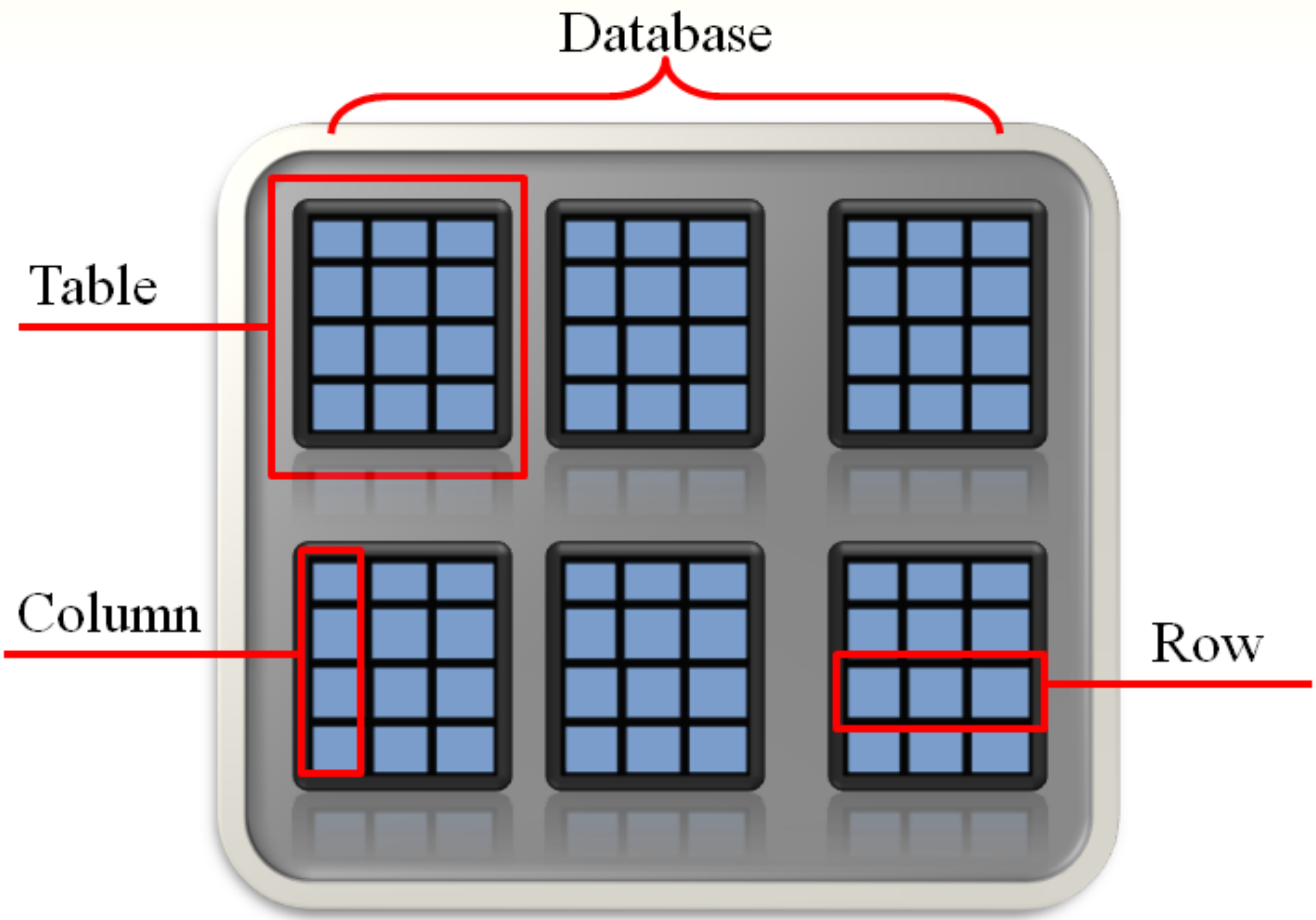
```
Final String DB_URL = "jdbc:derby:CoffeeDB";
Connection conn = DriverManager.getConnection(DB_URL);
```

- Example: TestConnection.java

Tables, Rows, and Columns

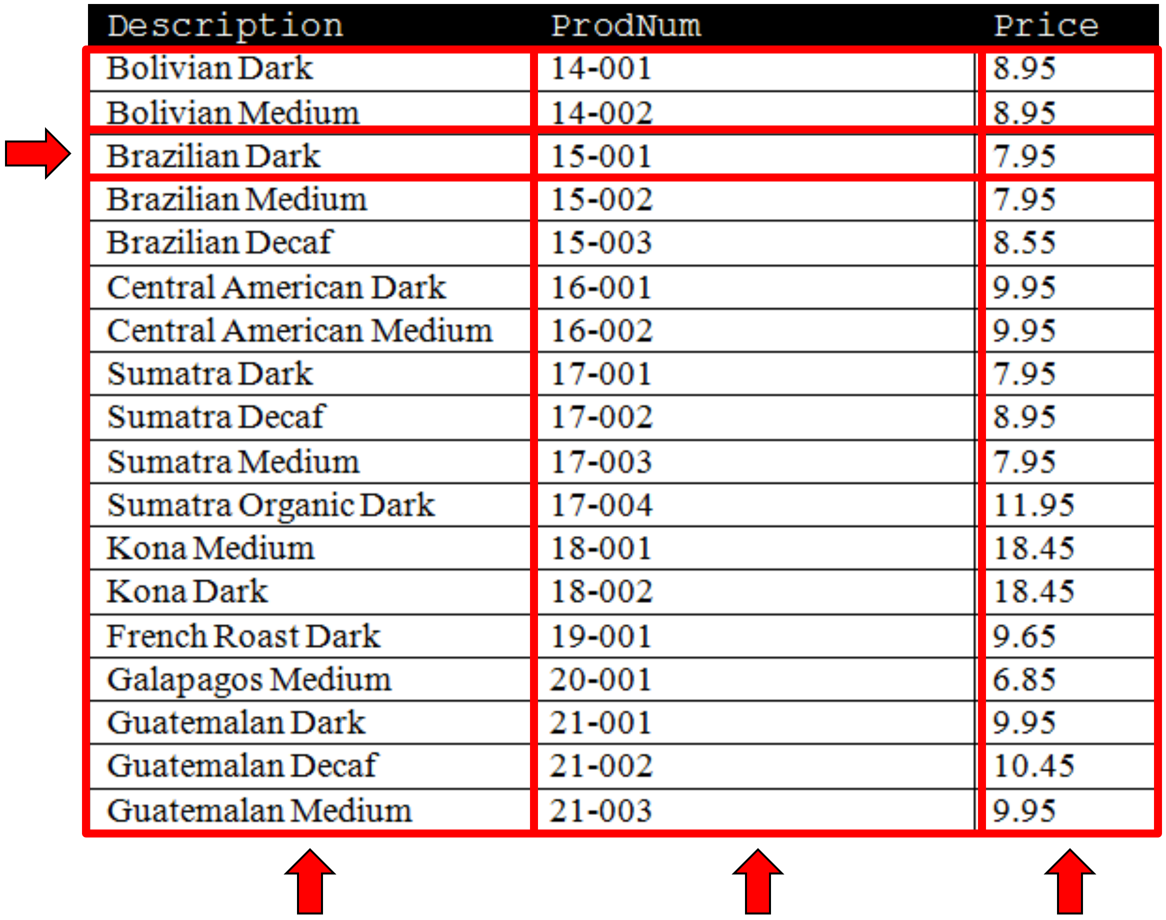
- A database management system stores data in a database
- A *database* is organized into one or more tables
- Each *table* holds a collection of related data, organized into rows and columns
- A *row* is a complete set of information about a single item, divided into columns
- Each *column* is an individual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

Database Organization



Pearson

Parts of the Coffee Database Table



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 Dark	16-001	9.95
Central American Medium	16-002	9.95
Sumatra Dark	17-001	7.95
Sumatra Decaf	17-002	8.95
Sumatra Medium	17-003	7.95
Sumatra Organic Dark	17-004	11.95
Kona Medium	18-001	18.45
Kona Dark	18-002	18.45
French Roast Dark	19-001	9.65
Galapagos Medium	20-001	6.85
Guatemalan Dark	21-001	9.95
Guatemalan Decaf	21-002	10.45
Guatemalan Medium	21-003	9.95

Each row contains data for a single item.

Description Column

ProdNum Column

Price Column

Column Data Types

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

SQL Data Type	Description	Corresponding Java Data Type
INTEGER or INT	An integer number	int
CHARACTER(*n*) or CHAR(*n*)	A fixed-length string with a length of *n* characters	string
VARCHAR(*n*)	A variable-length string with a maximum length of *n* characters	String
REAL	A single-precision floating-point number	float
DOUBLE	A double-precision floating-point number	double
DECIMAL(*t*, *d*)	A decimal value with *t* total digits and *d* digits appearing after the decimal point	java.math.BigDecimal
DATE	A date	java.sql.Date

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

```
SELECT Columns FROM Table
```

  - *`Columns`* is one or more column names
  - *`Table`* is a table name
- Example 1:

```
SELECT Description FROM Coffee
```

- Example 2:

```
SELECT Description, Price FROM Coffee
```

  - Multiple column names are separated with a comma
- Example 3:

```
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:*

```
SELECT * FROM Coffee

SELECT
   *
FROM
   Coffee

select * from coffee
```

Passing an SQL Statement to the DBMS

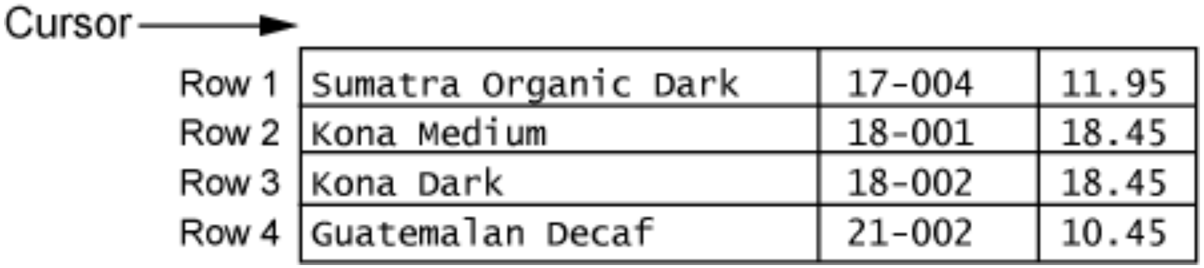
- Once you have established a connection, you must get a reference to a `Statement` object before you can issue SQL statements to the DBMS
  - A `Statement` object has an `executeQuery` method that returns a reference to a `ResultSet` object
  - A `ResultSet` object contains the results of the query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

```
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 (1 of 3)

- 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

Initially the cursor is positioned just before the first row in the `ResultSet`.



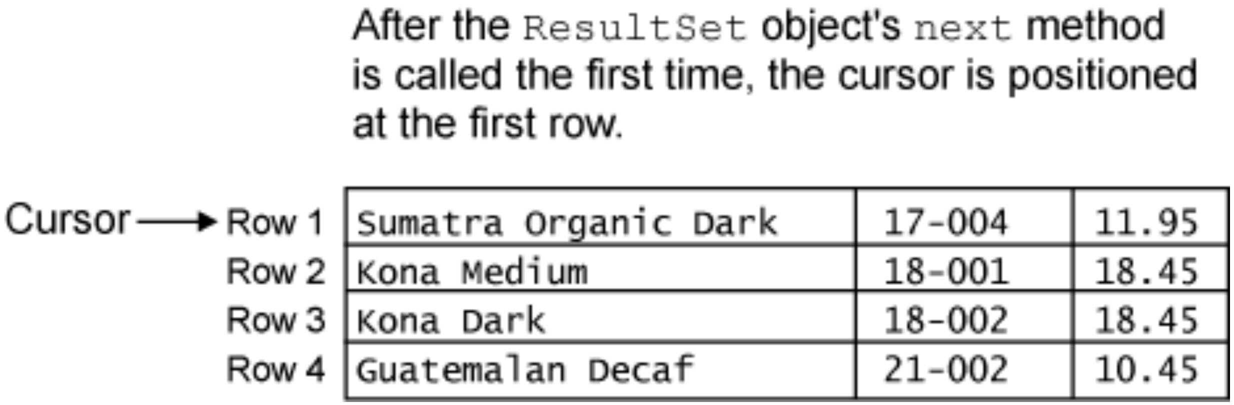
Row 1	Sumatra Organic Dark	17-004	11.95
Row 2	Kona Medium	18-001	18.45
Row 3	Kona Dark	18-002	18.45
Row 4	Guatemalan Decaf	21-002	10.45

Getting a Row from the `ResultSet` Object (2 of 3)

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

```
result.next();
```

- moves to first row in a newly created `ResultSet`
- moves to the next row each time it is called



After the `ResultSet` object's `next` method is called the first time, the cursor is positioned at the first row.

Row 1	Sumatra Organic Dark	17-004	11.95
Row 2	Kona Medium	18-001	18.45
Row 3	Kona Dark	18-002	18.45
Row 4	Guatemalan Decaf	21-002	10.45

Getting a Row from the `ResultSet` Object (3 of 3)

- 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`

```
while (result.next())
{
    // Process the current row.
}
```

Getting Columns in a `ResultSet` Object (1 of 2)

- 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

```
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 (2 of 2)

`ResultSet` Method	Description

`double getDouble(int colNumber)`
`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)`
`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)`
`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`.

Pearson

Specifying Search Criteria with the `WHERE` clause

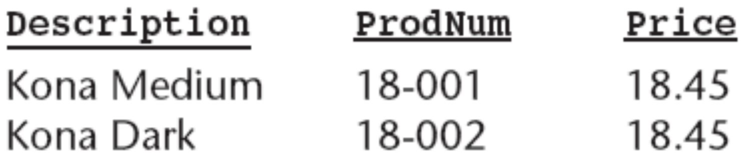
- The `WHERE` clause can be used with the `SELECT` statement to specify a search criteria

```
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

Pearson

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

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
```

Pearson

Inserting Rows (1 of 2)

- 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 (2 of 2)

- 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

```
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

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:

```
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 (1 of 2)

- 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 (2 of 2)

- 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 `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

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

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

Pearson

`ResultSet` Navigation Methods (1 of 2)

- `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 (2 of 2)

- `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 (1 of 2)

- 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 (2 of 2)

- `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

```
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.

Pearson

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

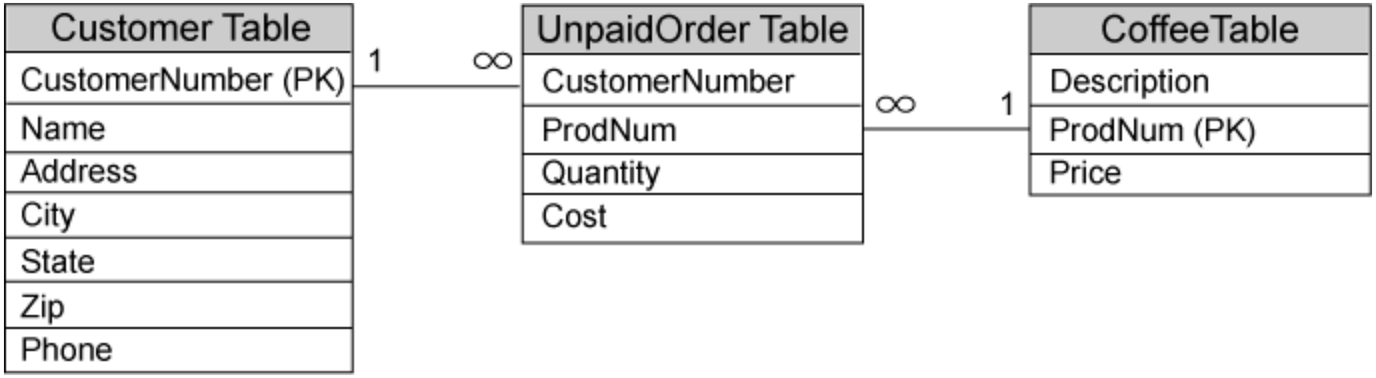
- The following SQL statement creates the UnpaidOrder table in the CoffeeDB database:

```
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



Customer Table
CustomerNumber (PK)
Name
Address
City
State
Zip
Phone

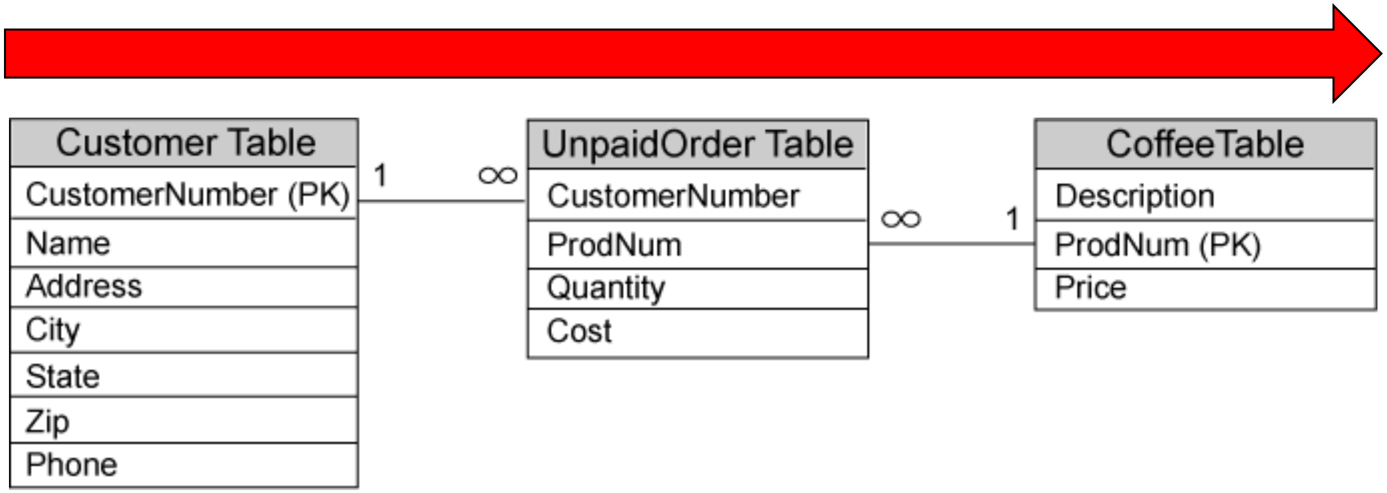
UnpaidOrder Table
CustomerNumber
ProdNum
Quantity
Cost

CoffeeTable
Description
ProdNum (PK)
Price

- 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*.

Pearson

`CoffeeDB` Relationships Left to Right



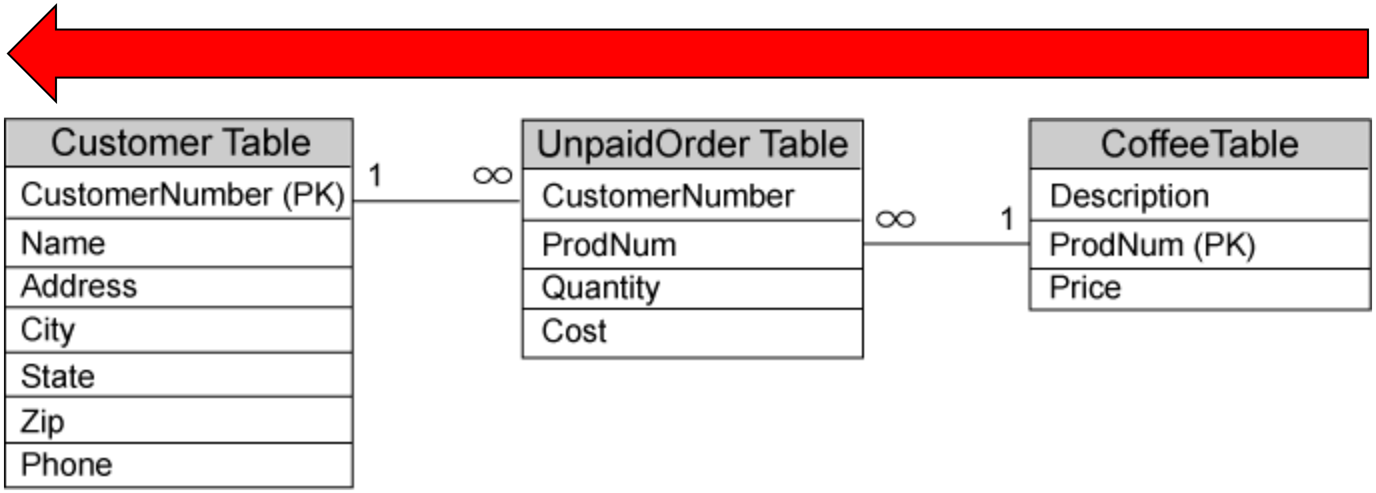
Customer Table
CustomerNumber (PK)
Name
Address
City
State
Zip
Phone

UnpaidOrder Table
CustomerNumber
ProdNum
Quantity
Cost

CoffeeTable
Description
ProdNum (PK)
Price

- 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



Customer Table
CustomerNumber (PK)
Name
Address
City
State
Zip
Phone

UnpaidOrder Table
CustomerNumber
ProdNum
Quantity
Cost

CoffeeTable
Description
ProdNum (PK)
Price

- 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:

  `TableName.ColumnName`

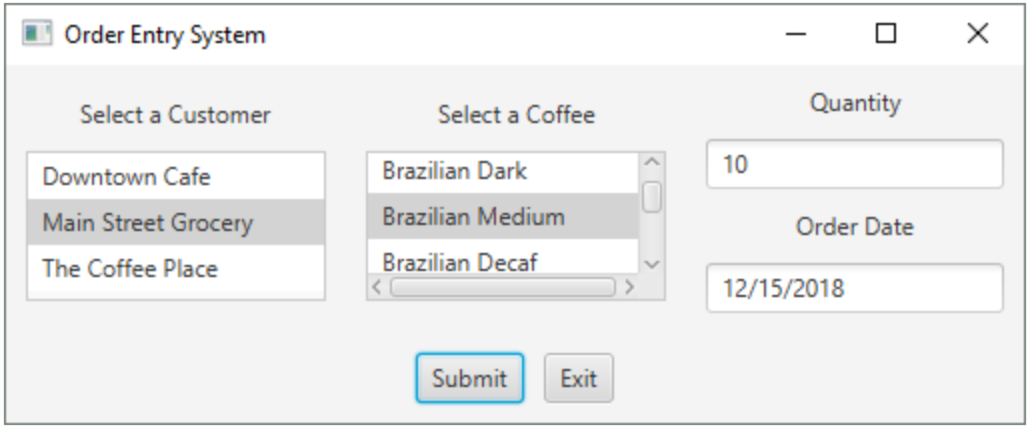
- Example:

```
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

An Order Entry System

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



Select a Customer

Downtown Cafe
Main Street Grocery
The Coffee Place

Select a Coffee

Brazilian Dark
Brazilian Medium
Brazilian Decaf

Quantity

10

Order Date

12/15/2018

- Example: CoffeeDBManager.java, OrderEntrySystem.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`

```
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 `rollback` method

```
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

Copyright