Database Applications (1)

Han-fen Hu

Tony Gaddis (2019) Starting Out with Java: From Control Structures through Data Structures, 4th Edition

UNLV

Outline

- Introduction to Database Management Systems (DBMS)
- Java and DBMS
- Getting a Database Connection
- SQL Statements
- Statements and Prepared Statements
- Inserting, Updating and Deleting Rows

UNLV

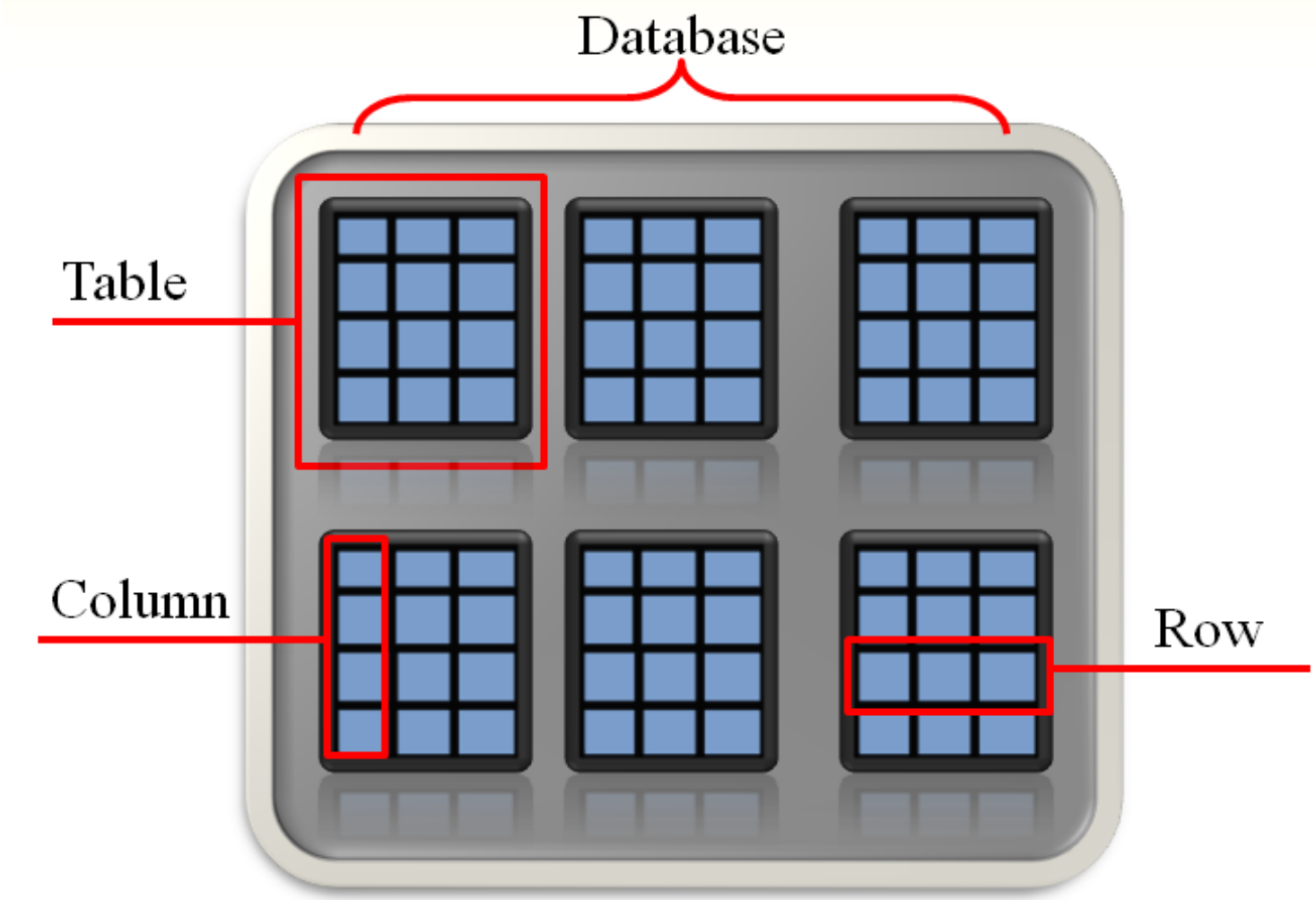
Database Management Systems (1)

- 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

Database Management Systems (2)

- A database management system stores data in a database
- A relational 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

Concept of Database



UNLV

Example: Coffee table

Each row contains data for a single item.

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

`Description` Column

`ProdNum` Column

`Price` Column

UNLV

Database Management Systems (3)

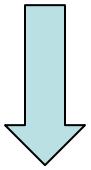
- A primary key is a field (or a set of fields) that contain(s) a unique identifier for each record
  - A primary key can be a single column
  - Or multiple columns together as the key
- Primary key values must be unique
  - Student ID
  - Semester+Year+ClassNumber

Database Management Systems (4)

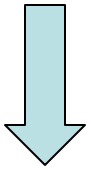
- A *database management system (DBMS)* is software that is specifically designed to work with large amounts of data in an efficient and organized manner
  - Data is stored using the database management system
  - Applications written in Java or other languages communicate with the DBMS rather than manipulate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

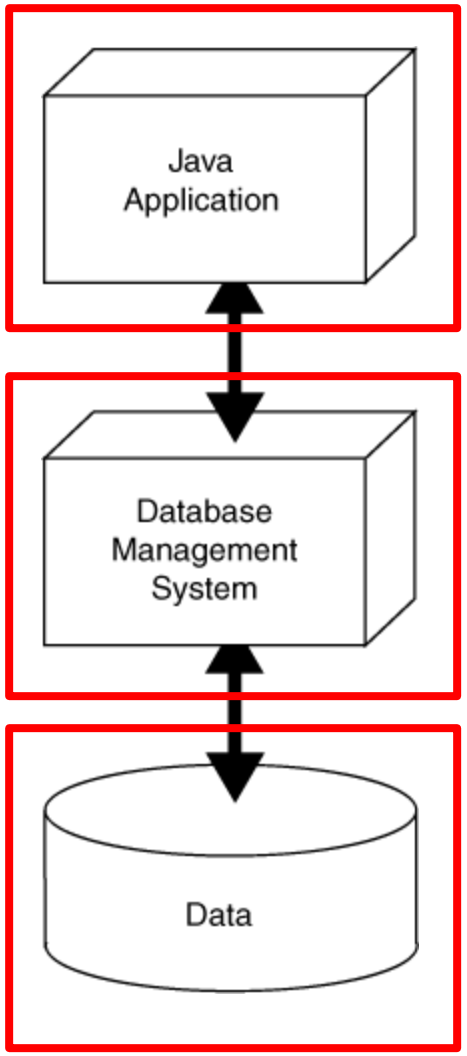
Java Applications Interact with a DBMS

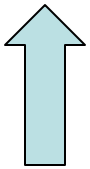
The Application sends a command to the DBMS



The DBMS executes the command on the Data







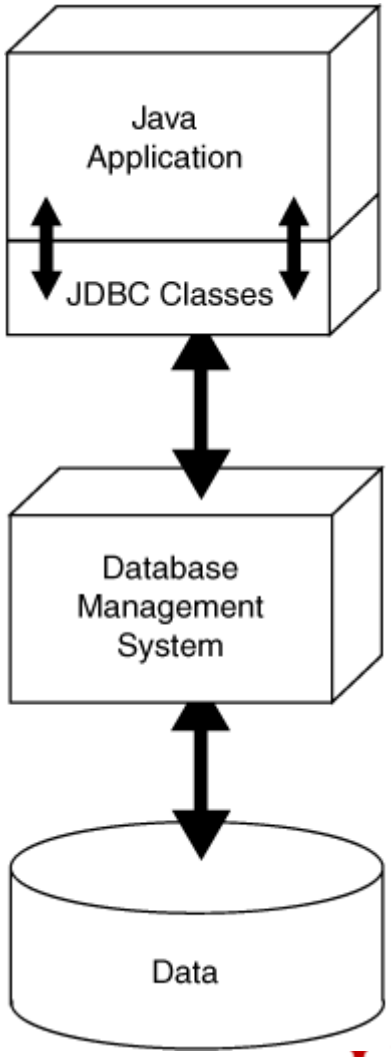
The DBMS sends the result back to the Application

The Application displays the result to the user

UNLV

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



UNLV

JDBC (1)

- To use JDBC to work with a database you will need a DBMS
  - Java DB
  - Oracle
  - Microsoft SQL Server
  - MySQL

JDBC (2)

- 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

UNLV

Getting a Database Connection (1)

- 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)

- ***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 **mysql** when using MySQL
  - **databaseName** is the name of the database, include the URL and port number
- Using MySQL, the URL for the **CoffeeDB** database is:

```
jdbc:mysql://localhost:3306/coffeeShopData
```

Getting a Database Connection (3)

- Using Microsoft SQL Seiver, the URL for the **`CoffeeDB`** database is:

```
jdbc:sqlserver://localhost:1433;databaseName=
CoffeeShopData
```

Getting a Database Connection (4)

- 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:mysql://localhost:3306/coffeeDB";
Connection conn = DriverManager.getConnection(DB_URL);
```

Lab (1)

- CreateDB.java
- TestConnection.java

SQL (1)

- SQL stands for structured query language
  - A standard language for working with database management systems
- Statements or queries are strings passed from the application to the DBMS using API method calls
  - Consists of several key words, used to construct statements known as queries
  - Serve as instructions for the DBMS to carry out operations on its data

SQL (2)

- 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

SQL (3)

- 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

Passing an SQL Statement to the DBMS

- 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,USER_NAME,PASSWORD);
Statement stmt = conn.createStatement();
String sql = "SELECT Description FROM Coffee";
ResultSet result = stmt.executeQuery(sql);
```

Getting a Row from the `ResultSet` Object (1)

- 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`.

Cursor →

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

UNLV

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

- 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.

Cursor →	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

UNLV

Getting a Row from the **`ResultSet`** Object (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

- You use one of the **`ResultSet`** object's "get" methods to retrieve the contents of a specific column in the current row.

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

Lab (2)

- TestConnection.java
  - We can add some SELECT statement to query data

Inserting Rows (1)

- 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)

- 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')
```

Inserting Rows (3)

- 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

Inserting Rows: 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

Inserting Rows (4)

- We can also use prepared statement for insertion
  - A parameterized query in which placeholders used for parameters and the parameter values supplied at execution time.

Inserting Rows: Example

```
String sqlStatement =
"Insert into Coffee (ProdNum, Description, Price) values (?,?,?)";

PreparedStatement prepStmt = conn.prepareStatement(sqlStatement);

prepStmt.setString(1, prodNum);
prepStmt.setString(2, description);
prepStmt.setDouble(3, price);

int row = prepStmt.executeUpdate();

// prodNum, description, and price are variables with values to be
inserted.
```

Lab (3)

- CoffeeInserter.java
  - The program would allow the user to input the description, product number, and price.
  - Then the program connects to the database, sets up the prepared statement object, and executes the SQL statement.

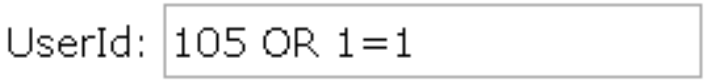
Statement vs. Prepared Statement

- Statement
  - Executing static, simple SQL statements
  - Not accepting input parameters
  - Prone to security issue
- PreparedStatement
  - Executing SQL statements dynamically, with conditions
  - Accepting input parameters
  - More secured

SQL Injection

- Based on 1=1 is Always True
- Example
  - If your code used to validate username/password looked like this

```
sqlStatement = "SELECT * FROM Users WHERE UserId = " +
userIdTextField.getText();
```



  - When the user enters the SQL statement will look like this

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```


UNLV

Use SQL Parameters for Protection

- Example

```
sqlStatement = "SELECT * FROM Users WHERE
UserId = ?";

PreparedStatement prepStmt =
conn.prepareStatement(sqlStatement);

prepStmt.setString(1, userIdTextField.getText());

prepStmt.executeQuery();
```

Lab (4)

- CustomerFinder.java
  - Use a prepared statement in a query.

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 Price >= 9.95
```

  - Updates the price of all rows where the current product price is no less than 9.95

- 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 (1)

- 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

Updating Rows with JDBC (2)

- Similarly, we can use a prepared statement to execute an update statement

```
String sqlStatement = "UPDATE Coffee SET Price = ? WHERE ProdNum = ?";

PreparedStatement prepStmt = conn.prepareStatement(sqlStatement);

prepStmt.setDouble(1, price);
prepStmt.setString(2, prodNum);

int rows = prepStmt.executeUpdate();
```

Lab (5)

- CoffeePriceUpdater.java
  - The program finds a specific product and update its price.
  - It first calls findProduct () to determine whether the record exists. If yes, ask the user to input a new price, and then class updatePrice() to update the price.

Deleting Rows (1)

- 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

Deleting Rows (2)

- 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 (1)

- 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

Deleting Rows with JDBC (1)

- With prepared statement

```
String sqlStatement = "DELETE FROM Coffee WHERE ProdNum = ?";

PreparedStatement prepStmt = conn.prepareStatement(sqlStatement);

prepStmt.setString(1, prodNum);


int rows = prepStmt.executeUpdate();
```

Lab (6)

- CoffeeDeletion.java
  - This program is very similar to CoffeePriceUpdater.java.
  - After calling findProduct() to determine whether the record exists, ask the user whether he/she wants to remove the record. If yes, call deleteCoffee() to remove the record.