Introduction to Java for C++ Programmers

JDBC

BY: Mahboob Ali

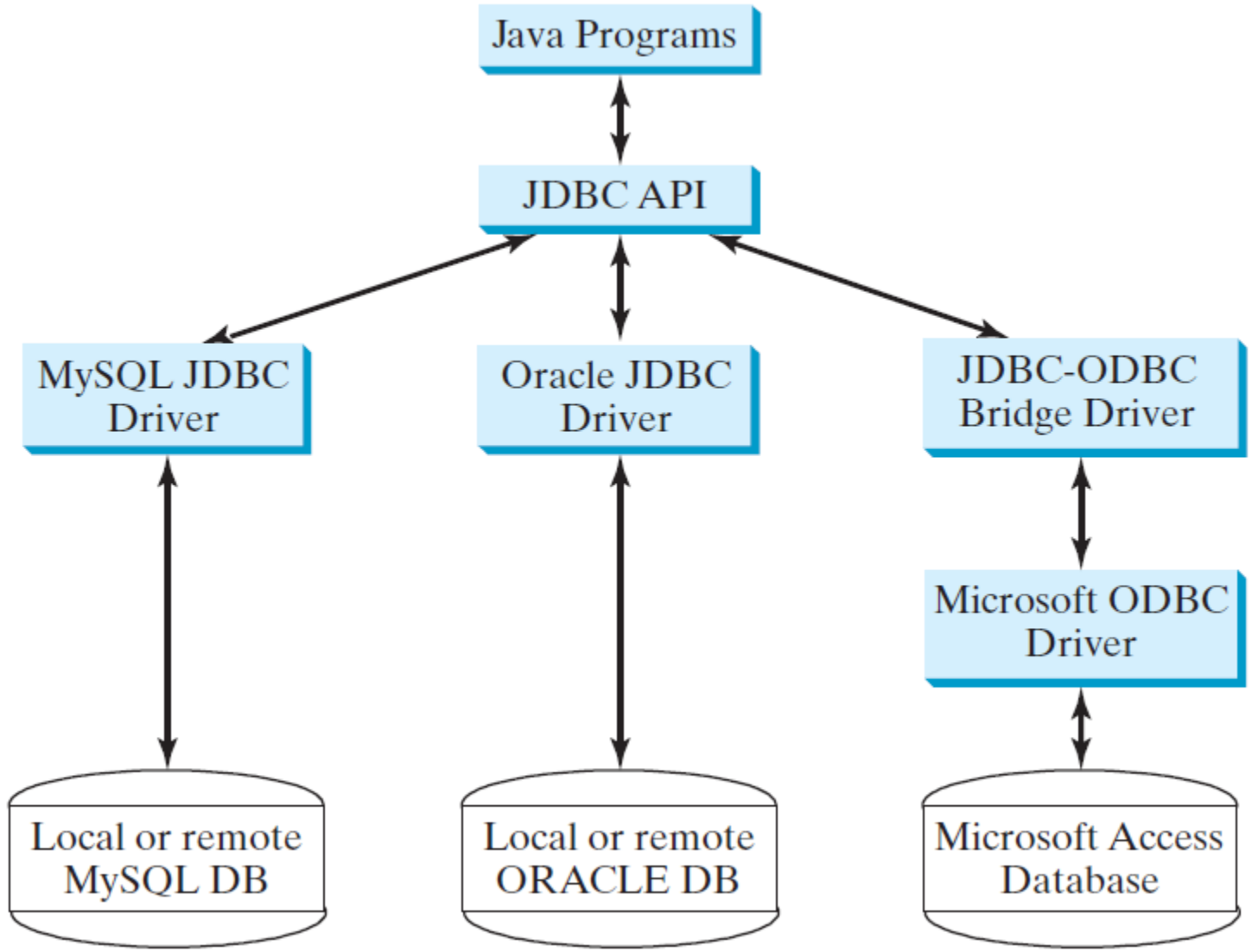
Why Java for Database Programming?

- First, Java is platform independent. You can develop platform-independent database applications using SQL and Java for any relational database systems.
- Second, the support for accessing database systems from Java is built into Java API, so you can create database applications using all Java code with a common interface.
- Third, Java is taught in almost every university either as the first programming language or as the second programming language.

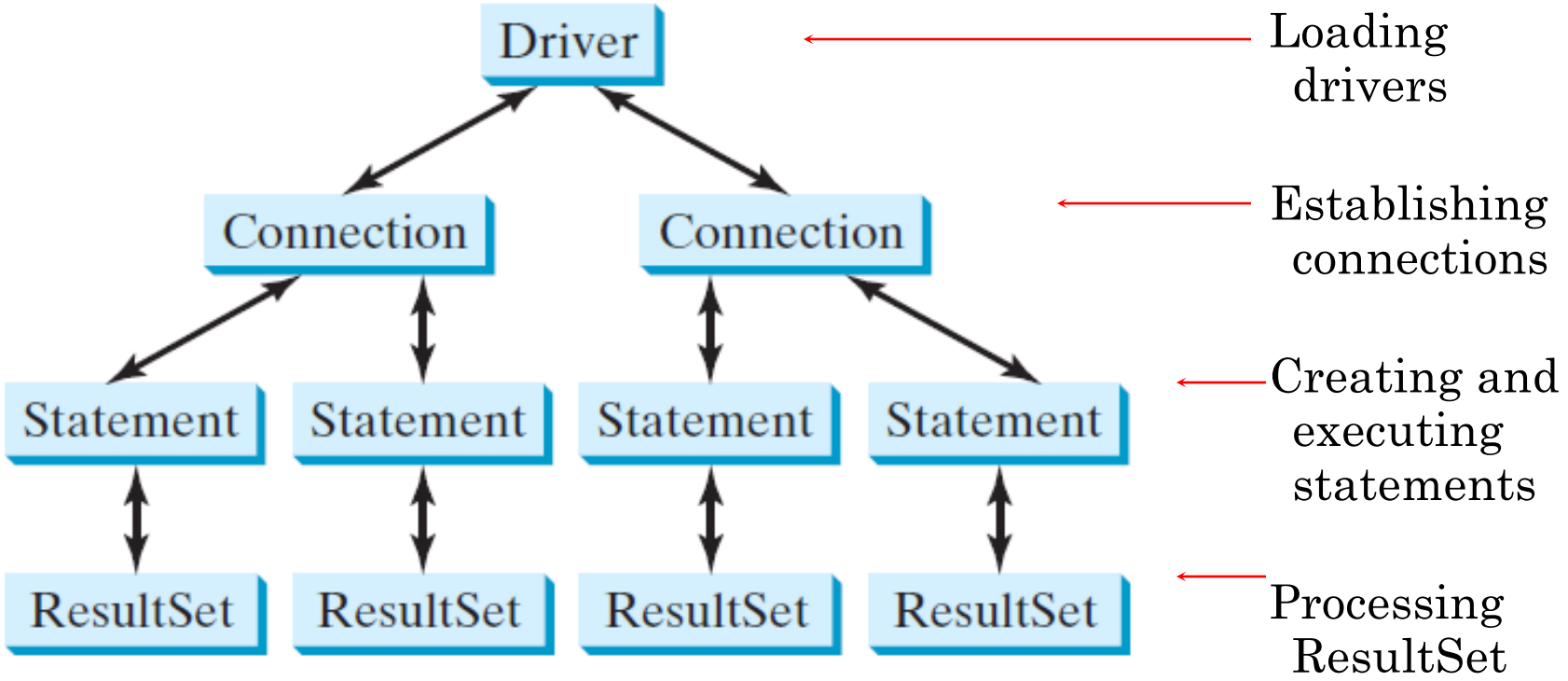
Database Applications Using Java

- GUI
- Client/Server
- Server-Side programming

The Architecture of JDBC



The JDBC Interfaces



- Driver is simply a java *library* containing classes that implement the JDBC API.

- All JDBC drivers have to implement the same interface, it's not difficult to change the data source.

- For example, while using an sqlite database if you want to switch to MySQL database, then just needed to install MySQL JDBC driver.

Downloading the JDBC for SQLite

- https://bitbucket.org/xerial/sqlite-jdbc/downloads/
- Click on the latest version which is “sqlite-jdbc-3.21.0.jar” or whatever latest version is available

- Lets also download and install SQLite browser for windows (GUI for SQLite)
- http://sqlitebrowser.org/

Choose appropriate windows version 32-bit or 64-bit.
Run the executable file and follow the instructions.
Once finish installation run the DB-browser.

JDBC Technology

Four steps required to design apps with JDBC

- Connect to the database
- Create a statement and execute the query
- Look at the result set
- Close connection

Creating Databases with JDBC in Java

- Create New project in Eclipse.

- Add the sqlite jdbc jar file that we downloaded earlier to the library.

Basic CRUD operations

```
import java.sql.Statement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class TestDBExample {
  public static void main(String[] args) {
    try {
    Connection conn =
      DriverManager.getConnection("jdbc:sqlite:E:\\databases\\testjava.db");
    Statement statement = conn.createStatement();
    statement.execute("CREATE TABLE contacts (name TEXT, phone INTEGER, email TEXT)");

    //closing resources manually
    statement.close();
    conn.close();

    }catch(SQLException e) {
    System.out.println("Something went wrong: "+e.getMessage());
    }
  }  }
```

```
import java.sql.Statement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;



public class TestDBExample {
  public static void main(String[] args) {
    try(Connection conn =
     DriverManager.getConnection("jdbc:sqlite:E:\\databases\\testjava.db");
    Statement statement = conn.createStatement()){


    statement.execute("CREATE TABLE contacts (name TEXT, phone INTEGER, email TEXT)");


    }catch(SQLException e) {
    System.out.println("Something went wrong: "+e.getMessage());
    }
  }

}
```

```
. . .
public class InsertAndUpdateExample {
  public static void main(String[] args) {
    try {
    Connection conn =
      DriverManager.getConnection("jdbc:sqlite:E:\\databases\\testjava.db");
    Statement statement = conn.createStatement();
    statement.execute("CREATE TABLE IF NOT EXISTS contacts "+
    " (name TEXT, phone INTEGER, email TEXT)");

    statement.execute("INSERT INTO contacts (name, phone, email)" +
              "VALUES ('Ali', 123456, 'ali@myemail.com')");
    statement.execute("INSERT INTO contacts (name, phone, email)" +
              "VALUES ('John', 789456, 'john@myemail.com')");
    statement.execute("INSERT INTO contacts (name, phone, email)" +
              "VALUES ('Roy', 753159, 'roy@myemail.com')");
    //closing resources manually
    statement.close();
    conn.close();
    }catch(SQLException e) {
    System.out.println("Something went wrong: "+e.getMessage());
    }
  }  }
```

```
public class InsertAndUpdateExample {
  public static void main(String[] args) {
    try {
    Connection conn =
     DriverManager.getConnection("jdbc:sqlite:E:\\databases\\testjava.db");
    Statement statement = conn.createStatement();
    statement.execute("CREATE TABLE IF NOT EXISTS contacts "+
    " (name TEXT, phone INTEGER, email TEXT)");

      statement.execute("UPDATE contacts SET phone=159357 WHERE
 name='Ali'");

      statement.execute("DELETE FROM contacts WHERE name='Ali'");


    statement.close();
    conn.close();
    }catch(SQLException e) {
    System.out.println("Something went wrong: "+e.getMessage());
    }
  } }
```

```
public class SelectStatementExample {
   public static void main(String[] args) {
      try {
      Connection conn = DriverManager.getConnection("jdbc:sqlite:E:\\databases\\testjava.db");
      Statement statement = conn.createStatement();
      statement.execute("CREATE TABLE IF NOT EXISTS contacts "+
      " (name TEXT, phone INTEGER, email TEXT)");
      statement.execute("SELECT * FROM contacts");
      ResultSet results = statement.getResultSet();
      while(results.next()) {
      System.out.println(results.getString("name") + " " +
      results.getInt("phone") + " " +
      results.getString("email"));
      }
      results.close();
      statement.close();
      conn.close();
      }catch(SQLException e) {
      System.out.println("Something went wrong: "+e.getMessage());
      }
   } }
```

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class UdatedCodeExample {
  public static final String DB_NAME = "testjava.db";
  public static final String CONNECTION_STRING = "jdbc:sqlite:E:\\databases\\" + DB_NAME;
  public static final String TABLE_CONTACTS = "contacts";
  public static final String COLUMN_NAME = "name";
  public static final String COLUMN_PHONE = "phone";
  public static final String COLUMN_EMAIL = "email";

  public static void main(String[] args) {

        try {

                Connection conn = DriverManager.getConnection(CONNECTION_STRING);
                Statement statement = conn.createStatement();

                statement.execute("DROP TABLE IF EXISTS " + TABLE_CONTACTS);

                statement.execute("CREATE TABLE IF NOT EXISTS "+TABLE_CONTACTS +
                                  " (" + COLUMN_NAME + " text, " +
                                         COLUMN_PHONE + " integer, " +
                                         COLUMN_EMAIL + " text" +
                                  ")" );
```

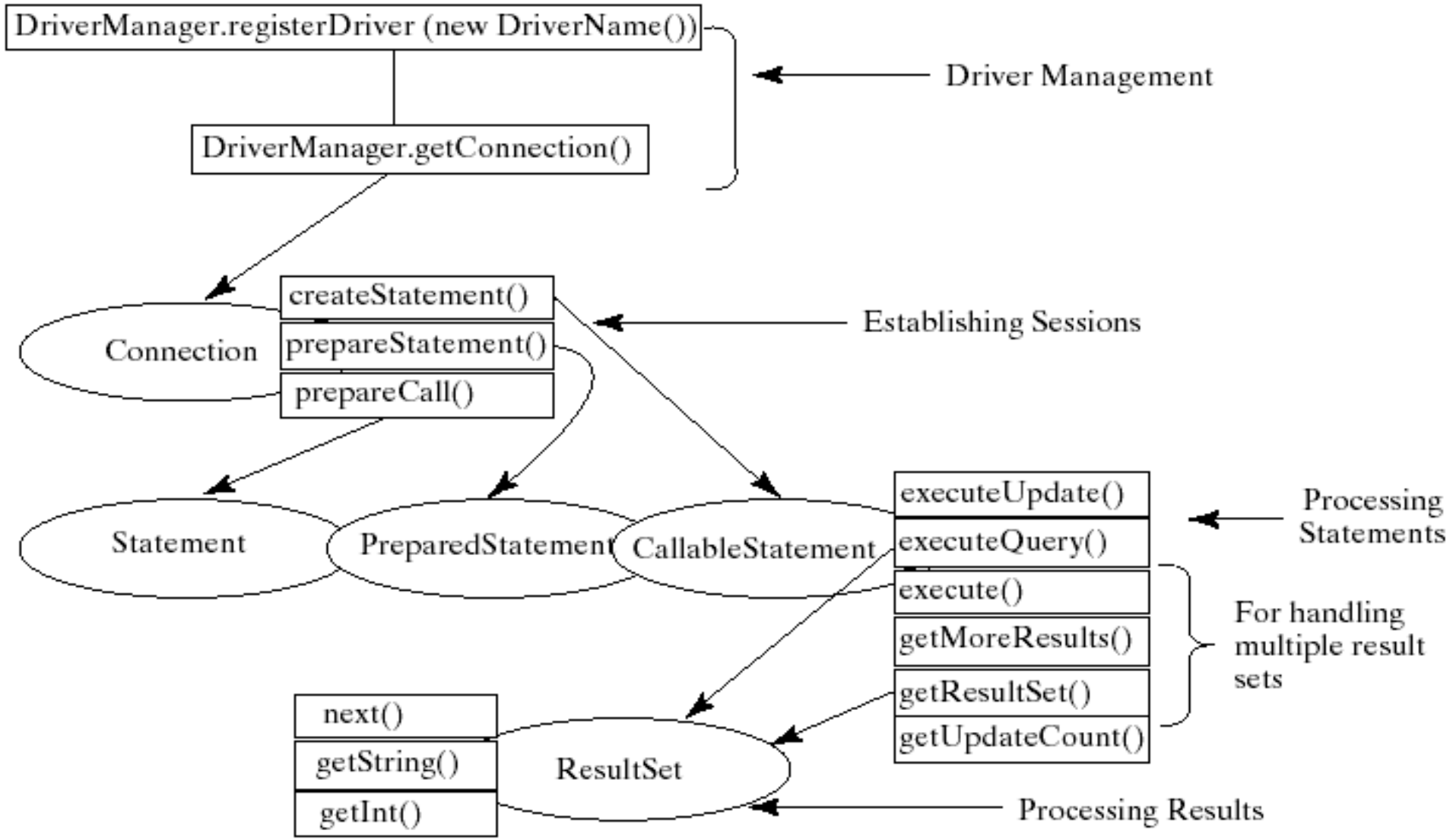
```
statement.execute("INSERT INTO " + TABLE_CONTACTS +
        " (" + COLUMN_NAME + ", " +
        COLUMN_PHONE + ", " +
        COLUMN_EMAIL +
        ")" +
        "VALUES('Ali', 159753,'ali@myemail.com')");
statement.execute("INSERT INTO " + TABLE_CONTACTS +
        "(" + COLUMN_NAME + ", " +
        COLUMN_PHONE + ", " +
        COLUMN_EMAIL +
        ")" +
        "VALUES('Jhon', 123456,'jhon@jhonemail.com')");
statement.execute("INSERT INTO " + TABLE_CONTACTS +
        "(" + COLUMN_NAME + ", " +
        COLUMN_PHONE + ", " +
        COLUMN_EMAIL +
        ")" +
        "VALUES('Roy', 7894562,'roy@royemail.com')");
statement.execute("INSERT INTO " + TABLE_CONTACTS +
        "(" + COLUMN_NAME + ", " +
        COLUMN_PHONE + ", " +
        COLUMN_EMAIL +
        ")" +
        "VALUES('Nick', 753654,'nick@nickemail.com')");
statement.execute("UPDATE " + TABLE_CONTACTS + " SET "+
        COLUMN_PHONE + "=321654" + " WHERE "+
        COLUMN_NAME + "='Ali'");
statement.execute("DELETE FROM " + TABLE_CONTACTS + " WHERE "+
            COLUMN_NAME + "='Nick'");
```

```
ResultSet results = statement.executeQuery("SELECT * FROM "+
      TABLE_CONTACTS);
while(results.next()) {
      System.out.println(results.getString(COLUMN_NAME) + " " +
                  results.getInt(COLUMN_PHONE) + " " +
                  results.getString(COLUMN_EMAIL));
            }
      results.close();

      //closing resources manually
      statement.close();
      conn.close();

}catch(SQLException e) {
      System.out.println("Something went wrong: "+e.getMessage());
      e.printStackTrace();
}
} }
```

Processing Statements

- Once a connection to a particular database is established, it can be used to
  - send SQL statements from your program to the database.
- JDBC provides the Statement,
  - PreparedStatement
  - CallableStatement interfaces
  - to facilitate sending statements to a database for execution and receiving execution results from the database.

Processing Statements Diagram



The execute, executeQuery, and executeUpdate Methods

- The methods for executing SQL statements are
  - execute,
  - executeQuery, and
  - executeUpdate
- Each of which accepts a string containing a SQL statement as an argument.
- This string is passed to the database for execution.
- The execute method should be used if the execution produces
  - multiple result sets,
  - multiple update counts, or
  - a combination of result sets and update counts.

The execute, executeQuery, and executeUpdate Methods, cont.

- The executeQuery method should be used if the execution produces a single result set, such as the SQL select statement.
- The executeUpdate method should be used if the statement results in a single update count or no update count, such as a SQL INSERT, DELETE, UPDATE, or DDL statement.

PreparedStatement

The PreparedStatement interface is designed to execute dynamic SQL statements and SQL-stored procedures with IN parameters. These SQL statements and stored procedures are precompiled for efficient use when repeatedly executed.

```
Statement pstmt = connection.prepareStatement

  ("insert into Student (firstName, mi, lastName) +

    values (?, ?, ?)");
```

Retrieving Database Metadata

- Database metadata is the information that describes database itself.

- JDBC provides the DatabaseMetaData interface for obtaining database wide information and the ResultSetMetaData interface for obtaining the information on the specific ResultSet.

DatabaseMetadata, cont.

- The DatabaseMetaData interface provides more than 100 methods for getting database metadata concerning the database as a whole.

- JDBC Metadata API can be used to retrieve the following information about the database:
  - Database users, tables, views, stored procedures
  - Database schema and catalog information
  - Table, view, column privileges
  - Information about primary key, foreign key of a table