JDBC (Java / SQL Programming)

CS 377: Database Systems

JDBC

- Acronym for Java Database Connection
- Provides capability to access a database server through a set of library functions
  - Set of library functions forms a standardized Application Program Interface (API)
  - Allows programmer to send SQL statements for execution and query retrieval
- Supported by most major database vendors

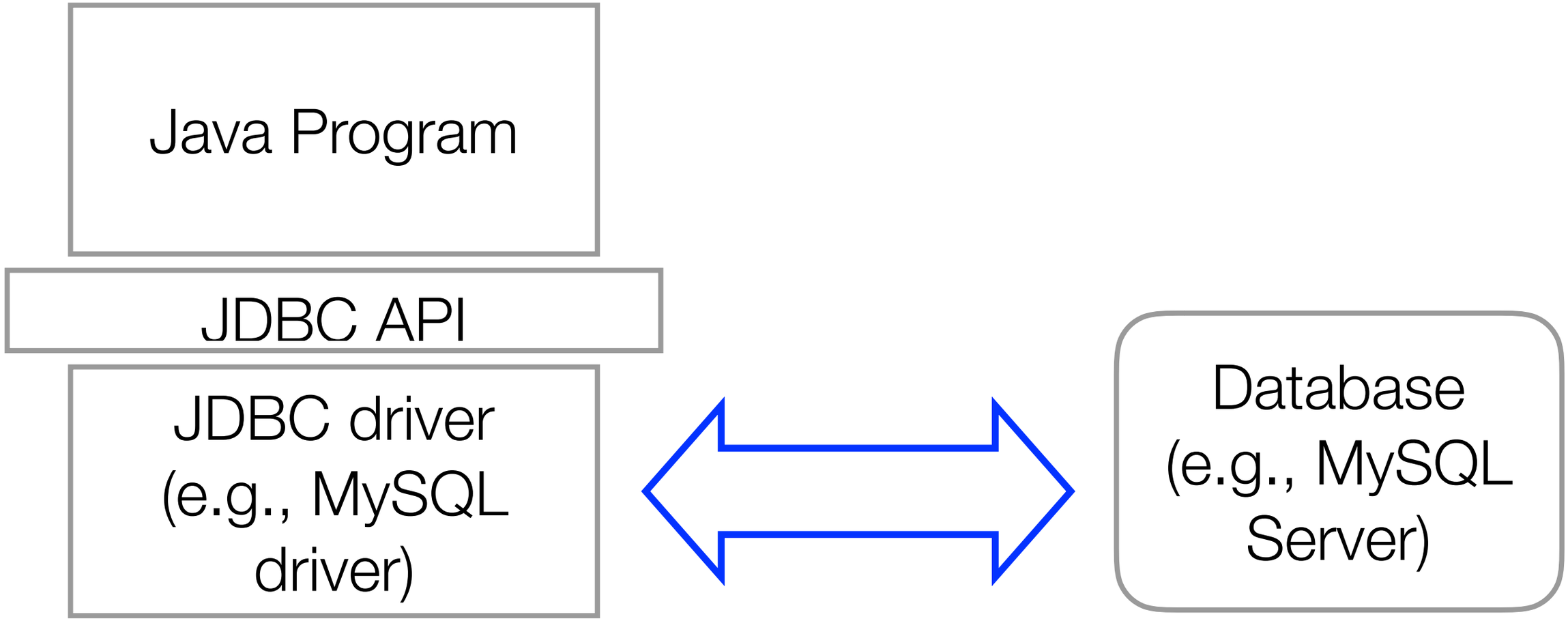
JDBC Program Steps

- Import JDBC library (java.sql.*)
- Load appropriate JDBC driver
- Create a connection object
- Create a statement object
- Submit SQL statement
- Process query results
- Close connections

JDBC: java.sql Package

- Library functions are contained in the java.sql package
- Every JDBC must import the classes in this package **import java.sql.***
- Designed to access any database platform
  - Un-avoidable that there will be a system-dependent component to access a specific database type
  - Drivers are used to support communication transparency between the different vendors

JDBC Driver


Java Program
JDBC API
JDBC driver (e.g., MySQL driver)
Database (e.g., MySQL Server)
direct calls using specific database protocols

JDBC Driver

- A communication driver is a system dependent software module that is written specifically according to a given communication protocol

- Different vendors can provide the same data service through different communication protocols

- A JDBC program must first load the desired communication driver

    - Each system has its own way to load the driver (Biggest headache in JDBC programming)

Dealing with SQLException

Most methods in the JDBC SQL library will throw SQLException

- Catch the exception

```
try
{
 < method in java.sql package >
}
catch ( Exception e)
{
< statements to execute when there is an error>
}
```

- Specify throws SQLException to each method in your program - exit program upon error

DriverManager: Managing the JDBC Driver

- Attempt at standardization of loading the JDBC communication driver
- Contains methods for managing a set of JDBC drivers
- Methods contained in the class:
  - **static void registerDriver(Driver driver)** - registers the given driver with the device manager by reading in the driver code from the installed library
  - **static Connection getConnection(String url, String user, String password)** - attempts to establish a connection and should only be used after the driver was registered

Registering a Driver (Textbook Way)

- Standard way to register a platform dependent JDBC driver is to use the registerDriver() method

- Example: registering the JDBC driver for Oracle: **DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());**

- Unfortunately not all vendors use this approach to load its JDBC driver (e.g., MySQL)

Registering a MySQL Driver

- Exploits Java's built-in capability to load a class
- Driver is loaded using the java.lang.reflect package
- Syntax:

```
Class.forName( "com.mysql.jdbc.Driver" );
```

Dynamic Loading Feature

- Java has the ability to load user-written classes dynamically into a compiled program and execute it
    - Load a different class that has a method with the same name, you can get the behavior of the method to change
- Example: 2 Java classes, Add and Sub, each with a method with the same name main
    - Compile and run each separately

Add.java

```
public class Add
{
	public static void main (String args[])
	{
	 int a = Integer.parseInt(args[0]);
	 int b = Integer.parseInt(args[1]);

	 System.out.println("Sum = " + (a + b));
	}
}
```

Sub.java

```
 public class Sub
{
    public static void main (String args[])
    {
     int a = Integer.parseInt(args[0]);
     int b = Integer.parseInt(args[1]);

     System.out.println("Difference = " + (a - b));
    }
}
```

Dynamic Loading: java.lang.reflect

- Commonly used by programs which require ability to examine or modify runtime behavior of applications
  - Applications: create instances of objects using their fully-qualified names
  - Debuggers & Test Tools: examine private members of classes
    - Class browser: enumerate members of a class
- Drawbacks:
  - Performance overhead
  - Security restrictions
  - Exposure of internals

Java Demo (DynamicLoader.java)

Location of the JDBC Driver Software

- Java must be able to find (locate) the JDBC Driver
    - CLASSPATH variable must point to the SQL Java JDBC library (depends on your installation)
- Include the PATH to run the JDBC program
- Example:
  java -cp <location of jdbc driver library> <your program>

Create a Connection Object

- Network connection to a database server is established using the getConnection method in the DriverManager class

- Syntax:
  Connection SQLconnection; // variable for connection
  SQLconnection = DriverManager.getConnection(URL, user, password);

- Parameters:
  - URL = location of the database server
  - user = userid
  - password = password associated with userid

JDBC: SQL Connection

- Connection contains a reference to the data structure that stores information on the network connection

- Connection must be passed to subsequent methods to communicate with the MySQL server

- Only need a single connection to the server

- URL must contain the protocol, the host name, the port number, and the database name (e.g., “jdbc:mysql://cs377spring16.mathcs.emory.edu:3306/companyDB”)

Creating a Statement Object

- java.sql.Statement class is used to execute a SQL statement (by sending it to the database)

- It also has buffers to receive the result tuples

- Before submitting a query, you must first create a Statement object for the processing of the query

- Syntax:
  Statement SQLstatement; // variable ref for obj
  SQLstatement = <sqlconnection>.createStatement();

Submit a SQL Query

- executeQuery method sends the SQL query using the DBMS connection to the DBMS server for processing

- Syntax:
  ResultSet rset; // reference variable for results
  rset = SQLstatement.executeQuery(“<SQL query>”) ;

- Example:
  ResultSet rset; // reference variable for results
  rset = SQLstatement.executeQuery(“select * from employee”);

Submit a SQL Query (2)

- Statement object can be recycled if SQL queries are executed in serial
    - Execute one query and read the result completely before executing next query
- For multiple queries at the same time, you need to create multiple Statement objects — one per parallel query

Submit a SQL Update

- executeUpdate method sends the SQL command using the DBMS connection to the DBMS server for processing

- SQL command maybe an INSERT, UPDATE, or DELETE statement or even creation of a table or constraint

- Returns an update count

Process Query Results

- ResultSet returns an iterable that contains all the tuples in the output relation

- Retrieve one tuple:
  rset.next() - returns null if there are no more tuples otherwise returns the next tuple

- Retrieve all tuples in the result set

```
while (rset.next() != null) {
   <process the tuple>
}
```

Closing Connections

- Upon completion, you should close the various connections and free resources

- Close and free the result set:
  rset.close();

- Close and free the Statement object
  SQLstatement.close();

- Close and free the Connection buffer
  SQLconnection.close();

JDBC Demo (Employee.java)

Useful ResultSet Methods

- **beforeFirst()**: moves the read cursor to the front of the ResultSet object, just before the first row (can be used to re-read the data again)
- **first()**: moves the read cursor to the first row of the ResultSet object
- **absolute(rowNumber)**: moves the read cursor to the row rowNumber of the ResultSet object
- **afterLast()**: moves the read cursor to the end of this ResultSet object, after the last row (can be used to read the data in reverse order)
- **last()**: moves the read cursor to the last row of this Result object (can be used to find number of rows)
- **getRow()**: returns the row index of the current row

JDBC Demo (Employee2.java & Employee3.java)

Retrieve a Field in the Result Tuple

Use getter methods to retrieve attribute values from the current row

- **rset.getString(index)**: returns attribute at position index as a String
- **rset.getInt(index)**: returns attribute at position index as int type
- **rset.getFloat(index)**: returns attribute at position index as float type
- **rset.getDouble(index)**: returns attribute at position index as double type

Common Java and SQL Type Equivalence

Java Method	SQL Type
getInt	INTEGER
getLong	BIG INT
getFloat	REAL / FLOAT
getDouble	DOUBLE
getBignum	DECIMAL
getBoolean	BIT / BOOLEAN
getString	VARCHAR / CHAR
getDate	DATE
getTime	TIME
getTimeStamp	TIMESTAMP
getOjbect	any type

Metadata About ResultSet

- ResultSetMetData class contains meta information about the ResultSet
  - Methods to retrieve/obtain the meta data
  - Variables to store values of the meta data
- Syntax:

```
ResultSet rset;
rset = SQLstatement.executeQuery("<SQL query">);
ResultSetMetaData metaData;
metaData = rset.getMetaData();
```

Useful ResultSetMetaData Methods

- **int getColumnCount()**: returns the number of columns in the tuples of the ResultSet

- **String getColumnName(int columnIndex)**: returns the name of the column whose index is specified

- **String getColumnType(int columnIndex)**: returns the integer code for the data type of the attribute whose index is specified (see java.lang.Types for the codes)

- **String getColumnTypeName(int columnIndex)**: returns the type of column whose index is specified

Useful ResultSetMetaData Methods (2)

- **int getColumnDisplaySize(int columnIndex)**: returns the display width (number of characters needed to display the value) of the attribute whose index is specified

- **int getPrecision(int columnIndex)**: returns the number of digits of the field/column whose index is specified - data type must be numeric

- **int getScale(int columnIndex)**: returns the number of decimal places of the field/column whose index is specified - data type must be numeric

- **String getColumnClassName(int columnIndex)**: returns the name of the Java class (e.g., “java.lang.String”) for the attribute whose index is specified

JDBC Demo (MetaData.java)