320341 Programming in Java

JACOBS UNIVERSITY

Fall Semester 2014

Lecture 16: Introduction to Database Programming

Instructor:	Jürgen Schönwälder
Slides:	Bendick Mahleko

Objectives

This lecture introduces the following

- Basic JDBC programming concepts
- Query execution
- Transactions
- Connection management

Overview

First version of **Java Database Connectivity (JDBC)** in 1996

- ***De facto industry standard*** *for database-independent connectivity between the Java programming language and a wide range of databases for example Microsoft SQL Server, Oracle, Informix, MySQL etc*
- *Allows Java program access to any database using* ***standard SQL statements***
- *Java programs communicate with databases and manipulate their data using the* ***JDBC^{TM} API***
- *A* ***JDBC Driver*** *enables Java applications to connect to a database in a particular DBMS and allows to manipulate the database using* ***JDBC API***

Overview

Several JDBC versions have been released

- Current specification: JDBC 4.0
- JDBC 3.0 is included in JDK 1.4, 5.0, 6.0 & 7.0

JDBC API

- Pure Java API for SQL access for application programmers
- JDBC 3.0 includes 2 packages: `java.sql & javax.sql(server side)`

JDBC Driver API

- Third party drivers to connect to specific databases
- You can locate rivers for your DBMS from the vendor

DB examples

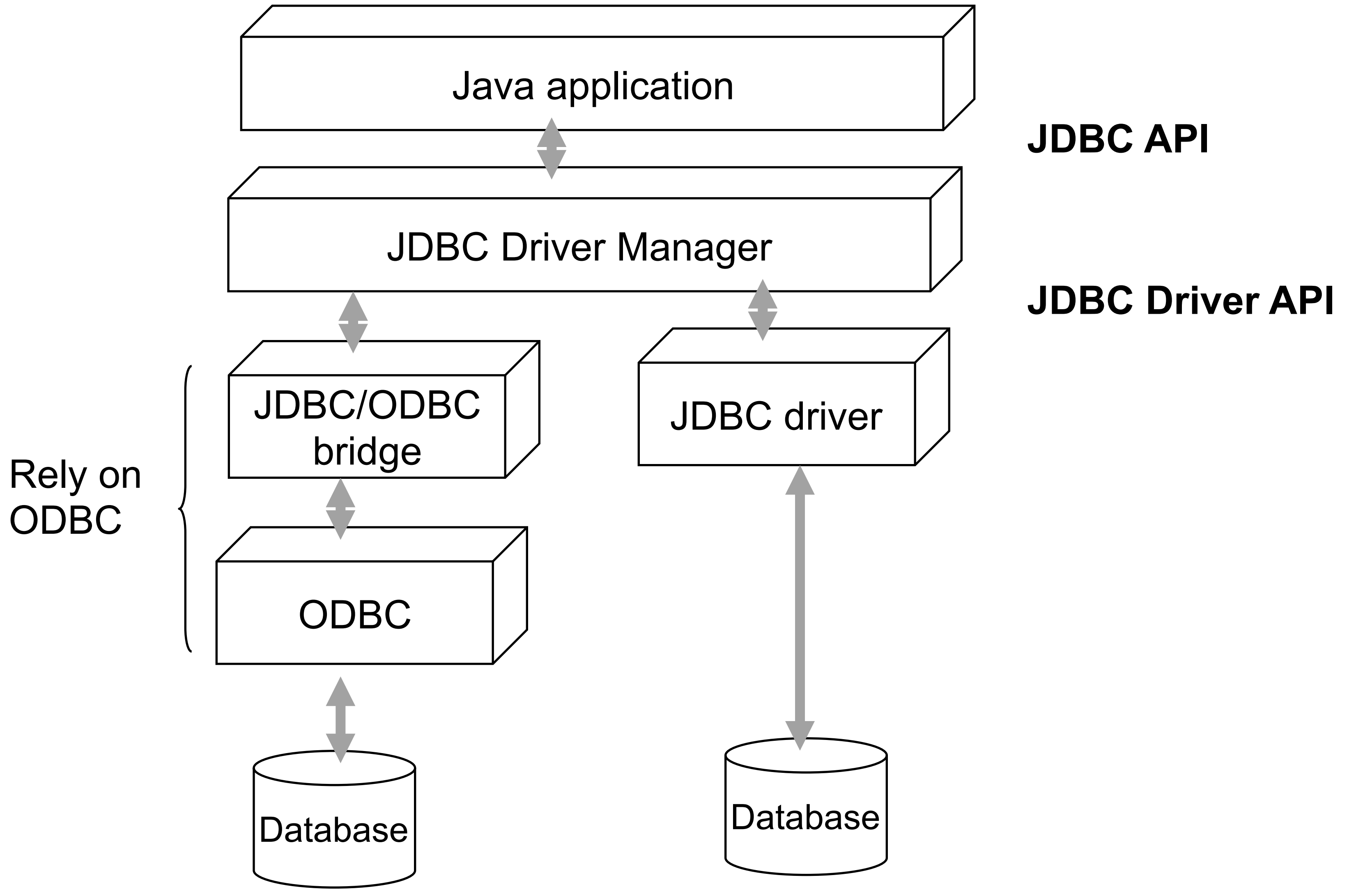
JDK comes with a pure-Java RDBMS called Java DB

Other examples of relational DBMS are:

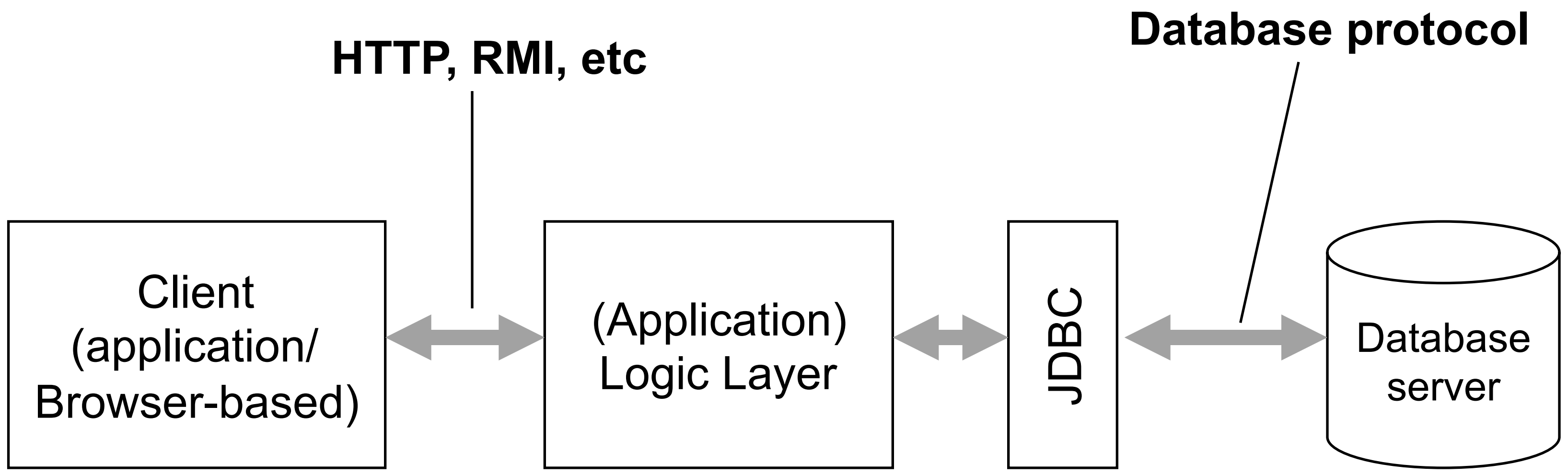
- Microsoft SQL Server
- Oracle
- Sybase
- IBM DB2
- Informix
- PostgreSQL
- MySQL

(source: Deitel etc., “Java: how to program”, 9th edition)

JDBC to Database



Typical Architecture


HTTP, RMI, etc
Database protocol
Client (application/ Browser-based)
(Application) Logic Layer
JDBC
Database server

Structured Query Language (SQL)

JDBC lets you communicate with databases using SQL

- SQL is the command language for most modern relational databases

- JDBC package can be regarded as an API for communicating SQL statements to databases

Structured Query Language (SQL)

Database URLs

- Specify a **data source** when connecting to a database
- JDBC uses syntax similar to ordinary URLs to describe data sources

- Ex: Specify local Derby database & a PostgreSQL database named COREJAVA

```
jdbc:derby://localhost:1527/COREJAVA;create=true
jdbc:postgresql:COREJAVA
```

$$\texttt{jdbc:}\underbrace{\texttt{subprotocol}}_{\uparrow}\texttt{:other stuff}$$

Selects specific driver for connecting to database

Structured Query Language (SQL)

Connecting to database

- Find the names of classes used by vendor (download JAR file)

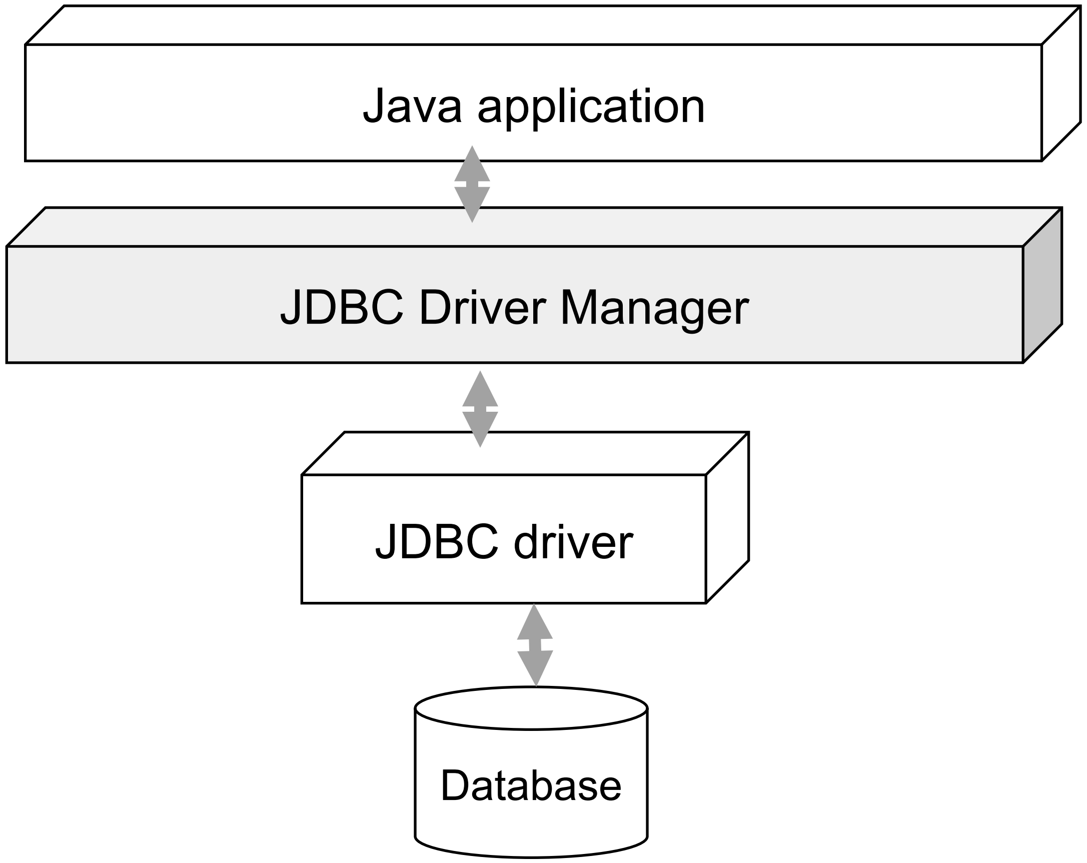
Find the library in which the driver is located e.g., mkjdbc.jar

- Launch your programs with `-classpath` command line argument OR
- Copy the database library into the `jre/lib/ext` directory

Structured Query Language (SQL)

The **DriverManager** [package `java.sql`]

- Selects database drivers & creates new database connections



Structured Query Language (SQL)

Registering Drivers

- A driver must be registered before the drive manager can activate it
- There are two methods

```
java -Djdbc.drivers=org.postgresql.Driver MyProg
```

- or set a system property with the call

```
System.setProperty(“jdbc.drivers”,”org.postgresql.Driver
”
```

- Supply multiple drivers, separated with colons
- `org.postgresql.Driver:com.mckoi.JDBCDriver`

Open Connection

After registering drivers *open a connection*

Example

```
String url = “jdbc:postgresql:COREJAVA”;
String username = “dbuser”;
String password = “secret”;

Connection conn = DriverManager.getConnection(url, username, password);
```

The driver manager iterates over the registered drivers to find the driver which can be used by the specified subprotocol in the database URL

Open Connection

The driver manager iterates over available drivers until it finds a matching subprotocol

You can use a property file to specify the *URL*, *user name* etc

```
jdbc.drivers=org.postgresql.Driver
jdbc.url=jdbc:postgresql:COREJAVA
jdbc.username=dbuser
jdbc.password=secret
```

The **Connection** object returned by the *getConnection* method is used to execute SQL statements

Executing SQL Commands

First, create a *statement* object

- Use the **Connection** object from the call to DriverManager.getConnection

```
Statement stat = conn.createStatement();
```

Next, place the statement you want to execute into a string e.g.

```
String command = "Update Books" +
                 " SET Price = Price – 5.00" +
                 " WHERE Title NOT LIKE '%Introduction%'";
```

Then call the **`executeUpdate`** method of the Statement class

```
stat.executeUpdate(command)
```

The `executeUpdate` method returns a count on the rows affected by the SQL command

Executing SQL Commands

The **`executeUpdate`** method

- Can execute actions such as **INSERT**, **UPDATE**, and **DELETE**
- Can execute data definition commands such as **CREATE TABLE** and **DROP TABLE**

The **`executeQuery`** method

- Use **`executeQuery`** to execute **SELECT** queries
- Returns an object of type **ResultSet**
- Use **ResultSet** to walk through results row by row

```
ResultSet rs = stat.executeQuery(“SELECT * FROM Books");
```

Executing SQL Commands

Basic loop for analyzing results:

```
ResultSet rs = stat.executeQuery(“SELECT * FROM Books");

while (rs.next()) {
   look at a row of the result set
}
```

Reading fields:

- Accessor methods are supplied to read field information
- Each accessor has two forms: *takes numeric argument* & *takes string argument*

```
String isbn = rs.getString(1);
double price = rs.getDouble(“Price”);
```

Managing Connections

Every **Connection** object can create one or more **Statement** objects

- We can use the same statement for multiple unrelated commands & queries
- A statement has *at most one open* result set
- If you issue multiple queries whose results you analyze concurrently, then you need multiple `Statement` objects

Managing Connections

Freeing resources

- When done using a `ResultSet`, `Statement` or `Connection`, call `close` method immediately
- `close` method of `Statement` object automatically closes associated result set if one exists
- `close` method of `Connection` class closes all statements of the connection

Managing Connections

Ensure that the connection object does not remain open

```
Connection conn = …;

try {
   Statement stat = conn.createStatement();
   ResultSet result = stat.executeQuery(queryString);
   process query result
} finally {
    conn.close();
}
```

Transactions

Group a number of statements into a transaction

- A **transaction** can be committed if all has gone well
- The **transaction** can be rolled back if an error has occurred
- The purpose is to ensure database integrity

Default, every SQL command is committed to database after execution

- Can’t be rolled back
- Turn off autocommit

```
conn.setAutoCommit(false);
```

Transactions

- Now create a statement object as usual

```
Statement stat = conn.createStatement();
```

Call `executeUpdate` any number of times

```
Stat.executeUpdate(command1);
Stat.executeUpdate(command2);
Stat.executeUpdate(command3);
…
```

Then call `commit` when all commands executed successfully

```
conn.commit();
```

Otherwise, rollback if error occurred

```
conn.rollback();
```

Reading Assignment

- Horstmann, C. S. & Cornell, G. (2008) Core Java 2, Volume II, 8th Ed. Ch. 4., Prentice Hall.

- Oracle (n.d.) JDBC Overview. http://www.oracle.com/technetwork/java/overview-141217.html (Last visited 23 November 2012).

- Oracle (n. d.) JDBC Introduction [online]. Available from: http://download.oracle.com/javase/tutorial/jdbc/overview/index.html (Last visited 23 November 2012).