Databases - JDBC I
This lecture

The lecture starts discussion of how a Java-based application program connects to a database using JDBC.
Application Programs

The `mysql` client program provides a low-level interface to the raw commands of MySQL, and as such is suitable only for expert users writing their own SQL/MySQL queries.

However there are many reasons to connect other application programs to an SQL and/or MySQL database — these programs may essentially be about providing access to the database in a special way or they may just use the database “in the background” to provide other functionality.
An organization is likely to have only a few DBA professionals, but many more users who need to query the database.

Suitable client programs can be written to provide simplified interfaces, usually *graphical user interfaces* for end-users. These interfaces would usually

- Customize the handling and error-checking of input
- Construct valid SQL queries from the input
- Run the SQL queries against the database
- Customize and format the output
A second use for additional client programs would be to provide expert users with specialized tools for visualizing and working with a whole database — we have already seen such a tool in *DBVisualizer* and there are many more such tools.
Database Integration

As well as client programs whose *primary purpose* is providing database access, there are many situations where a client program may benefit from database integration.

In this situation the client program may wish to *use* a database in order to accomplish its primary purpose—for example, a management information system may access a database to obtain recent sales figures, but then perform significant additional processing or analysis on the results.
Database Backed Applications

A third reason to connect an application to a database is to use the database as a *tool* to provide the essential functionality of the application — here the database is supporting the use of the application, rather than the application supporting the use of the database.

The typical example of such an application is a web-based bulletin board system where users post and view comments, perform searches and so on. Frequently the actual comments are stored in a database, and the web-pages that are displayed are constructed “on-the-fly” from the results of database queries.

In this situation the user may not even be aware of the underlying database.
Connecting

There are many different ways of connecting an application written in a general purpose programming language (henceforth called the host language) to a MySQL server.

- The C API is a low-level client library for connecting C programs directly to a MySQL server.
- The Perl and PHP APIs provide higher level APIs for connecting to a number of different databases from Perl or PHP respectively.
- JDBC is a standard API for connecting Java programs to a number of different databases.

In this lecture, we will consider how to connect Java applications to MySQL.
What is JDBC?

JDBC stands for Java DataBase Connectivity — it is an API (Application Programmers Interface) that defines a set of interfaces for interaction with a database.

Each database vendor writes a specific JDBC driver which is a set of classes that implements the JDBC interface on their database. For example, Oracle has a JDBC driver for Oracle databases, MySQL has a JDBC driver for MySQL databases and so on.

If the application programmer also writes code that conforms to the JDBC standard, then the application can interact with a number of different databases simply by loading a different driver.
Conceptual Diagram

Diagram from “Java Enterprise in a Nutshell” by Crawford, Farley & Flanagan, O’Reilly.
JDBC Architecture

A JDBC (Java DataBase Connectivity) application can be viewed as consisting of four components:

- The Java *application program*
  The application initiates a connection with the server, submits SQL statements and processes the results.

- The *driver manager*
  The driver manager loads JDBC drivers and passes the JDBC calls to the correct driver.

- The *drivers*
  A driver is database-specific code that *translates* JDBC calls into the right form for its particular database, and translates the results back into the JDBC standard.

- The *data sources*
  The underlying database or databases.
JDBC Basics

A JDBC application must perform the following steps:

- Load the driver class for the particular database being used.
- Open a *connection* to the database.
- Create SQL queries and use the connection to *execute* them.
- Process the results obtained from the database.
Loading a driver

In your class file you must always import Java’s sql package.

```java
import java.sql.*
```

A driver is loaded by using Java’s dynamic loading mechanism `Class.forName` with the correct driver name, which for the MySQL driver is `com.mysql.jdbc.Driver`.

This method might throw an exception, so this must be caught.

```java
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
    System.out.println("Cannot load Driver");
}
```
Opening a connection

Once the driver has been loaded the static methods in the class DriverManager\(^1\) become available; the important method here is the method

```java
public static Connection getConnection(String url)
throws SQLException
```

This method takes a `String` in a URL-like format

```
jdbc:driver:database
```

which specifies JDBC, then the name of the driver and then connection details for the database (user name, password if any, and database).

\(^1\)The JDBC-related classes discussed here are in the package `java.sql`.

(GF Royle, N Spadaccini 2006-2010)
Opening a local connection

Connecting as the root user to the world database stored on a MySQL server running on the local host would be done as follows:

```java
try {
    String url = "jdbc:mysql://localhost/world?user=root";
    Connection conn = DriverManager.getConnection(url);
} catch (SQLException e) {
    System.out.println("Cannot open Connection");
    System.exit(0);
}
```

The precise form of the URL will depend on the particular driver, but in all cases the result is the object `conn` which represents an open connection to the database.
Statements

An object from the `Statement` class is used to send your SQL statements to the database — such an object is obtained from the `Connection` object that we just obtained from the `DriverManager`.

```java
try {
    String url = "jdbc:mysql://localhost/world?user=root";
    Connection conn = DriverManager.getConnection(url);
    Statement stmt = conn.createStatement();
} catch (SQLException e) {
    System.out.println("Cannot open Connection");
    System.exit(0);
}
```
Queries

An actual SQL query is executed by the method

```java
public ResultSet executeQuery(String sql)
```

of the `Statement` object.

A query that executes successfully returns a `ResultSet` object which represents the table that the SQL query returns.

It is important to note that the `ResultSet` object “belongs” to the `Statement` object and that if the `Statement` object is re-used for another query, then the `ResultSet` object will also be closed.
Executing a query

Suppose we simply want to execute a query that returns the data stored in the `country` table from our database.

```java
Statement stmt = conn.createStatement();
String sql = "SELECT * FROM country";
ResultSet rs = stmt.executeQuery(sql);
```

(This code must all be placed within the `try/catch` clause because pretty much all of these methods might throw an `SQLException`.)
Processing the results

A ResultSet object can be viewed as a table of results that can be stepped through “row-by-row” — at each stage the object has a notion of a “current row”.

In order to process the results we need methods to

- Change the current row
- Determine column values from the current row
Changing the current row

The key methods for moving through a ResultSet are

- `public boolean first()`  
  This moves to the *first row* of the result set

- `public boolean next()`  
  This moves to the *next row* of the result set, returning `false` if there are no further rows to move to.

In earlier versions of JDBC a result set could *only* be processed by starting at the first row and going through one row at a time. This is still the JDBC specification of the default ResultSet but the MySQL drivers return a “scrollable” ResultSet no matter what you ask for.
Process each row

Thus to examine each row in turn uses code of the following form (assume that rs has already been obtained in the manner described above).

```java
while (rs.next()) {
    // process the row
}
```
Reading column values

Reading a column value from the current row of the ResultSet is done by one of the `getXXX()` methods, where `XXX` is a Java type. For example, any column that is a `CHAR`, `VARCHAR` or `TEXT` type in the database can be accessed by using

- `getString(String columnName)`
- `getString(int columnIndex)`

where the column to be retrieved can either be specified by `name` or by `position`.

Just to confuse everyone, the columns in SQL are indexed from position 1, not the 0-based indexing of Java, C, C++ etc.
Brief Example

This code simply pulls out the columns name and population from each row of the ResultSet and prints them out.

```java
while (rs.next()) {
    String cntry = rs.getString("name");
    long popn = rs.getLong("population");

    System.out.println(cntry + " " + popn);
}
```
What have we achieved?

In one sense, all we have done is written a Java client program that executes the SQL statement

```sql
SELECT name, population FROM country;
```

in a round-about way.

The important thing to realize is that we now have the database data inside the Java program where it can be post-processed in any number of ways.