Introduction To Java and Databases

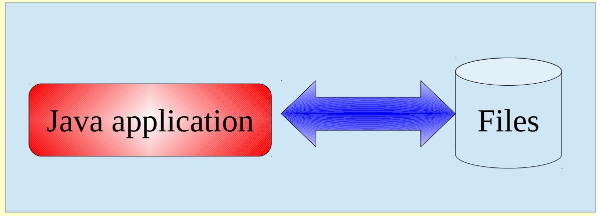
Table of Contents

1 Java and databases

1.1 What is a database?

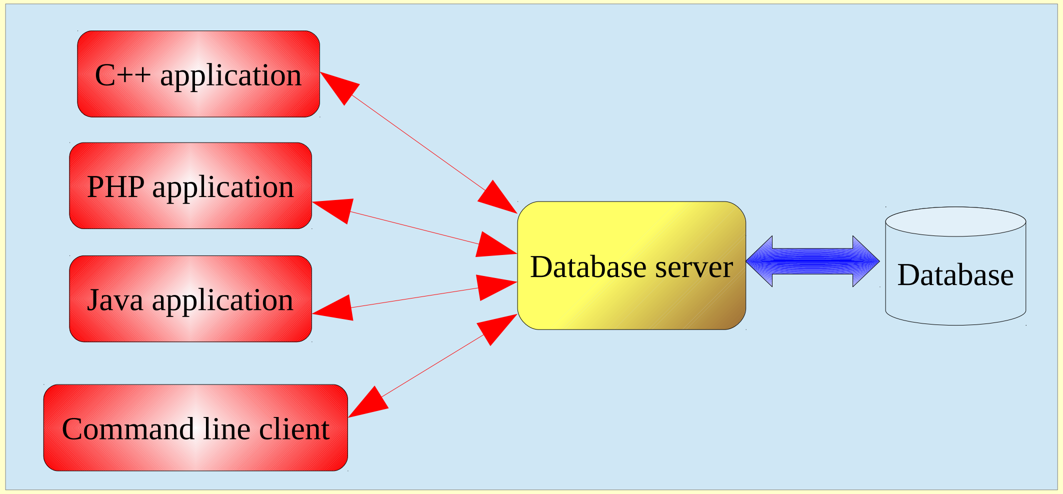
In general terms, a database is just a store information, usually on a computer system. The data is held in files on non-volatile media, which keep their contents when power is removed – such as a hard disc.

A Java program can read and write data files directly. These can be text files (containing text only, no formatting like bold or italic), or binary files containing for example int, doubles, graphics like .gifs and jpegs, or OOP objects. Tutorials on Java I/O describe how this is done, using Readers and Writers, BufferedReaders, FileStreams ObjectStreams and so on.



But usually by a database we mean a data store which is accessed through a database server. Instead of dealing with the data files directly, the Java program uses the server to handle the data like this:

A database server is just a software application.



There are many advantages to this idea. The database server has only one task, to handle the database, so its code is specialised to that function, meaning it can handle very large amounts of data at high speed and with security. It means we are not restricted to using Java – we can use any application language, or issue commands to the server directly. The database can handle many connections at the same time, so we can have many users on the same database at the same time. And the database server and data might be on different machines, and the application program could be on different machines over a network – maybe on different continents.

1.2 Relational databases

There are different types of database – the common ones are ***relational***, ***hierarchical*** and ***network*** (which means that the data is structured in a net – not that the compters are networked). Of these, a relational database (***RDBMS***) is currently the most common.

The idea of a relational database came from Edgar Codd in 1970 (when he was working for IBM). A relational database consist of *tables* of data. Each table has rows and columns, and all data is held in the cells in these tables. This model is very simple.

1.3 Commercial vendors

Several companies offer RDBMS servers:

- Oracle
- Microsoft SQL Server
- MySQL (now owned by Oracle)
- IBM DB2
- IBM Informix
- PostgreSQL

1.4 SQL

Commands are normally issued to the database server in SQL, Structured Query Language. This is a language specifically concerned with RDBMS. It became an ISO standard in 1987, going through several versions since. Different vendors use slight variations on the standard, but core commands are the same. We can issue SQL commands directly to a server, or from a Java program (or from most other general purpose languages).

This text will use MySQL for examples – but since the idea of a RDBMS, and SQL, is common to the different vendors, other databases are also covered.

1.5 Information systems

It is tempting to focus on code. But something which uses a database is an ***information system***. An information system contains the following:

- Computer hardware and system software and communication infra-structure
- Application software (your Java program)
- A database server and a database
- Forms and reports (on paperand on screen)
- Real-world procedures
- System administrators
- Users

The design of the application program and the database are consequences of the design of the system. In other words, you do not start writing code, or setting up a database. You start by analysing an existing information and designing a new one. From that design, and the analysis of who needs what information and when and how, you design the database, and what the application software needs to do.

1.6 Table terminology

Suppose in a library management system we have a table of books. It might be like this:

BookID	Title	Author	DateAcquired	ISBN	LoanCount
1	War and Peace	L. Tolstoy	1.1.2001	978-3-16-148410-0	4
2	Keeping Chickens	Mike Hatcher	7.5.2007	975-3-16-148510-2	3
3	Java for Fun and Profit	John Jones	8.8.2010	178-3-16-148416-4	0
4	XML for Beginners	J. Jones	1.2.2008	278-3-16-148413-5	2
5	How to Bake Cakes	D. Smith	5.5.2010	338-3-16-148411-2	1

The meanings of the columns are:

BookID – simply identifies the book uniquely. Each book has a different ID. If War and Peace is a popular book, we might have several copies of it. In that case we would have several books with the same title, but each would have a different ID so we can tell which copy is which. Typically we let the server choose the ID to ensure it is unique.

Title – The title of the book

Author – The author. Note there might be a problem – is J. Jones and John Jones the same person?

DateAcquired – The date the library obtained the book. Note there is an issue about the date format. Is 1.2.2008 the first of February or the second of January?

ISBN – The ISBN. This might be a number, but it is split into different sections, separated by a -, so we take it as a string not a number.

LoanCount – The number of times the book has been borrowed.

A ***column*** is sometimes called a ***field*** or an ***attribute***

A ***row*** is sometimes called a ***record***. Each row represents one 'thing' which the table is about. Real databases have tables with maybe less than ten rows to millions of rows.

Each column has a ***data type***. This is the same idea as a Java type, but different in detail. BookID and LoanCount would be integers, DateAcquired a date, Title a string and so on. Ideally a column has a ***domain*** – a set of allowed values, or some ***validation*** rule. For example we might require an ISBN to have the format of digits and dashes as shown, and to require the LoanCount to be greater than -1.

In this table we would have BookID as the primary key. The primary key uniquely identifies the record. To create a record with a key value the same as an existing row would fail. The server would typically create an index on a table on its primary key, so that records with a given key can be found quickly.

Since the primary key identifies the row, it should be impossible to alter it.

In our book table, we might have had the ISBN as the key. But in that case we would not be allowed to have two copies of the same book.

There are SQL commands to create tables with columns with given names and types.

It is common to show a table with a shorthand structure like this:

books(<u>BookID</u>, Title, Author, DateAcquired, ISNB, LoanCount)

BookID is underlined to show it is the primary key.

1.7 Entities and database design

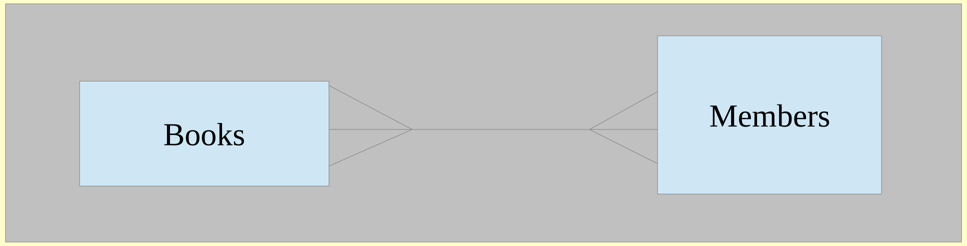
Database design means choosing what tables to have with what columns and keys.

Database design should be preceded by an ***entity analysis*** of the required system. An entity is just a thing about which the system will hold information. People, staff and users, invoices, orders, cheques, departments, and physical items like cars, DVDs, books and phones might all be entities. An entity is a thing (physical or logical) and not a process. But a process (like a sale) will often generate entities – such as an invoice.

An entity is just a thing – but entity analysis sounds cleverer than 'thing analysis'.

Once entities are identified, the relationships between them should be considered. This often reveals more entities.

In our library management system, two obvious entities are books and members (members of the library who borrow books). One relation between these simply that members borrow books. This is a 'many-many' relationship. Over a period of time, one book is borrowed by many members. And over time, one member borrows many books. This is usually shown in an entity-relationship diagram like this:



The 'crows-feet' are intended to show the many-many relationship.

We might set up table structures like these:

members(MemberID, FirstName, LastName, DOB....)

and

books(BookID, Title, Author...)

but how to show who borrowed the book? We could do it like this:

books(BookID, Title, Author..., B1, B2, B3) where B1 would be the MemberID of the first person to borrow the book, B2 is the second person, and B3 is the third. But suppose four people borrow a book? We cannot do this, because we need more and more columns as more people borrow a book, and in a table all rows must have the same number of columns.

We could try to put all the borrowers in one table cell. So a row in the books table might be for example:

BookID	Title	Author	DateAcquired	ISBN	LoanCount	Borrowers
1	War and Peace	L. Tolstoy	1.1.2001	978-3-16-148410-0	4	5,28,3,56,93

This would mean the book had been borrowed first by member number 5, then 28, then 3, then 56 and 93.

Do not do this!

Each table cell should contain just *one* piece of data. Having multiple values in one cell makes it impossible to process the data with SQL queries. A database with a design like this is said to be not ***normalised***.

A problem like this will always arise from a many-many relationship. The issue is usually resolved by identifying another entity. In this case this is a Loan. This table could be like this:

loan(LoanID, BookID, MemberID, DateOut, DateBack)

We invent some data to show how this works. The books are

BookID	Title	Author	DateAcquired	ISBN
1	War and Peace	L. Tolstoy	1.1.2001	978-3-16-148410-0
2	Keeping Chickens	Mike Hatcher	7.5.2007	975-3-16-148510-2

The members are

MemberID	FirstName	LastName	DateOfBirth

1	John	Smith	1.15.1980
2	Jane	Jones	2.2.1990

And the loans table is

LoanID	BookID	MemberID	DateOut	DateBack
1	2	1	1.1.2014	14.1.2014
2	2	2	15.1.2014	
3	1	1	7.1.2014	13.1.2014

This shows that

On the 1st January John borrowed the book on chickens, and returned it on the 14th January

On the 15th Jane borrowed it. The DateBack cell is blank ('null') which means it has not yet been returned.

On the 7th. John borrowed War and Peace, and returned it on the 13th.

We have removed the LoanCount field from the books table, since if we need that information we can find it from the loans table. For the data above, book 2 has been borrowed twice and book 1 once.

This change has good and bad effects:

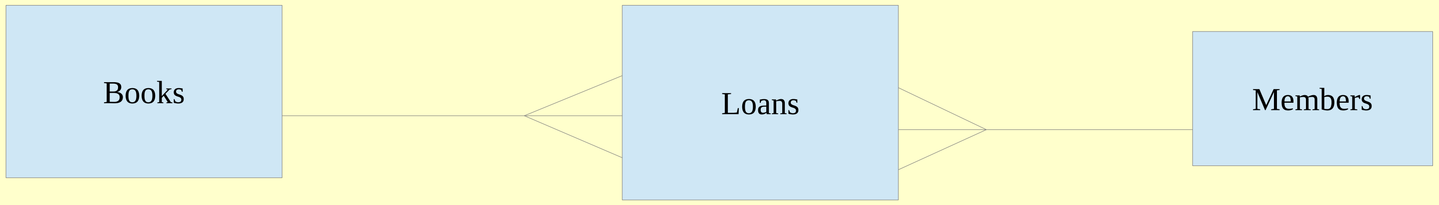
1. We avoid storing data which is implicitly already in the system, so we save storage space.
2. We avoid any possible ***inconsistency***. For example the LoanCount might be 10, when there are only 9 loans in the loan table. If we only have the data once, it cannot be inconsistent.
3. It would take time to calculate the loan count. Typically we can save space in an information system at the expense of time.

In the shorthand form:

loan($\underline{\text{LoanID}}$, $\overline{\text{BookID}}$, $\overline{\text{MemberID}}$, DateOut, DateBack)

BookID and MemberID have lines over them, because they are ***foreign keys***. That is it say, they are primary keys in another table.

On this basis our entity-relationship diagram becomes:



Books to loans is one-many. Over time, each book is loaned out may times. But one loan only refers to one book. Similarly members-loans is one-many. One loan refers to just one member, but over time one member will make many loans.

In a similar way we could have a reservations entity.

reservation($\underline{\text{ReservationID}}$, $\overline{\text{BookID}}$, $\overline{\text{MemberID}}$, DateMade, DateOut)

DateMade would mean the date the member reserved the book, and DateOut would be the date they actually got the book. When they take the book out we could delete the reservation record. But in general we do not delete data – it is useful and cost money to get it. Instead we fill in the DateOut. This will give us useful management statistics about who reserved which books.

1. We would need to design a 'make reservation' procedure:
   Member selects book to reserve
2. System checks book is on loan. We stop if the book is on the shelf
3. Create a corresponding reservation record
4. When a book is returned, a check is made whether it is reserved
5. If it is, it is put to one side (not returned to shelves) and the reserver is notified.
6. On issue of reserved item, put current date as DateOut

This is not purely a computer procedure. The librarian must put the physical book to one side.

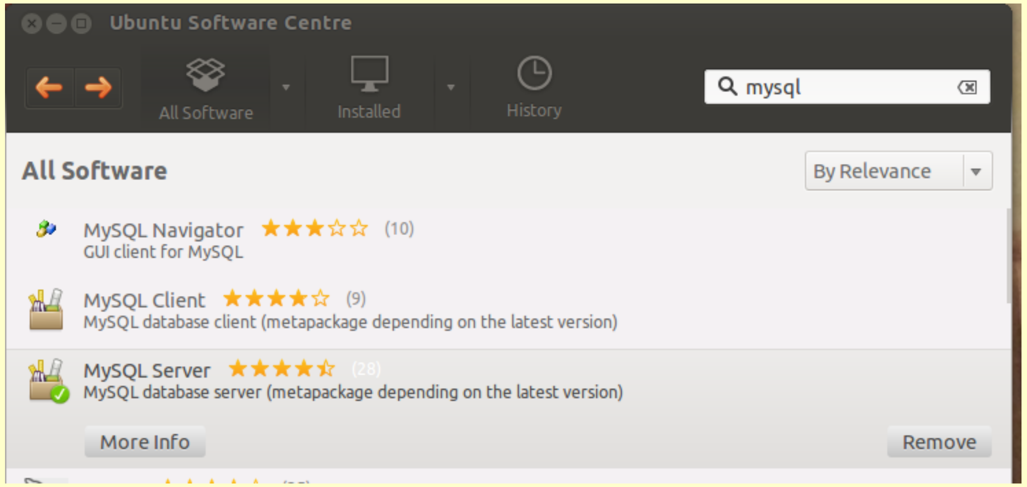
2 MySQL

MySQL started life in 1995 as an open source project, free for personal use, and it became very widely offered as a default database on hosted servers. In 2008 Sun Microsystems bought it for $1 billion, and the following year Oracle bought Sun. As of 2014 MySQL was the second most widely used open source RDBMS, after Sqlite which is used on iPhone and Android phones.

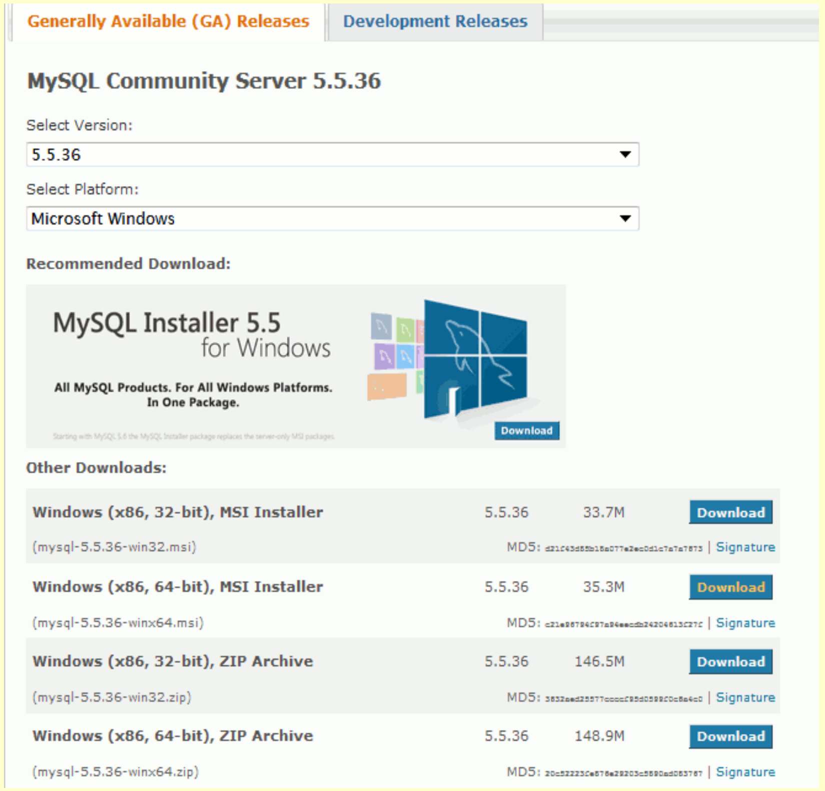
MySQL Workbench is a GUI application to administer MySQL and design databases. Because we want to emphasise SQL (since we need it for Java applications), we will look at setting up and using a database directly from SQL at a command line.

2.1 Installation

How to do this depends on the operating system and version. For Ubuntu we can do it simply from the Software Centre:



For Windows, we need to download the appropriate version (Community Server, 32 or 64 bit):


Generally Available (GA) Releases
Development Releases

MySQL Community Server 5.5.36

Select Version: 5.5.36

Select Platform: Microsoft Windows

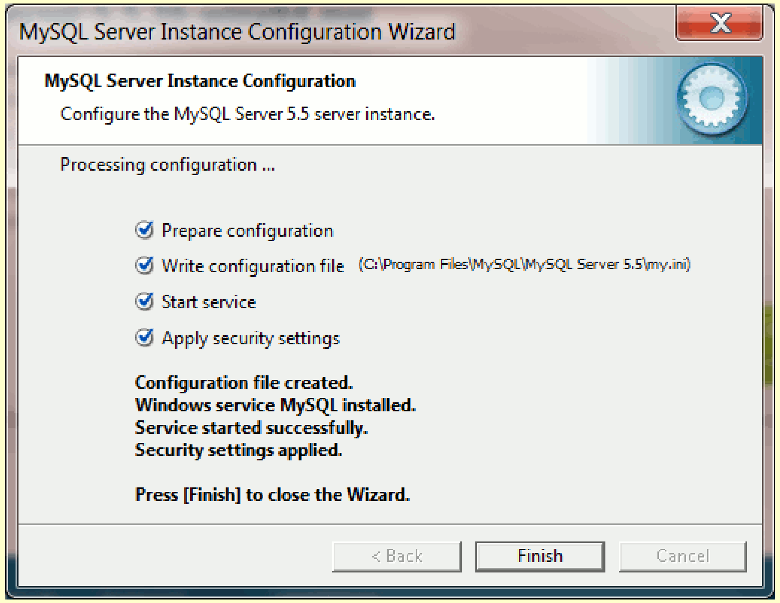
Recommended Download:

MySQL Installer 5.5
for Windows
All MySQL Products. For All Windows Platforms.
In One Package.
Starting with MySQL 5.6 the MySQL Installer package replaces the server-only MSI packages.
Download

Other Downloads:

Windows (x86, 32-bit), MSI Installer	5.5.36	33.7M	Download
(mysql-5.5.36-win32.msi)		MD5: d21f43d85b16a077e2ec0d1c7a7a7873 | Signature	
Windows (x86, 64-bit), MSI Installer	5.5.36	35.3M	Download
(mysql-5.5.36-winx64.msi)		MD5: c21e86794f97a94eecdb24204613f27c | Signature	
Windows (x86, 32-bit), ZIP Archive	5.5.36	146.5M	Download
(mysql-5.5.36-win32.zip)		MD5: 3832aed25577ccccf85d0598f0c8a4c0 | Signature	
Windows (x86, 64-bit), ZIP Archive	5.5.36	148.9M	Download
(mysql-5.5.36-winx64.zip)		MD5: 20c52223fe876e29203c5590ad053767 | Signature	

and follow through the installation wizard:


MySQL Server Instance Configuration Wizard
MySQL Server Instance Configuration
Configure the MySQL Server 5.5 server instance.

Processing configuration ...

- ✓ Prepare configuration
- ✓ Write configuration file (C:\Program Files\MySQL\MySQL Server 5.5\my.ini)
- ✓ Start service
- ✓ Apply security settings

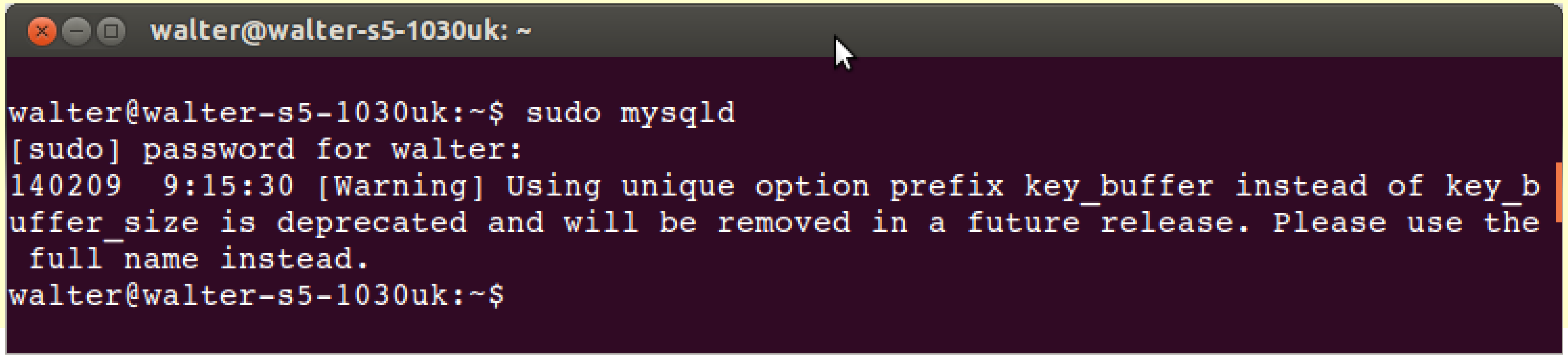
Configuration file created.
Windows service MySQL installed.
Service started successfully.
Security settings applied.

Press [Finish] to close the Wizard.

2.2 Server and client

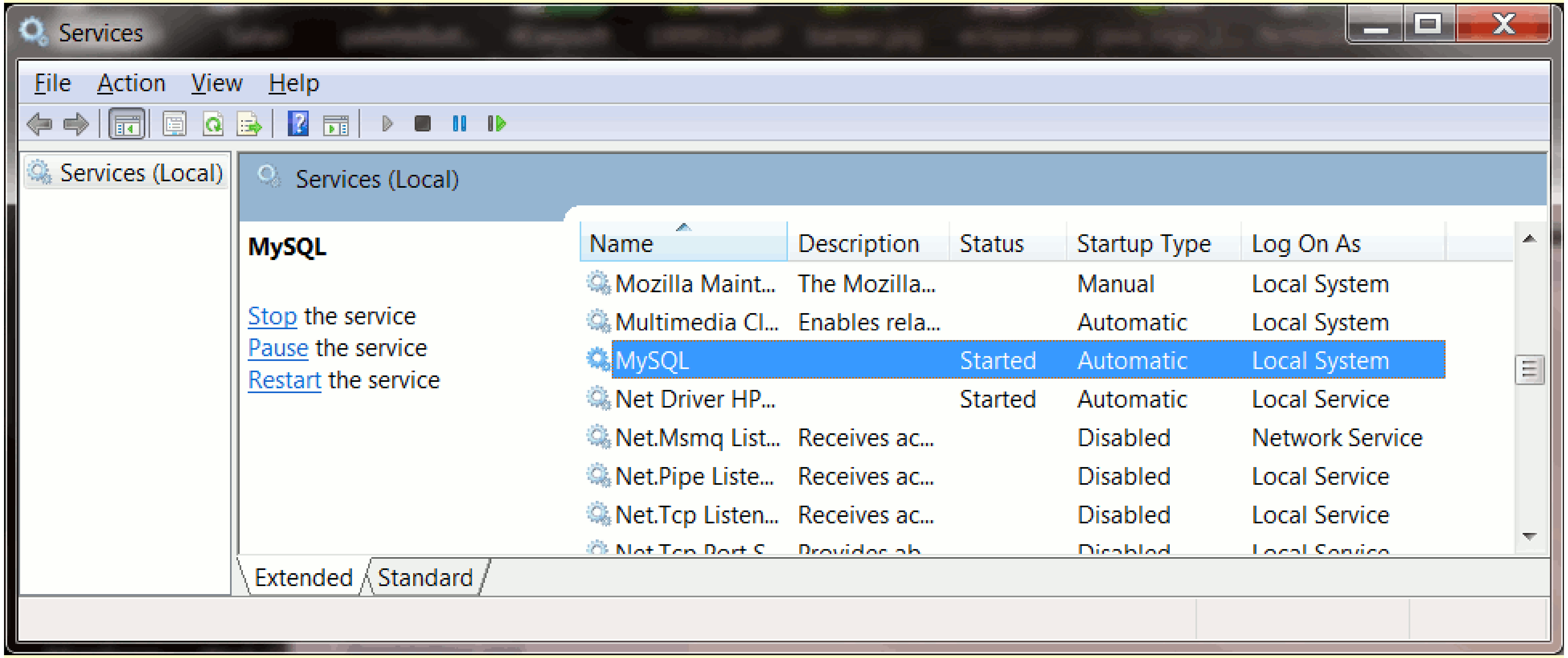
We want to run two processes – the server, and a client. The client is a piece of software which will connect to the server, into which we can enter commands, and see text responses from the server.

We can start the server with mysqld like this:



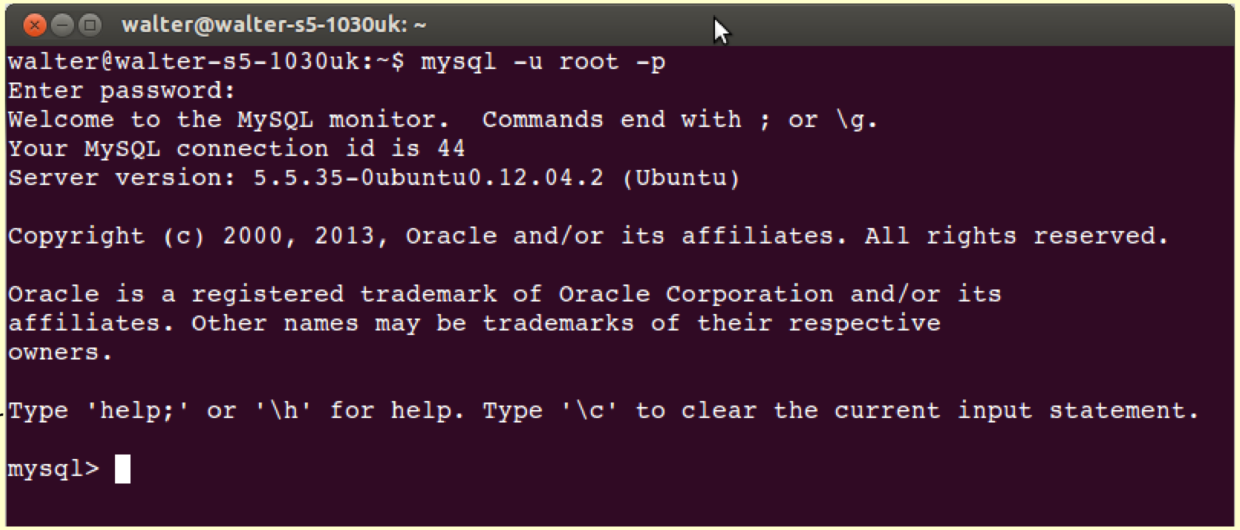
```
walter@walter-s5-1030uk:~$ sudo mysqld
[sudo] password for walter:
140209  9:15:30 [Warning] Using unique option prefix key_buffer instead of key_b
uffer_size is deprecated and will be removed in a future release. Please use the
 full name instead.
walter@walter-s5-1030uk:~$
```

Or the server can run as a service. Here its running as a service on Windows 7:



Name	Description	Status	Startup Type	Log On As
Mozilla Maint...	The Mozilla...		Manual	Local System
Multimedia Cl...	Enables rela...		Automatic	Local System
MySQL		Started	Automatic	Local System
Net Driver HP...		Started	Automatic	Local Service
Net.Msmq List...	Receives ac...		Disabled	Network Service
Net.Pipe Liste...	Receives ac...		Disabled	Local Service
Net.Tcp Listen...	Receives ac...		Disabled	Local Service

We can start a client with mysql:



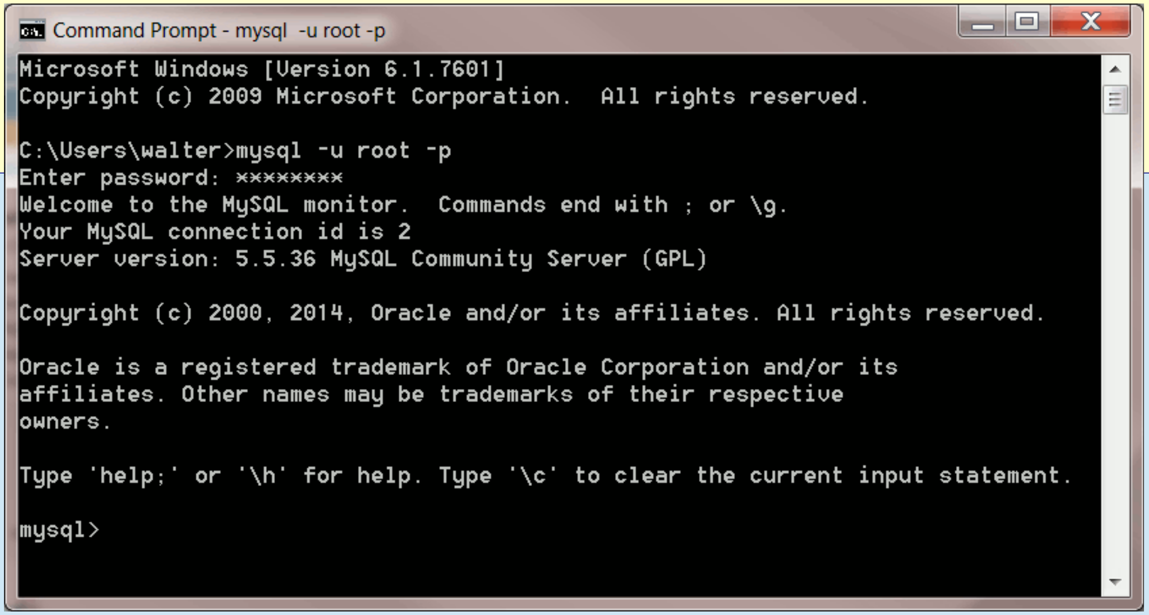
```
walter@walter-s5-1030uk:~$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 44
Server version: 5.5.35-0ubuntu0.12.04.2 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

We have used command line options -u root to connect as user 'root', and -p, so that we enter the password when the application starts. We have not specified which server we are connecting to, nor on which port, so we get the default values and connect to the server running on the local machine. Now we have a prompt

```
mysql>
```

within the client, at which we can enter commands which are sent to the server. Commands must end with a semi-colon ; - they can be multi-line for clarity.

Here is the client running on Windows 7:



```
Command Prompt - mysql  -u root -p
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\walter>mysql -u root -p
Enter password: ********
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.5.36 MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

We create a new empty database named libman and check what we have:

```
mysql> create database libman;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| libman             |
| mysql              |
| performance_schema |
| test               |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

Rather than being user 'root', we should create a user to use the libman database:

```
mysql> create user 'libuser'@'%';
Query OK, 0 rows affected (0.04 sec)
mysql> grant all on libman to libuser;
Query OK, 0 rows affected (0.02 sec)
mysql> set password for libuser = password('secret');
Query OK, 0 rows affected (0.01 sec)
```

Here we have created a user named libuser, allowed them to do anything to the database libman, and given them a secret password.

In practice the database administrator would create different users with different roles, and different access rights according to their roles, with better passwords than 'secret'.

We can then disconnect:

```
mysql> quit;
Bye
walter@walter-s5-1030uk:~$
```

and log on again as libuser using the libman database -

```
walter@walter-s5-1030uk:~$ mysql -u libuser --database=libman -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 51
Server version: 5.5.35-0ubuntu0.12.04.2 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

2.3 Access from a remote machine

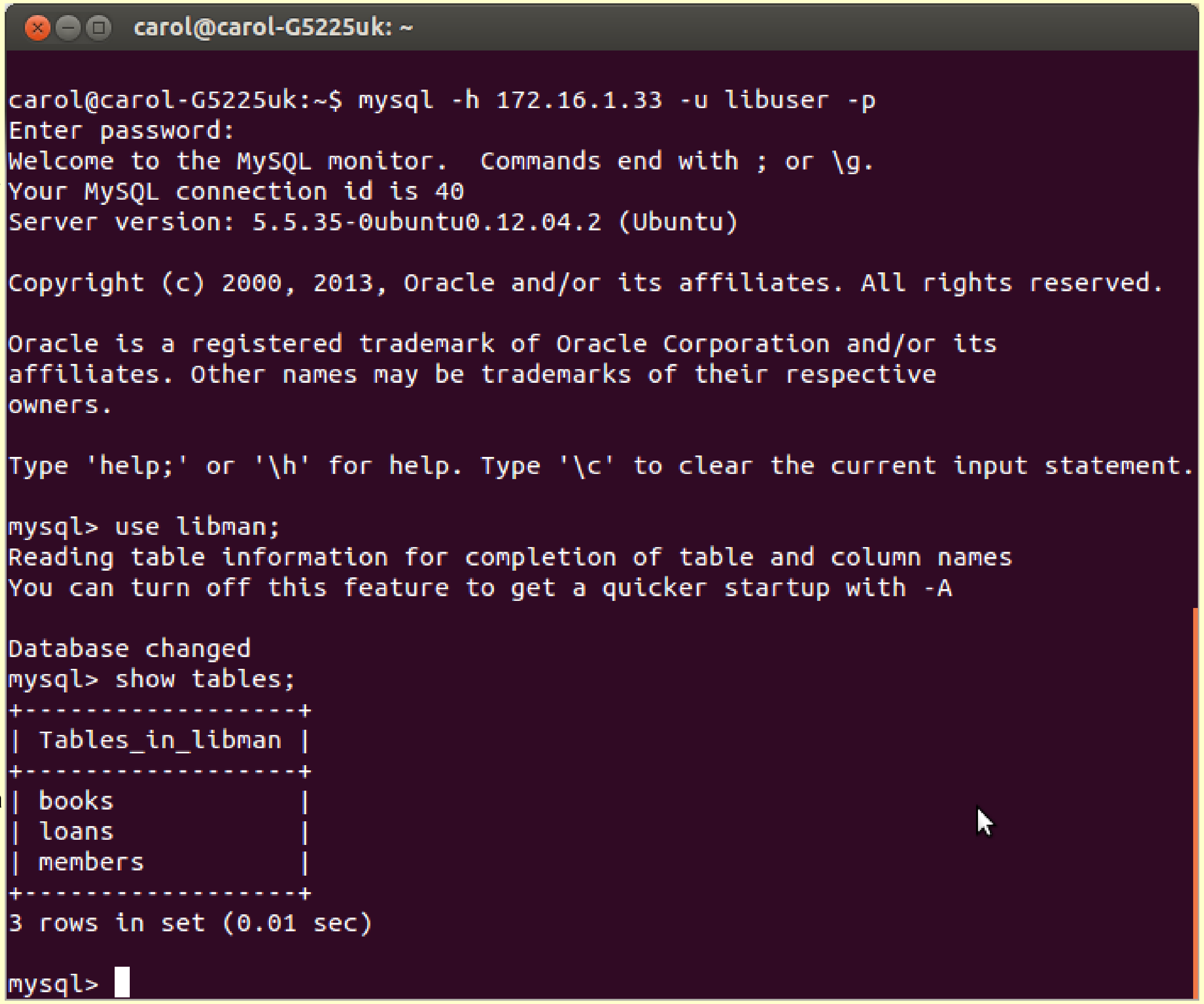
It is possible to run the client installed on one machine, and connect to a server running on another machine, if the configuration is correct. The server must be configured to accept connections from the appropriate machines – maybe any. This is probably set in the configuration my.cnf – check the manual for your version and OS. Also the connecting user must have a host name which is appropriate.

In this example, the server is running on a machine with IP 172.16.1.33, and we are logging in from another machine on the same LAN. (-h means specify host). The user is 'libman'@'%', the % meaning from any host. User libman has been granted all privileges on all databases.

This has obvious big advantages. In a real library, it means that several librarians can use the database at the same time, and that users (with different privilege levels) can search the information system for themselves from other computers.

On the other hand having more than one user having simultaneous access to the data raises potential problems. For example, what if two users reserve the same book at the same time? This is usually fixed by a locking process, in outline like the following

1. The record is locked
2. The book is reserved (or cancelled)
3. The lock is released



```
carol@carol-G5225uk:~$ mysql -h 172.16.1.33 -u libuser -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 40
Server version: 5.5.35-0ubuntu0.12.04.2 (Ubuntu)

Copyright (c) 2000, 2013, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use libman;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+------------------+
| Tables_in_libman |
+------------------+
| books            |
| loans            |
| members          |
+------------------+
3 rows in set (0.01 sec)

mysql> 
```

3 SQL

3.1 DDL and DML

SQL has two parts. DDL Is the ***data definition language***. This is used to specify what tables there are in a database, what columns are in each table and so on. It controls the structure of the data.

DML Is the ***data manipulation language***. With DML commands you can search tables for values, insert new ones, delete rows and change the data. The DML is used to manipulate the actual data, held in the structure defined by the DDL.

A fundamental DDL command is to create a table.

```
mysql> CREATE TABLE members (
    -> MemberID int AUTO_INCREMENT,
    -> FName varchar(30) NOT NULL,
    -> LName varchar(30) NOT NULL,
    -> DateOfBirth date NOT NULL,
    -> PRIMARY KEY (MemberID)
    -> );
```

This is creating a table named members, with 4 columns. The first column is named MemberID. Its data type is int, which is a 4 byte integer as in Java. The field is declared as AUTO_INCREMENT, which means the first row inserted will have the value 1, the second 2 and so on.

The next field is FName (first name). This is type varchar, which is a variable length string of characters, up to a maximum of 30. NOT NULL means that this field cannot be empty. Insertion of a row with an empty FName will fail.

LName (last name) is similar.

DateOfBirth is data type date. (There is also a datetime type which includes a time as well as a date)

The table will have MemberID as its primary key.

The books table is similar.

```
mysql> CREATE TABLE books (
    -> BookID int AUTO_INCREMENT,
    -> Title varchar(30) NOT NULL,
    -> Author varchar(30) NOT NULL,
    -> ISBN varchar(20),
    -> PRIMARY KEY (BookID)
    -> );
Query OK, 0 rows affected (0.10 sec)
```

and the loans table:

```
mysql> CREATE TABLE loans (
    -> LoanID int AUTO_INCREMENT,
    -> MemberID int,
    -> BookID int,
    -> DateOut date,
    -> DateBack date,
    -> PRIMARY KEY (LoanID)
    -> );
Query OK, 0 rows affected (0.10 sec)
```

Other DDL commands include altering, dropping (deleting), and renaming tables, and things called ***views***.

3.2 Inserting rows

We can insert new rows in a table using the INSERT INTO command. We need to say the name of the table, a list of the column headings, and a matching list of values. For example:

```
mysql> INSERT INTO members
    -> (FName, LName, DateOfBirth)
    -> VALUES ('John', 'Smith', '1980.01.01')
    -> ;
Query OK, 1 row affected (0.05 sec)
```

Check it worked (SELECT commands are discussed later):

```
mysql> select * from members;
+----------+-------+-------+-------------+
| MemberID | FName | LName | DateOfBirth |
+----------+-------+-------+-------------+
|        1 | John  | Smith | 1980-01-01  |
+----------+-------+-------+-------------+
1 row in set (0.00 sec)
```

We did not specify the MemberID field, since it is auto-increment.

There is another form of INSERT INTO where you only specify values, not column names, but in that case you must provide values for every column in the order they were defined.

3.3 Validity Constraints

Data is ***valid*** if it *could be correct*. For example, for a NumberOfPassengers field, 'yellow' is an invalid value, since its not an integer. The value 3 is valid (but may not be correct).

When data is entered (through a keyboard in a form) we might carry out some validity checks before writing it into a database. We can improve the data quality further by writing the constraints as part of the database definition.

We have already seen some. PRIMARY KEY ensures we will not have two rows with the same value. NOT NULL ensures the cell will not be empty (whether a cell could be empty or not depends on the situation).

Referential integrity is a type of validity constraint where we require values in one table to be present in another table. For example in the loans table, we have said MemberID is an int. But we want more than that – it must be equal to a member id present in the members table - as a foreign key. Similarly BookID in the loan table must match a BookID in the books table. We can do that using a DDL alter command:

```
mysql> ALTER TABLE loans
    -> ADD FOREIGN KEY (MemberID) REFERENCES members (MemberID);
Query OK, 0 rows affected (0.41 sec)
```

```
Records: 0  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE loans
    -> ADD FOREIGN KEY (BookID) REFERENCES books (BookID);
Query OK, 0 rows affected (0.22 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

We check it works. We insert a book into the library:

```
mysql> INSERT INTO books
    -> (Title, Author, ISBN)
    -> VALUES ('War and Peace', 'Leo Tolstoy', '123-333');
Query OK, 1 row affected (0.03 sec)
```

Then try to loan it:

```
mysql> INSERT INTO loans
    -> (MemberID, BookID, DateOut)
    -> VALUES (2,2,'2014-2-2');
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails (`libman`.`loans`, CONSTRAINT
`loans_ibfk_1` FOREIGN KEY (`MemberID`) REFERENCES `members` (`MemberID`))
```

but we cannot – there is no BookID 2 or member 2. Try a valid one:

```
mysql> INSERT INTO loans (MemberID, BookID, DateOut) VALUES (1,1,'2014-2-2');
Query OK, 1 row affected (0.04 sec)
```

3.4 SELECT fieldnames

We search and retrieve data from a database using a SELECT SQL command, which is therefore very common, and flexible. We can first choose which columns to fetch. If we say *, this means all fields. For example:

```
mysql> SELECT * FROM members;
+----------+---------+--------+-------------+
| MemberID | FName   | LName  | DateOfBirth |
+----------+---------+--------+------------+
|        1 | John    | Smith  | 1980-01-01 |
|        2 | June    | Miller | 1910-02-01 |
|        3 | Max     | Land   | 1990-02-01 |
|        4 | Leonard | Zimmer | 1999-02-01 |
+----------+---------+--------+------------+
4 rows in set (0.00 sec)
```

Or we can pick out the ones we want:

```
mysql> SELECT FName, LName FROM members;
+---------+--------+
| FName   | LName  |
+---------+--------+
| John    | Smith  |
| June    | Miller |
| Max     | Land   |
| Leonard | Zimmer |
+---------+--------+
4 rows in set (0.00 sec)
```

The order of the fields might differ from the original definition of the table:

```
mysql> SELECT DateOfBirth, FName FROM members;
+-------------+---------+
| DateOfBirth | FName   |
+-------------+---------+
| 1980-01-01  | John    |
| 1910-02-01  | June    |
| 1990-02-01  | Max     |
| 1999-02-01  | Leonard |
+-------------+---------+
4 rows in set (0.00 sec)
```

The columns in a RDBMS table have no order. There is no first column or second column. Only when the data is fetched and displayed is there any left-right order to the columns.

3.5 SELECT rows by WHERE

We can control which rows are fetched by using a WHERE clause. For example

```
mysql> SELECT * FROM members WHERE MemberID>2;
+----------+---------+--------+-------------+
| MemberID | FName   | LName  | DateOfBirth |
+----------+---------+--------+-------------+
|        3 | Max     | Land   | 1990-02-01  |
|        4 | Leonard | Zimmer | 1999-02-01  |
+----------+---------+--------+-------------+
2 rows in set (0.01 sec)
```

The WHERE clause can be compound, such as

```
mysql> SELECT * FROM members WHERE MemberID>1 AND DateOfBirth<'1950-01-01';
+----------+-------+--------+-------------+
| MemberID | FName | LName  | DateOfBirth |
+----------+-------+--------+-------------+
|        2 | June  | Miller | 1910-02-01  |
+----------+-------+--------+-------------+
1 row in set (0.00 sec)
```

Since the primary key identifies the row, we will often fetch a single row using it:

```
mysql> SELECT * FROM members WHERE MemberID=3;
+----------+-------+-------+-------------+
| MemberID | FName | LName | DateOfBirth |
+----------+-------+-------+-------------+
|        3 | Max   | Land  | 1990-02-01  |
+----------+-------+-------+-------------+
1 row in set (0.02 sec)
```

A SQL engine can also do pattern matching. For example, suppose we want the people whose first name starts with a J:

```
 mysql> select * from members where FName like 'J%';
+----------+-------+--------+-------------+
| MemberID | FName | LName  | DateOfBirth |
+----------+-------+--------+-------------+
|        1 | John  | Smith  | 1980-01-01  |
|        2 | June  | Miller | 1910-02-01  |
|        5 | Jake  | Smith  | 1980-01-01  |
+----------+-------+--------+-------------+
3 rows in set (0.00 sec)
```

A % is a ***wildcard*** means any sequence of characters. A _ means any single character.

3.6   Order rows

We can control the order of the rows fetched by an 'order by' clause. For example if the library members are:

```
mysql> select * from members;
+----------+---------+--------+-------------+
| MemberID | FName   | LName  | DateOfBirth |
+----------+---------+--------+-------------+
|        1 | John    | Smith  | 1980-01-01  |
|        2 | June    | Miller | 1910-02-01  |
|        3 | Max     | Land   | 1990-02-01  |
|        4 | Leonard | Zimmer | 1999-02-01  |
|        5 | Jake    | Smith  | 1980-01-01  |
+----------+---------+--------+-------------+
5 rows in set (0.00 sec)
```

we can get them in age order by:

```
mysql> select * from members order by DateOfBirth;
```

```
+----------+---------+--------+-------------+
| MemberID | FName   | LName  | DateOfBirth |
+----------+---------+--------+-------------+
|        2 | June    | Miller | 1910-02-01  |
|        1 | John    | Smith  | 1980-01-01  |
|        5 | Jake    | Smith  | 1980-01-01  |
|        3 | Max     | Land   | 1990-02-01  |
|        4 | Leonard | Zimmer | 1999-02-01  |
+----------+---------+--------+-------------+
5 rows in set (0.00 sec)
```

This defaults to ascending order, so it is the same as

```
mysql> select * from members order by DateOfBirth asc;
```

but we can have descending:

```
mysql> select * from members order by DateOfBirth desc;
+----------+---------+--------+-------------+
| MemberID | FName   | LName  | DateOfBirth |
+----------+---------+--------+-------------+
|        4 | Leonard | Zimmer | 1999-02-01  |
|        3 | Max     | Land   | 1990-02-01  |
|        1 | John    | Smith  | 1980-01-01  |
|        5 | Jake    | Smith  | 1980-01-01  |
|        2 | June    | Miller | 1910-02-01  |
+----------+---------+--------+-------------+
5 rows in set (0.00 sec)
```

and we can order on more than one field:

```
mysql> select * from members order by LName, FName;
+----------+---------+--------+-------------+
| MemberID | FName   | LName  | DateOfBirth |
+----------+---------+--------+-------------+
|        3 | Max     | Land   | 1990-02-01  |
|        2 | June    | Miller | 1910-02-01  |
|        5 | Jake    | Smith  | 1980-01-01  |
|        1 | John    | Smith  | 1980-01-01  |
|        4 | Leonard | Zimmer | 1999-02-01  |
+----------+---------+--------+-------------+
5 rows in set (0.00 sec)
```

so this orders the rows by LName, and if there are two rows with the same LName, puts them into order of FName.

3.7 DISTINCT to avoid duplicates

Some queries may return repeated rows, such as

```
 mysql> select LName from members;
+--------+
| LName  |
+--------+
| Smith  |
| Miller |
| Land   |
| Zimmer |
| Smith  |
+--------+
5 rows in set (0.00 sec)
```

We can remove duplicates with DISTINCT:

```
mysql> select distinct LName from members;
+--------+
| LName  |
+--------+
| Smith  |
| Miller |
| Land   |
| Zimmer |
+--------+
4 rows in set (0.00 sec)
```

3.8 SQL functions

A SQL database engine can 'work out' various functions.

For example COUNT can count things

```
mysql> select count(MemberID) from members;
+-----------------+
| count(MemberID) |
+-----------------+
|               5 |
+-----------------+
1 row in set (0.00 sec)
```

A SELECT returns a new (temporary) table. We can set the column names using AS:

```
 mysql> select count(MemberID) as RowCount from members;
+----------+
| RowCount |
+----------+
|        5 |
+----------+
1 row in set (0.00 sec)
```

You would often combine this with a WHERE:

```
mysql> select count(*) from members where DateOfBirth>'1970-01-01';
+----------+
| count(*) |
+----------+
|        4 |
+----------+
1 row in set (0.00 sec)
```

or counting distinct values:

```
mysql> select count(distinct lname)  from members;
+-----------------------+
| count(distinct lname) |
+-----------------------+
|                     4 |
+-----------------------+
1 row in set (0.00 sec)
```

The min function finds the minimum value. For example, to find the oldest member.

```
mysql> select min(DateOfBirth), FName, LName  from members;
+------------------+-------+-------+
| min(DateOfBirth) | FName | LName |
+------------------+-------+-------+
| 1910-02-01       | John  | Smith |
+------------------+-------+-------+
1 row in set (0.00 sec)
```

Similarly max finds the largest.

Maybe we want the youngest person whose first name starts with J:

```
mysql> select max(DateOfBirth), FName, LName  from members where FName like 'J%';
+------------------+-------+-------+
| max(DateOfBirth) | FName | LName |
+------------------+-------+-------+
| 1980-01-01       | John  | Smith |
+------------------+-------+-------+
1 row in set (0.00 sec)
```

3.9 Joining tables

A JOIN returns fields from two tables linked by a common field. This will usually be the case for a normalised database.

In our library database, a table called loans tells us what loans have been made (and maybe returned).

Suppose the loans table is

```
mysql> select * from loans;
+--------+----------+--------+------------+----------+
| LoanID | MemberID | BookID | DateOut    | DateBack |
+--------+----------+--------+------------+----------+
|      2 |        1 |      1 | 2014-02-02 | NULL     |
|      3 |        3 |      2 | 2014-02-03 | NULL     |
|      4 |        4 |      3 | 2014-02-04 | NULL     |
|      5 |        5 |      4 | 2014-02-04 | NULL     |
+--------+----------+--------+------------+----------+
4 rows in set (0.00 sec)
```

How do we know which member has which book? By joining these table on MemberID:

```
mysql> select * from
    -> members M
    -> inner join
    -> loans L
    -> on M.MemberID = L.MemberID;
+----------+---------+--------+-------------+--------+----------+--------+------------+----------+
| MemberID | FName   | LName  | DateOfBirth | LoanID | MemberID | BookID | DateOut    | DateBack |
+----------+---------+--------+-------------+--------+----------+--------+------------+----------+
|        1 | John    | Smith  | 1980-01-01  |      2 |         1 |      1 | 2014-02-02 | NULL     |
|        3 | Max     | Land   | 1990-02-01  |      3 |         3 |      2 | 2014-02-03 | NULL     |
|        4 | Leonard | Zimmer | 1999-02-01  |      4 |         4 |      3 | 2014-02-04 | NULL     |
|        5 | Jake    | Smith  | 1980-01-01  |      5 |         5 |      4 | 2014-02-04 | NULL     |
+----------+---------+--------+------------+--------+----------+--------+------------+----------+
4 rows in set (0.00 sec)
```

we give the tables members and loans aliases of M and L, so we can quickly refer to M.MemberID in the members table, and L.MemberID in the loans table.

We probably just want to know the person and the book, so

```
mysql> select FName, LName, BookID from
    -> members M
    -> inner join
    -> loans L
    -> on M.MemberID = L.MemberID;
+---------+--------+--------+
| FName   | LName  | BookID |
+---------+--------+--------+
| John    | Smith  |      1 |
| Max     | Land   |      2 |
| Leonard | Zimmer |      3 |
| Jake    | Smith  |      4 |
+---------+--------+--------+
4 rows in set (0.00 sec)
```

But we want the book title, not its ID. So we need to take this, and join it with the books table on the BookID field:

```
mysql> select FName, LName, Title from
    -> members M
    -> inner join
    -> loans L
    -> on M.MemberID = L.MemberID
    -> inner join
    -> books B
    -> on L.BookID=B.BookID;
```

```
+---------+--------+------------------+
| FName   | LName  | Title            |
+---------+--------+------------------+
| John    | Smith  | War and Peace    |
| Max     | Land   | Keeping Chickens |
| Leonard | Zimmer | XML for Fun      |
| Jake    | Smith  | Learning JDBC    |
+---------+--------+------------------+
4 rows in set (0.00 sec)
```

3.10 Delete rows

This has the form

delete from table where conditions;

For example if we want to delete the member with id 2:

```
mysql> delete from members where MemberID=2;
Query OK, 1 row affected (0.04 sec)
```

There is no verification check and no way to undelete a row, so you need to be *very careful* with this.

An attempt to delete a row which would produce a referential integrity would fail. For example, if we tried to delete a member for whom there were loan records, it would fail.

```
mysql> delete from members where MemberID=1;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`libman`.`loans`,
CONSTRAINT `loans_ibfk_1` FOREIGN KEY (`MemberID`) REFERENCES `members` (`MemberID`))
```

In terms of information system design, it is not a good idea to delete data. It cost money to collect and is always useful.

For example, we should have a JoinDate and a LeaveDate field in the members table. If a member leaves the library, we just fill in the LeaveDate.

In the UK the Data Protection Act might oblige you to delete some data.

3.11 Update rows

The SQL update command changes values in existing rows:

update table

set column=value, column=value ..

where ...;

For example, if the loans table is

```
mysql> select * from loans;
+--------+----------+--------+------------+------------+
| LoanID | MemberID | BookID | DateOut    | DateBack   |
+--------+----------+--------+------------+------------+
|      2 |        1 |      1 | 2014-02-02 | 2014-02-13 |
|      3 |        3 |      2 | 2014-02-03 | NULL       |
|      4 |        4 |      3 | 2014-02-04 | NULL       |
|      5 |        5 |      4 | 2014-02-04 | NULL       |
+--------+----------+--------+------------+------------+
4 rows in set (0.00 sec)
```

and book 2 is returned on 2014-02-10, we can say:

```
mysql> update loans
    -> set DateBack='2014-02-10'
    -> where LoanID=3;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> select * from loans;
+--------+----------+--------+------------+------------+
| LoanID | MemberID | BookID | DateOut    | DateBack   |
+--------+----------+--------+------------+------------+
|      2 |        1 |      1 | 2014-02-02 | 2014-02-13 |
|      3 |        3 |      2 | 2014-02-03 | 2014-02-10 |
|      4 |        4 |      3 | 2014-02-04 | NULL       |
|      5 |        5 |      4 | 2014-02-04 | NULL       |
+--------+----------+--------+------------+------------+
4 rows in set (0.00 sec)
```

Note that without a 'where' clause, *all rows are updated*.

4 JDBC

4.1 Getting connected

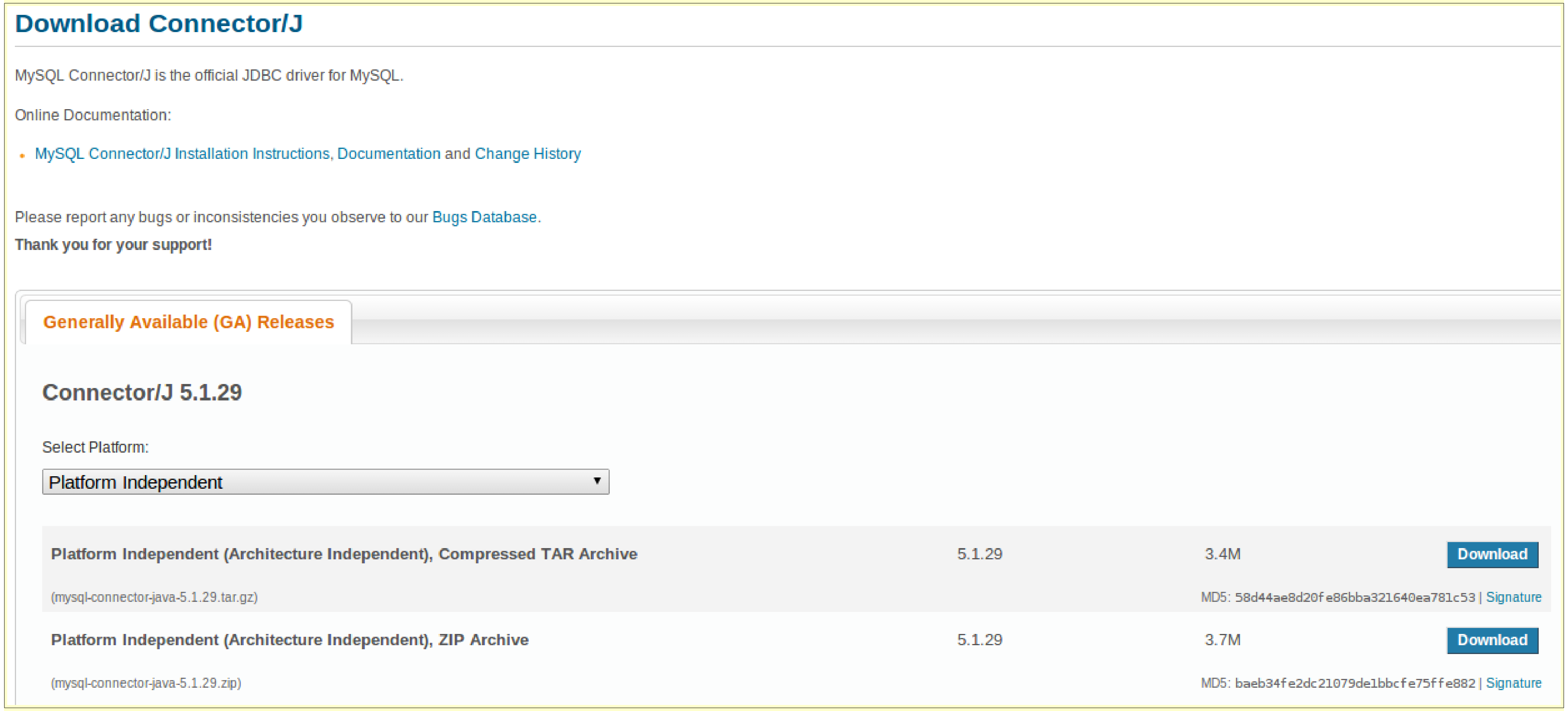
This section is about JDBC, the framework for using database through a Java application.

Usually an interaction with a database will be like this:

1. Get relevant data from the user
2. Connect to the database
3. Do the database transaction
4. Output results to the user
5. Close the connections

4.2 Downloading a driver

This requires a suitable Java database driver, which will be specific to the server in use. For MySQL this is Connector/J, which we can Google and download:



Download Connector/J

MySQL Connector/J is the official JDBC driver for MySQL.

Online Documentation:

- MySQL Connector/J Installation Instructions, Documentation and Change History

Please report any bugs or inconsistencies you observe to our Bugs Database.

Thank you for your support!

Generally Available (GA) Releases

Connector/J 5.1.29

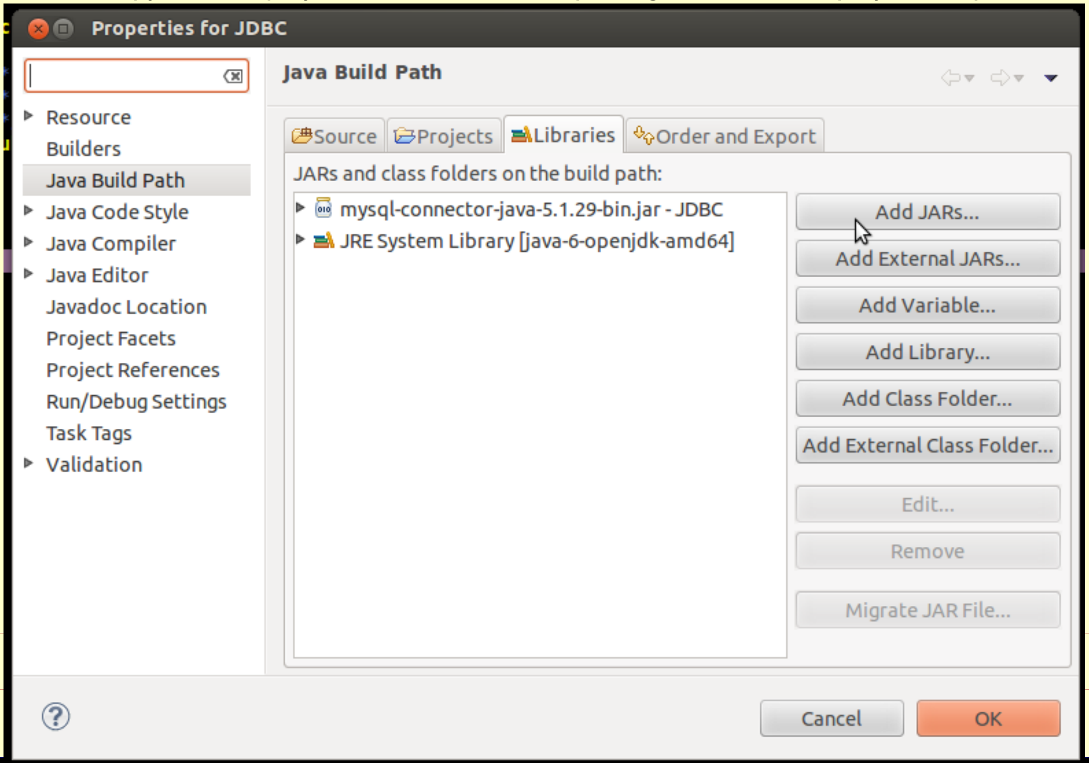
Select Platform:

Platform Independent

Platform Independent (Architecture Independent), Compressed TAR Archive	5.1.29	3.4M	Download
(mysql-connector-java-5.1.29.tar.gz)			MD5: 58d44ae8d20fe86bba321640ea781c53 | Signature
Platform Independent (Architecture Independent), ZIP Archive	5.1.29	3.7M	Download
(mysql-connector-java-5.1.29.zip)			MD5: baeb34fe2dc21079de1bbcfe75ffe882 | Signature

In this case we get a zipped file. Unzip it and we get a jar.

We must then include it in the classpath of the project. You might want to copy it to the project folder. Then in Eclipse, right-click on the project, Properties, Build path, Libraries and add JAR:



JARs and class folders on the build path:

mysql-connector-java-5.1.29-bin.jar - JDBC
JRE System Library [java-6-openjdk-amd64]

4.3 Connection strings

We can connect to the server like this:

```
	public static void main(String[] args) {
		Connection conn = null;

		try {
			// connect
			conn = DriverManager.getConnection("jdbc:mysql://172.16.1.33:3306", "libuser", "secret");
			// choose database
			Statement stmt = conn.createStatement();
			stmt.execute("use libman");
			// do some transactions here
```

```
                  // ..
                  conn.close();
            } catch (SQLException ex) {
                  // handle any errors
                  System.out.println("SQLException: " + ex.getMessage());
                  System.out.println("SQLState: " + ex.getSQLState());
                  System.out.println("VendorError: " + ex.getErrorCode());
            }

      }
```

In

```
conn = DriverManager.getConnection("jdbc:mysql://172.16.1.33:3306", "libuser", "secret");
```

the user name is libuser, and the password is 'secret'. The string "jdbc:mysql://172.16.1.33:3306" is called the connection string, and will vary for different servers and drivers. This version is for MySQL and the Connector/J driver. 172.16.1.33 is the IP address of where the server is, and 3306 is the port it is listening on (3306 is the default). For this to work, check

1. The server is configured correctly in my.cnf to accept incoming connections
2. User name and password is correct
3. The user has appropriate privileges.
4. The user's hostname in the user's table is appropriate.

In this case, we have

```
mysql> SELECT User, host FROM mysql.user;
+------------------+------------------+
| User             | host             |
+------------------+------------------+
| libuser          | %                |
| root             | 127.0.0.1        |
| root             | ::1              |
| debian-sys-maint | localhost        |
| root             | localhost        |
|                   | walter-s5-1030uk |
| root              | walter-s5-1030uk |
+-------------------+------------------+
7 rows in set (0.00 sec)
```

so that libuser is allowed to log on from anywhere (host is wildcard %).

4.4 Using a ResultSet

A SQL select query will return a ResultSet instance. For example:

```
            Connection conn = null;
            try {
                  // connect
                  conn = DriverManager.getConnection("jdbc:mysql://172.16.1.33:3306", "libuser", "secret");
                  // choose database
                  Statement stmt = conn.createStatement();
                  stmt.execute("use libman");
                  ResultSet rs = stmt.executeQuery("select * from members");
                  while (rs.next())
                  {
                        System.out.print(rs.getInt("MemberID"));
                        System.out.print(" "+rs.getString("FName"));
                        System.out.print(" "+rs.getString("LName"));
                        System.out.println(" "+rs.getDate("DateOfBirth"));
                  }
                  conn.close();
            } catch (SQLException ex) {
                  // handle any errors
                  System.out.println("SQLException: " + ex.getMessage());
                  System.out.println("SQLState: " + ex.getSQLState());
                  System.out.println("VendorError: " + ex.getErrorCode());
            }
```

which outputs:

```
1 John Smith 1980-01-01
3 Max Land 1990-02-01
4 Leonard Zimmer 1999-02-01
5 Jake Smith 1980-01-01
```

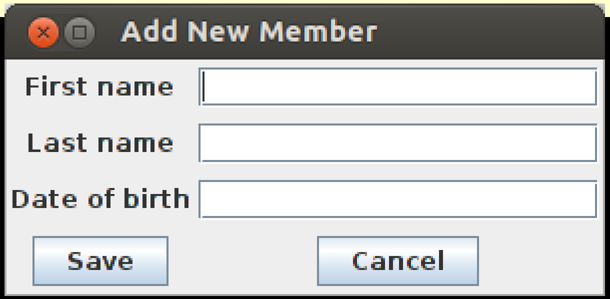
4.5 JDBC and Swing GUI – insert new record

Usually the user interface will be a GUI, perhaps Swing.

For example we might add a new library member by having a JDialog through which the user enters data, then an SQL insert query is executed.

Code to create a non-functioning dialog might be:

```
public class NewMember extends JDialog {
      private JTextField fnLabel;
      .. similarly

      NewMember() {

            setVisible(true);
            setTitle("Add New Member");
            setVisible(true);
            setLayout(new GridBagLayout());
            GridBagConstraints gbc = new GridBagConstraints();
            gbc.insets = new Insets(4, 2, 4, 2);
            // first name
            JLabel fnLabel = new JLabel("First name");
            gbc.gridx = 0;
            gbc.gridy = 0;
            add(fnLabel, gbc);
            fName = new JTextField();
            fName.setPreferredSize(new Dimension(200, 20));
            gbc.gridx = 1;
            add(fName, gbc);
            // and so on
```



```
       ..
            // buttons
            ok = new JButton("Save");
            gbc.gridx = 0;
            gbc.gridy = 3;
            add(ok, gbc);
            cancel = new JButton("Cancel");
            gbc.gridx = 1;
            gbc.gridy = 3;
            add(cancel, gbc);
            pack();
      }

}
```

To make it work, we can first declare the class as implementing ActionListener.

```
public class NewMember extends JDialog implements ActionListener {
```

Telling the buttons that this will listen to them:

```
            ok = new JButton("Save");
            ok.addActionListener(this);
            ..
            cancel = new JButton("Cancel");
            cancel.addActionListener(this);
```

and having an actionPerformed method:

```
      @Override
      public void actionPerformed(ActionEvent arg0) {
            Object src = arg0.getSource();

      }
```

To actually insert a new record, we need to:

1. Connect to the database
2. Get the values from the input fields, and construct the SQL insert statement
3. Execute it

Like this:

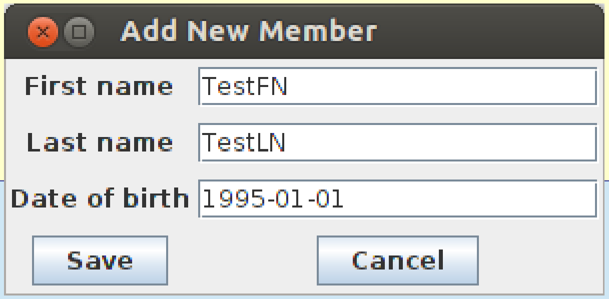
```
@Override
      public void actionPerformed(ActionEvent arg0) {
            Object src = arg0.getSource();
            if (src==ok)
            {
                  Connection conn = null;
                  try {
                        // connect
                        conn = DriverManager.getConnection("jdbc:mysql://172.16.1.33:3306", "libuser", "secret");
                        // choose database
                        Statement stmt = conn.createStatement();
                        stmt.execute("use libman");
                        String insert = "insert into members (FName,LName, DateOfBirth)";
                        insert +="values ('"+fName.getText()+"','"+lName.getText()+"','"+dob.getText()+"')";
                        stmt.execute(insert);
                        conn.close();
                        dispose();
                  } catch (SQLException ex) {
                        // handle any errors
                        System.out.println("SQLException: " + ex.getMessage());
                        System.out.println("SQLState: " + ex.getSQLState());
                        System.out.println("VendorError: " + ex.getErrorCode());
                  }
            }
      }
```

The tricky part is to construct the SQL string correctly. For each of the values of fname, lname and dob, we need to surround with quotes and separate with commas, which is what

```
String insert = "insert into members (FName,LName, DateOfBirth)";
insert +="values ('"+fName.getText()+"','"+lName.getText()+"','"+dob.getText()+"')";
```

is doing. Check this very carefully. You must ensure the result is what would work correctly if it were issued directly.

So we can run this:



First name	TestFN
Last name	TestLN
Date of birth	1995-01-01

and check it works:

```
mysql> select * from members;
+----------+---------+--------+-------------+
| MemberID | FName   | LName  | DateOfBirth |
+----------+---------+--------+-------------+
|        1 | John    | Smith  | 1980-01-01  |
|        3 | Max     | Land   | 1990-02-01  |
|        4 | Leonard | Zimmer | 1999-02-01  |
|        5 | Jake    | Smith  | 1980-01-01  |
|        6 | TestFN  | TestLN | 1995-01-01  |
+----------+---------+--------+-------------+
5 rows in set (0.00 sec)
```

This is just a sketch outline. There are several issues here:

1. sop'ing the exceptions is not much use. A JoptionPane message would be appropriate.

2. After the query, the dialog box is just disposed. A confirmation message to the user is needed.

3. Three buttons would be better – 'Save and close', 'Save and enter another' and 'Cancel'.

4. The biggest issue is the format of the date. The user needs to be told if it is year day month or year month day. And the inputted value must be validated before an attempt is made to write into the database.

4.6 Swing Tables

To display a set of rows, a Swing JTable is the obvious component.

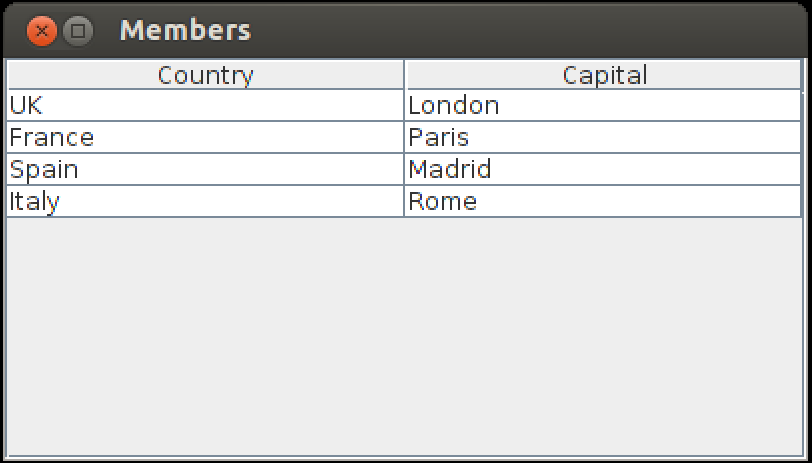
But JTables are complex. Additionally it is not usually a good design to display an entire database table – which might contain millions of rows. We are using the server to search and process rows, an dnot the user. If we expect the user to browse through a million records, it won't work.

With that qualification, here we go. An html table is very simple. A JTable is not. It uses a modified MVC design pattern, which separates handling the data (the model), how it appears (the view), and the control of the data. To achieve this several other classes are involved. In particular, despite how it appears, the data is not actually 'in' the JTable – it is in a separate 'model'.

We will present this in 3 stages – a default version with no explicit model, then 1 with a separate model, then 1 with the data coming from a database table.

First version is like this:

```
public class MembersTable extends JDialog {

    private JTable table;
    public Object[][] data = {
        {"UK", "London"},
        {"France", "Paris"},
        {"Spain", "Madrid"},
        {"Italy", "Rome"}
    };

    MembersTable(JFrame owner, String title, boolean modal) {
        super(owner, title, modal);
        setBounds(50, 50, 400, 200);
        initComponents();
        setVisible(true);
    }

    private void initComponents() {
        Object[] columns = {"Country", "Capital"};
        table = new JTable(data, columns);
        JScrollPane sp = new JScrollPane(table);
```



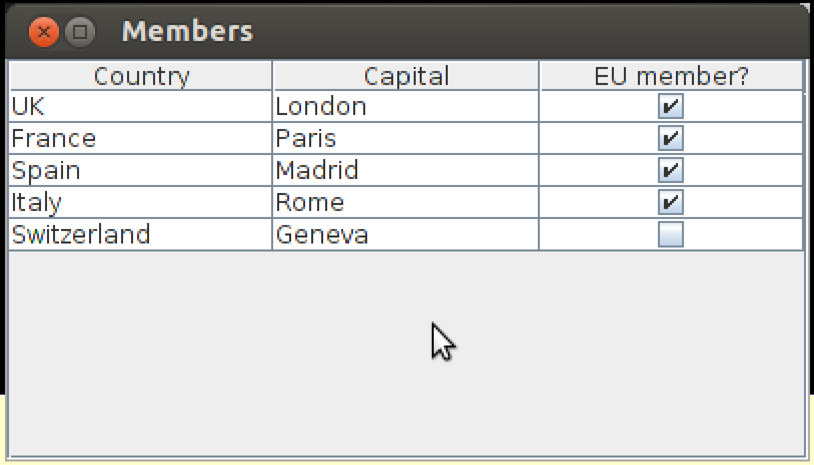
Country	Capital
UK	London
France	Paris
Spain	Madrid
Italy	Rome

```
        add(sp);
        sp.setBounds(0, 0, 400, 400);
    }
}
```

Here the column headings are an array of Objects (actually Strings) and the data is a 2D array of objects. We create the table using these as parameters, and display it inside a scroll pane (which is normal and convenient. If you don't use a scrollpane, you must display the headings and the data separately).

Thsi default version is pretty limited – for example you cannot have different data types in different columns. In our second version, we sub-class AbstractTableModel and use it as a separate data model:

```
class MyModel extends AbstractTableModel {
    private String[] columnNames = {"Country", "Capital", "EU member?"};
    private Object[][] data = {
        {"UK", "London"},
        {"France", "Paris"},
        {"Spain", "Madrid"},
        {"Italy", "Rome"},
        {"Switzerland", "Geneva"}
    };
    private Boolean[] endCol = {true, true, true, true, false};

    public int getColumnCount() {
        return columnNames.length;
    }

    public int getRowCount() {
        return data.length;
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public Object getValueAt(int row, int col) {
```


Members

Country	Capital	EU member?
UK	London	✓
France	Paris	✓
Spain	Madrid	✓
Italy	Rome	✓
Switzerland	Geneva	☐

```
        if (col == 2) {
            return endCol[row];
        }
        return data[row][col];
    }

    public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }

    public boolean isCellEditable(int row, int col) {
        if (col == 2) {
            return true;
        } else {
            return false;
        }

    }

    public void setValueAt(Object value, int row, int col) {
        if (col == 2) {
            endCol[row] = (Boolean) value;
        } else {
            data[row][col] = value;
        }
        fireTableCellUpdated(row, col);
    }
}
```

Then we create the JTable by:

```
      MyModel data = new MyModel();
     table = new JTable(data);
```

In this form, we can have a different data type in column 2 (Boolean), and we make column 2 editable, and the others not. The methods we are implementing from AbstractTableModel pretty much make sense. Most of these are call-back methods – that is, the JTable will call them when needed. For example when it needs to know what type a column is, it calls getColumnClass (and it needs to know this because different types can be darwn in different ways. For example a Boolean is drawn as a checkbox). When the JTable needs to knw the actual value in a cell, it calls getValueAt.

The final stage is to have the data read from the database into the model of the table. A 2D array is unsuitable for this, because we do not know how many rows there will be. One option would be to have an ArrayList of String arrays, with each list element being a String array of column values corresponding to a row.

This means we have:

```
class MyModel extends AbstractTableModel {
    private String[] columnNames = {"ID", "First name", "Last Name", "Date of birth"};
    private ArrayList<String[]> data = new ArrayList<String[]>();
    private void  fetchData()
    {
      Connection conn = null;
            try {
                  // connect
                  conn = DriverManager.getConnection("jdbc:mysql://172.16.1.33:3306", "libuser", "secret");
                  // choose database
                  Statement stmt = conn.createStatement();
                  stmt.execute("use libman");
                  ResultSet rs = stmt.executeQuery("select * from members");
                  while (rs.next())
                  {
                        String id = ""+rs.getInt("MemberID");
                        String fn=rs.getString("FName");
                        String ln = rs.getString("LName");
                        String db=""+rs.getDate("DateOfBirth");
                        String[] row = {id, fn, ln,db};
                        data.add(row);
                  }
                  conn.close();
            } catch (SQLException ex) {
                  // handle any errors
                  System.out.println("SQLException: " + ex.getMessage());
                  System.out.println("SQLState: " + ex.getSQLState());
                  System.out.println("VendorError: " + ex.getErrorCode());
```

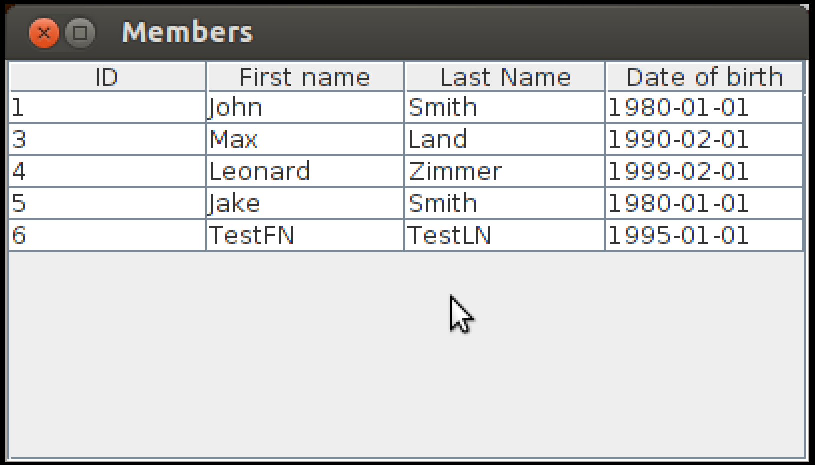
```
        }
    }
```

and we call fetchData in the constructor:

```
    MyModel()
    {
      fetchData();
    }
```

We make appropriate changes to the model methods:

```
    public int getColumnCount() {
        return columnNames.length;
    }

    public int getRowCount() {
        return data.size();
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public Object getValueAt(int row, int col) {
        return data.get(row)[col];
    }

    public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }

    public boolean isCellEditable(int row, int col) {
        return false;
    }
```



ID	First name	Last Name	Date of birth
1	John	Smith	1980-01-01
3	Max	Land	1990-02-01
4	Leonard	Zimmer	1999-02-01
5	Jake	Smith	1980-01-01
6	TestFN	TestLN	1995-01-01

```
    public void setValueAt(Object value, int row, int col) {
        String[] r = data.get(row);
        r[col]=(String)value;
        data.set(row, r);
        fireTableCellUpdated(row, col);
    }
```

We would probably want to allow the user to edit the data in the JTable, and save the result back into the database. They should be able to edit all columns except the ID, which cannot change:

```
      public boolean isCellEditable(int row, int col) {
            if (col == 0)
                  return false;
            else
                  return true;
      }
```

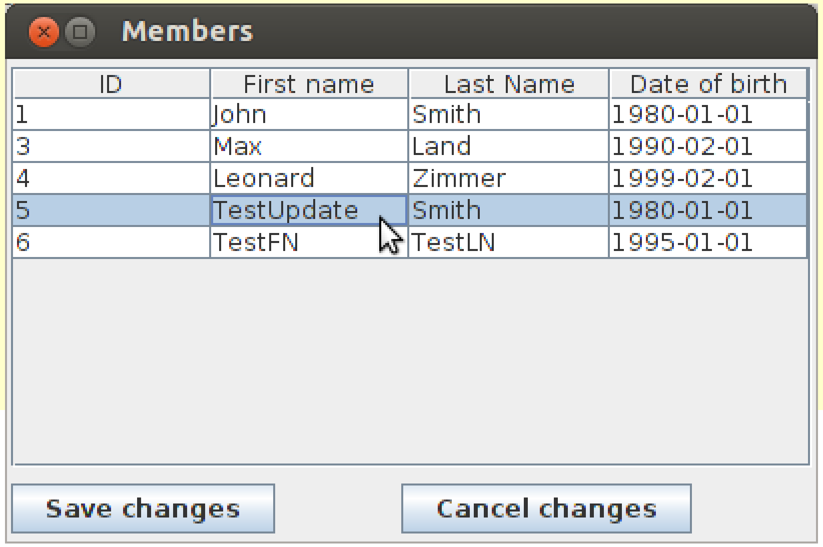
We add buttons to the JFrame allowing the user to save changes, or cancel them, and add a method saveData to the model. The OK button does:

```
      @Override
      public void actionPerformed(ActionEvent e) {
            Object src = e.getSource();
            if (src == ok) {
                  data.saveData();
            }

      }
```

and saveData iterates through the data in the model, and does a SQL update command for each row:

```
      void saveData() {
            Connection conn = null;
            try {
                  // connect
                  conn = DriverManager.getConnection("jdbc:mysql://172.16.1.33:3306",
                              "libuser", "secret");
                  // choose database
                  Statement stmt = conn.createStatement();
                  stmt.execute("use libman");
                  for (String[] row : data) {
                        String sql = "update members ";
                        sql += "set fname='" + row[1] + "', lname='" + row[2]
                                    + "', DateOfBirth='" + row[3] + "'";
                        sql += " where MemberID = " + row[0];
                        System.out.println(sql);
                        stmt.execute(sql);
                  }
                  conn.close();
            }
            catch (SQLException ex) {
                  // handle any errors
                  System.out.println("SQLException: " + ex.getMessage());
                  System.out.println("SQLState: " + ex.getSQLState());
                  System.out.println("VendorError: " + ex.getErrorCode());
            }
      }
```

which we test as:



ID	First name	Last Name	Date of birth
1	John	Smith	1980-01-01
3	Max	Land	1990-02-01
4	Leonard	Zimmer	1999-02-01
5	TestUpdate	Smith	1980-01-01
6	TestFN	TestLN	1995-01-01

giving:

```
mysql> select * from members;
+----------+------------+--------+-------------+
| MemberID | FName      | LName  | DateOfBirth |
+----------+------------+--------+-------------+
|        1 | John       | Smith  | 1980-01-01  |
|        3 | Max        | Land   | 1990-02-01  |
|        4 | Leonard    | Zimmer | 1999-02-01  |
|        5 | TestUpdate | Smith  | 1980-01-01  |
|        6 | TestFN     | TestLN | 1995-01-01  |
+----------+------------+--------+-------------+
5 rows in set (0.00 sec)
```

This method is crude – we update every table row from the JTable model, whether or not we have actually edited it. A more elegant approach would have been to have

- Sub-class an Editor for the table
- Have the editor remember which rows have been edited, in something like a TreeSet (no duplicates)
- Have saveData update only the rows in the TreeSet