Poul Klausen

JAVA 6

JDBC and database applications

Software Development

bookboon
LEARNING

POUL KLAUSEN

JAVA 6: JDBC AND DATABASE APPLICATIONS

SOFTWARE DEVELOPMENT

Java 6: JDBC and database applications: Software Development
1st edition

ISBN 978-87-403-1736-7
Peer review by Ove Thomsen, EA Dania

CONTENTS

FOREWORD

This book is the sixth in a series of books on software development. The programming language is Java, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process and how to develop good and robust applications. The subject of the current book is database applications, and how to write programs in Java that uses a database. In practice, programs that requiring many data transactions always uses databases, and the development of database applications is an important issue and necessary knowledge to work professionally as a software developer. In principle, it is simple, and a good piece of the road is just a question of accepting that you should write as shown in the book's examples, but database programming requires knowledge of SQL, and the hardest part is actually to write SQL statements correctly. Therefore, the book has an appendix that provides a quick introduction to SQL and describes the most basic syntax. In addition to SQL it is a prerequisite, to try out the book's examples and solve the book's exercises, that the reader has a running database, and MySQL will be used. The book has an appendix that shows how to install MySQL.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in Java. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples and exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code to the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with one or two larger sample programs, which focus primarily is on process and an explanation of how the program is written. On the other hand appears the code only to a limited extent – if at all – and the reader should instead study the finished program code perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

Most books also ends with an appendix dealing with a subject that would not be treated in the books. It may be issues on the installation of software or other topics in computer technology, which are not about software development, but where it is necessary to have an introductory knowledge. If the reader already is familiar with the subject, the current appendix can be skipped.

The programming language is, as mentioned Java, and besides the books use the following products:

- NetBeans as IDE for application development
- MySQL to the extent there is a need for a database server (from the book Java 6 onwards)
- GlassFish as a web server and application server (from the book Java 11 onwards)

It is products that are free of charge and free to install, and there is even talk about products, where the installation is progressing all by itself and without major efforts and challenges. In addition, there are on the web detailed installation instructions for all the three products. The products are available on Windows and Linux, and it therefore plays no special role if you use Linux or Windows.

All sample programs are developed and tested on machines running Linux. In fact, it plays no major role, as both Java and other products work in exactly the same way whether the platform is one or the other. Some places will be in the books where you could see that the platform is Linux, and this applies primarily commands that concerning the file system. Otherwise it has no meaning to the reader that the programs are developed on a Linux machine, and they can immediately also run under Windows unless a program refers to the file system where it may be necessary to change the name of a file.

Finally a little about what the books are not. It is not “how to write” or for that matter reference manuals in Java, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

Many applications need to use and maintain persistent data. In the preceding books, I have solved such tasks by using object serialization or plain text files, which indeed in some cases can be an excellent solution, but if there are large amounts of data with frequent updates, or data where multiple users must have access to the data, it is not a viable option. In these cases – and in practice, almost always – data are stored in databases.

A database is a collection of files that contains related data of one or another business, but it's a lot more because it is a software package that consists of programs that manages and maintains the database system's databases. This software package is called a *database management system*, and by a database system is meant partly the management system as well as the databases that the system maintains.

In this book I will address key concepts related to databases, but as the other of the books seen by a programmer, and thus especially what a programmer needs to know. The subject requires a specific database management system, and I will use *MySQL*. In fact, the system is not so important, but MySQL is available in an open source version that can be downloaded and installed for free. MySQL is today a powerful database system that supports all the services that one would expect from a good database management system, and besides MySQL is a widely used system in practice. In order to use MySQL from a Java program, you must at least install three products:

1. the database management system it self, that is the *MySQL server*
2. a program called *MySQL Workbench*, that the user can use to manage the system
3. a class library called *JDBC*, which makes it possible to write Java applications that uses databases

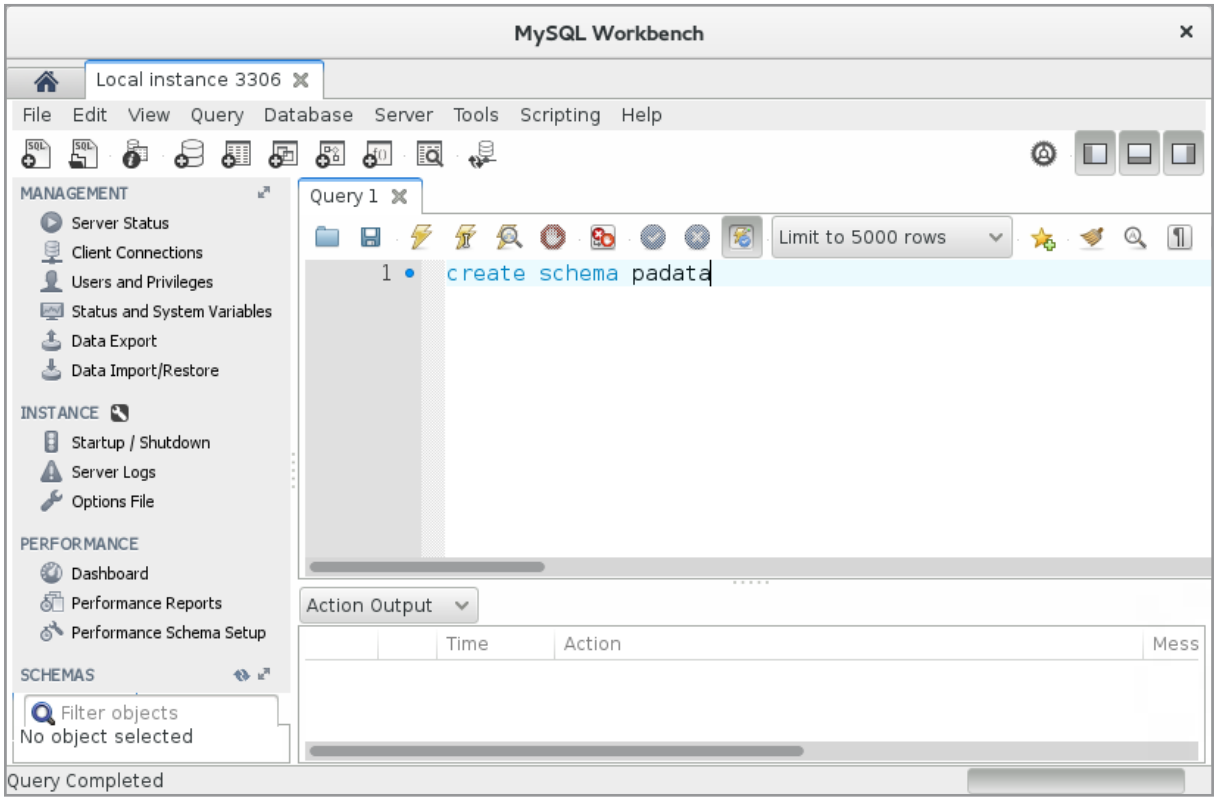
The book's appendix A shows how these products can be installed on the machine.

The following is based on two assumptions:

1. that you have a running *MySQL* server installed with the above products (see appendix A for how to install the products)
2. you have an introductory knowledge of *SQL* and know the most common statements (see appendix B for an introduction).

To show how to use databases from a Java program, you must also have a database. Open *MySQL Workbench* (see below). Perform then the command that is entered. You do this by clicking on the third icon (lightning) in the toolbar. The result is the creation of a database with the name *padata*. You may of course choose a different name if you wish, but it makes it easier to follow the book's examples and exercises if you choose the same name, which I have used. So far, it is an empty database, but through the book I will create several tables with data.

The folder to this book contains a *SQL* script called *CreateHistorie.sql*. Open this script in *MySQL Workbench*. The contents are as shown below:




```
use padata;
drop table if exists history;

create table history
(
 id int not null auto_increment primary key, # autogenerated surrogat key
 name varchar(50) not null,      # the person's name
 title varchar(30),              # the person's title
 birth int,                      # birth, start of reign, or equivalent
 death int,                      # death, end of reign, or equivalent
 country char(2),                # the country the person comes from
 description text                # a description
);
```

It is a script that creates a table with information about historical persons. A table consists of rows and columns, and each row contains data (information) about a particular person. A column has a type and a name. The name makes it possible to refer to the column in *SQL* statements, while the type determines what kind of values the column can hold. Taking the above script, indicating the first column, that the column must contain an auto-generated integer that starts with 1 and increases by 1 each time adding a row to the table. In addition is definied that the column must be the primary key, which means that the column's values must be unique and thus can serve to identify each row. The next two columns are of the type *varchar*, indicating that the column can contain a string, which in this case must be maximum of, respectively 50 and 30 characters. The next two columns again may contain integers, and the column with the name *country* must contain a string of 2 characters. Finally, the last column can contain any text (however, maximum 65535 characters). The column *name* is defined *not null*, which means that every row must have a value in that column. The other columns (except the first that is the primary key) does not need to have a value, but may be *null*.

The book's directory also contains a script *InsertHistory.sql* where below shows a few lines

```
use padata;
insert into history (name, death, title, country) values
 ('Gorm den Gamle', 958, 'King', 'DK');
insert into history (name, birth, death, title, country) values
 ('Harald Blåtand', 958, 987, 'King', 'DK');
...
```

If you opens the script in *MySQL Workbench* and executes it, the script adds 52 rows to the table *history*, where the rows contains data about Danish kings. The following assumes that this table is created, and the script *InsertHistory.sql* is performed.

ABOUT SURROGATE KEYS

When you have to create (design) a database table, you must choose one or more columns that can be used as the *primary key* (above *id*). This means that each row must have a unique value in this column or columns. Is it not possible, you can define a column, which then is assigned a unique value, and it is the case of the column *id* of the table *history,* when the value is automatically incremented by 1 each time you add a row to the table. This column can therefore be used as the primary key, and it is called a *surrogate key* corresponding to, that it is a replacement for another key column. Actually one should, if possible, avoid surrogate keys as they expand the table with data that does not tell anything, and in this case one might avoid the key as the Kings have unique names and thus the name could be used as the primary key. When I have nonetheless chosen a surrogate key, it is because a string – a column of the type *varchar* – not always is a good candidate for a primary key, first, it may be difficult to enter (you can misspell) and partly it is less effective than an *int*.

2 JDBC

JDBC is a family of classes that implements an interfaces defined in the *JDBC API*, and which makes it possible, that a Java program can communicate with a database server. In relation to the Java program the API has the necessary types as interfaces and classes defined in the package *java.sql*, while the specific types are implemented by a JDBC driver. So there has to be a JDBC driver available for each of the databases that the program must apply and often it is the database supplier that develops the JDBC driver. You can think of a JDBC driver as a layer between the database and the Java program that ensures that the program can use the database in the same way, whether it is one or the other database system.

2.1 HELLOJDBC

Before I go further and look at the details concerning JDBC, I will look at an example to show that it is quite easy to use a database from a Java program. The following program reads the rows in the table *history* and prints them on the screen:

```
package hellojdbc;

import java.sql.*;

public class HelloJDBC
{
 public static void main(String[] args)
 {
  Connection conn = null;
  Statement stmt = null;
  try
  {
   conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234");
   stmt = conn.createStatement();
   ResultSet res = stmt.executeQuery("SELECT * FROM history");
   while(res.next())
   {
    String name = res.getString("name");
    String title = res.getString("title");
    int birth = res.getInt("birth");
    int death = res.getInt("death");
    String country = res.getString("country");
    System.out.printf("%-25s%-10S%5d%5d %s\n", name, title, birth,
     death, country);
   }
  }
```

```
  catch(SQLException ex)
  {
   System.out.println(ex);
  }
  catch(Exception ex)
  {
   System.out.println(ex);
  }
  finally
  {
   try
   {
    if (stmt != null) stmt.close();
    if (conn != null) conn.close();
   }
   catch (SQLException ex)
   {
    System.out.println(ex);
   }
  }
 }
}
```

You must add an import statement to the necessary types:

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

Then you must define a connection to the database:

```
conn = DriverManager.getConnection(
 "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234");
```

Writing this connection is in fact the only difficulty of writing database applications, as it should be written correctly and as it depends on the specific database product. The method *getConnection()* has three parameters. The first is a connection string, while the last two are respectively the username and the password to the database. In this case, they are direct encoded as strings, and it may of course not be the solution to an application which is used in an operation condition, but about this later. Connection string consists of the database server that here is a MySQL server on the local machine:

```
jdbc:mysql://localhost
```

This is followed by the port number (which incidentally is not necessary, since 3306 is the default port number for MySQL), and finally follows the name of the database, which here is *padata*. In this case, there is also provided an information that the database does not require an SSL connection. If you need to use a database on another machine, you must type something else than *localhost* (possible an IP) and you may also need to use a different port number. Finally, be aware that the connection string looks different if is another database than MySQL.

After the connection is defined the program creates of a *Statement*, which is an object that represents a SQL statement, and the statement is executed by the method *executeQuery()*, wherein the parameter is a SQL SELECT command. The result of this method is a *ResultSet*, which is a collection with an object for each of the rows, the SQL command has extracted from the database. After the statement is successful excuted, the application performs a loop that iterates over the rows, and for each row prints a line with the values of the row's fields and thus the values in each column. You should note, how to refer to values using methods in the class *ResultSet* and the column names from the database.

The most database operations can raise a *SQLException*, so the statements are placed in a *try* block. The associated finally block closes the *Statement* and the *Connection*, and it is important to release the resources allocated.

If you executes the program, you get the following result, where I have only shown the first lines:

```
Gorm den Gamle      KING           0  958 DK
Harald Blåtand      KING         958  987 DK
Svend Tveskæg       KING         987 1014 DK
Harad d. 2.         KING        1014 1018 DK
Knud den Store      KING        1018 1035 DK
```

As is apparent from this example, it is basically simple to perform a data operation from a Java program. Of course there is more to tell than this example shows, and the the next chapter shows many other examples, but the differences are concerning SQL actually much more than Java.

EXERCISE 1

You must write a program that looks like *HelloJDBC*, but the program should only print the name and the years and only for those persons who are either kings or queens, and whose reign starts before 1500.

3 DATABASE OPERATIONS

The following describes a program, called *DbOperations* consisting of several test methods. The methods should show how to perform the most basic database operations using JDBC, where I only care about operations that maintains the database tables and hence SQL INSERT, UPDATE and DELETE statements, as well as operations that extract data from tables and thus SQL SELECT.

3.1 SQL STATEMENTS

The following test method uses the table *history*, and starts to delete a row in the table, insert a row, execute a query, update a row and finally execute a query again. This means that the method performs all the above four database operations. The code is as follows:

```
private static void test01()
{
 try (Connection conn = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  Statement stmt = conn.createStatement();
  System.out.println(stmt.executeUpdate(
```

```
    "DELETE FROM history WHERE name LIKE 'Ragnar Lodbrog'"));
   System.out.println(stmt.executeUpdate(
    "INSERT INTO history (name) VALUES('Ragnar Lodbrog')"));
   ResultSet res = stmt.executeQuery("SELECT name, title, birth, death, country
    FROM history WHERE name LIKE 'Ragnar Lodbrog'");
   while(res.next())
    System.out.printf("%-25s%-10s%5d%5d %s\n", res.getString("name"),
    res.getString("title"), res.getInt("birth"), res.getInt("death"),
    res.getString("country"));
   System.out.println(stmt.executeUpdate("UPDATE history SET title = 'Viking',
    country = 'DK' WHERE name LIKE 'Ragnar Lodbrog'"));
   res = stmt.executeQuery("SELECT name, title, birth, death, country FROM history
    WHERE name LIKE 'Ragnar Lodbrog'");
   while(res.next()) System.out.printf("%-25s%-10s%5d%5d %s\n",
    res.getString("name"), res.getString("title"), res.getInt("birth"),
    res.getInt("death"), res.getString("country"));
  }
  catch (SQLException ex)
  {
   System.out.println(ex);
  }
}
```

If the method is performed, you get the result:

```
0
1
Ragnar Lodbrog       NULL      0   0 null
1
Ragnar Lodbrog       VIKING     0   0 DK
```

In the same manner as in the program *HelloJDBC* the first thing is to open a connection to the database, and it is done in exactly the same way, but with the difference, that the *Connection* object this time is created within the parenteses in the *try* statement. This means that the connection is automatically closed when the method exits the *try* block, and in the same manner as in closing files it provides a more simple and also more readable code. It should be noted that when a database connection is closed, it also closes any opening statements that may be associated with the object.

The *try* block starts to create a *Statement* object, and the next statement performs a SQL DELETE. It is executed with the method *executeUpdate()* which, in addition to DELETE is used to perform INSERT and UPDATE statements. The method returns the number of rows that have changed. In this case, the statement delete the rows in which the name's value is *Ragnar Lodbrog*. If this is the first time that the method is run, there may not be such a row, and the method will return 0. If the method has been completed, there are a row with that value in the column *name*, and the method will return 1 and then will print 1 on the screen corresponding to the above result.

The next statement performs a SQL INSERT and inserts thus a row in the table. You should note that the statement only add a value in the column *name*, which is okay because the other columns allow *null* values. When this statement is executed, the method also prints 1 on the screen, because there are inserted one row in the table.

The next statement again performs a SQL SELECT, and this time it's with *executeQuery()*, which just is used to execute a SELECT. The query extract the rows where the column *name* has the value *Ragnar Lodbrog*. There is only one such row, and the row is printed in the next statement. The result is as shown in the third line above. Note that the method loops over the objects in the *ResultSet* object using a while loop. It is obviously no need for a loop when I know that there is only one row, but it is necessary to perform *res.next()* to step to the first row.

Finally is performed a SQL UPDATE and again with the method *executeUpdate()*, and as the last is performed a new query, showing that the row has been updated.

PREPAREDSTATEMENTS

In the above examples, all database operations are executed with a *Statement* object, which is an object that represents a static SQL statement, and thus an SQL statement that can not be changed. There is also a class called *PreparedStatement*, which represents a parameterized SQL statement. The objective of the following test method is to show how, a SQL expression can use parameters.

The project *DbOperations* have a class called *Person* that represents a row in the table *history*. This means that class contains a variable for each column in the table:

```
public class Person implements Comparable<Person>, Serializable
{
 private int id;                 // number, that is a surrogate key in the database
 private String name;            // the person's name
 private String title;           // the person's title
```

```
private Integer birth;        // birth, start to reign or other
private Integer death;        // year of death
private String country;       // country code as two characters
private String description;   // a description of the person
```

I will not show the full class here, partly because I earlier have seen on an almost identical class, and partly because the class consisting primarily of *get* and *set* methods. Note, however, that the type of *birth* and *death* is *Integer* not *int*. The reason is that the two columns in the database could contain *null* (and an *int* can not be *null*).

The class *DbOperations* has the following method that creates an *ArrayList* with some *Person* objects (I have only shown the first person – there are 12):

```
private static List<Person> createPersons()
{
 List<Person> list = new ArrayList();
 list.add(new Person(
  "Chlochilaicus", "Legends king", null, null, "DK", "Ruled about 515"));
 ...
 return list;
}
```

Then the test method:

```
private static void test02()
{
 List<Person> list = createPersons();
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  PreparedStatement stmt =
   conn.prepareStatement("DELETE FROM history WHERE name LIKE ?");
  for (Person pers : list)
  {
   stmt.setString(1, pers.getName());
   stmt.executeUpdate();
  }
  stmt = conn.prepareStatement("INSERT INTO history
   (name, title, birth, death, country, description) VALUES (?, ?, ?, ?, ?, ?)");
  for (Person pers : list)
  {
   stmt.setString(1, pers.getName());
   stmt.setString(2, pers.getTitle());
   if (pers.getBirth() == null) stmt.setNull(3, java.sql.Types.INTEGER);
   else stmt.setInt(3, pers.getBirth());
   if (pers.getDeath() == null) stmt.setNull(4, java.sql.Types.INTEGER);
   else stmt.setInt(4, pers.getDeath());
   stmt.setString(5, pers.getCountry());
   stmt.setString(6, pers.getDescription());
   stmt.executeUpdate();
  }
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
 print("SELECT * FROM history WHERE title LIKE 'Legends king'");
}
```

First the method creates a list with the 12 Person objects. Then, opening a connection to the database, and for this connection is created a *PreparedStatement* object. The class *PreparedStatement* is derived from *Statement* and represents as such a SQL expression, but an expression that may have parameters. In this case it is a SQL DELETE, and there is a single parameter that is indicated with a question mark. The next loop must delete all the people in the list (if available), and for each iteration the loop performs a DELETE, but first it must insert a value for the column *name* in the SQL expression. That is, that the question mark must be replaced by a value:

```
stmt.setString(1, pers.getNavn());
```

Here, 1 indicates it is the first parameter (and there is just the same), while the last parameter is the value. The value is assigned the *stmt* object with *setString()* as the type of the column is *VARCHAR*. There are similar methods for other column types.

When the loop is completed, another *PreparedStatement* is created, but this time for a SQL INSERT. It has no fewer than six parameters, and it is used to insert a row in the table for each person in the list. It happens in a loop, and for each repetition, the 6 parameters are initialized. You should especially note how to specify that a value must be *null*.

Finally the test method perfoms the method *print()* with a SQL SELECT as parameter, which is a method that prints the rows extracted.

EXERCISE 2

You should use *MySQL Workbench* to add a new table to the database *padata*. You should do this by writing a script (the same way as shown for the table *history*) when the table should be named *currency*, and it should have three columns:

1. *code*, which must be a currency code of three characters, which must be the primary key
2. *name*, that must be the name of the currency and can be up to 30 characters
3. *rate*, which must be the exchange rate and having the type DECIMAL with room for 10 digits and 4 after the decimal point

You must then write a program using the following array to update the *currency* table. The program should start by deleting the entire contents of the table (with a simple DELETE), and then it must use a *PreparedStatment* and insert a row for each string in the array:

```
private static String[] kurser =
{
 "Danske kroner;DKK;100.00",
 "Euro;EUR;746.00",
 "Amerikanske dollar;USD;674.44",
 "Britiske pund;GBP;1034.10",
 "Svenske kroner;SEK;79.31",
 "Norske kroner;NOK;79.68",
 "Schweiziske franc;CHF;685.66",
 "Japanske yen;JPY;5.6065",
 "Australske dollar;AUD;487.61",
 "Brasilianske real;BRL;171.98",
 "Bulgarske lev;BGN;381.43",
 "Canadiske dollar;CAD;510.19",
 "Filippinske peso;PHP;14.40",
 "Hongkong dollar;HKD;87.02",
 "Indiske rupee;INR;10.38",
```

```
 "Indonesiske rupiah;IDR;0.0494",
 "Israelske shekel;ILS;174.48",
 "Kinesiske Yuan renminbi;CNY;106.17",
 "Kroatiske kuna;HRK;97.87",
 "Malaysiske ringgit;MYR;157.78",
 "Mexicanske peso;MXN;40.65",
 "New Zealandske dollar;NZD;458.77",
 "Polske zloty;PLN;173.82",
 "Rumænske lei;RON;168.13",
 "Russiske rubel;RUB;10.42",
 "Singapore dollar;SGD;483.72",
 "Sydafrikanske rand;ZAR;49.08",
 "Sydkoreanske won;KRW;0.5933",
 "Thailandske baht;THB;19.00",
 "Tjekkiske koruna;CZK;27.53",
 "Tyrkiske lira;TRY;231.78",
 "Ungarske forint;HUF;2.385"
};
```

To insert a value for the rate in a *PreparedStatement*, you must specify that the type is DECIMAL, and here you can use *setBigDecimal()*.

Before you run the program, you may update the array with today's current rates.

4 RESULTSET

The most complex SQL statement is SELECT, and it is a very complex statement with many options. From a Java program you can perform a SQL SELECT with *executeQuery()*, and although the SELECT statement can be complex, the result seen from the Java program is always the same: A *ResultSet* object. You can think of the object as a collection of objects with an internal cursor that points to a current object (a row), or the cursor is *null* if it is not pointing to anything. The class *ResultSet* defines many methods that can basically be divided into three kinds:

1. methods to move the cursor
2. *get* methods that returns the value for a column at the current row
3. update methods that update the column value for the current row and you even have the opportunity to write the changes back to the database

To create a *Statement* object, I have so far used the following syntax:

```
Statement stmt = connection.createStatement();
```

that creates a *Statement* object, but *createStatement()* has an override of the form:

```
createStatement(int resultSetType, int resultSetConcurrency)
```

Here you can for the first parameter set three values:

1. *ResultSet.TYPE_FORWARD_ONLY*, that means, that the cursor can move forward and is the default value
2. *ResultSet.TYPE_SCROLL_INSENSITIVE*, that means, that the cursor kan move both back and forth, and that the *ResultSet* is not updated with changes that other users had to make to the database after this *ResulSet* is created
3. *ResultSet.TYPE_SCROLL_SENSITIVE*, which means the same as above, but with the difference that the resultset is updated with changes that other users had to make to the database after this *ResulSet* is created

As default a *ResulSet* is readonly, but for the last of above two parameters there are two options

1. *ResultSet.CONCUR_READ_ONLY*, that creates a readonly *ResultSet* and is the default
2. *ResultSet.CONCUR_UPDATABLE*, which creates a *ResultSet*, that is updatable

You can set the same parameters for *prepareStatement()* with the following syntax:

```
prepareStatement(String sql, int resultSetType, int resultSetConcurrency)
```

A ResulSet has many ways to navigate the cursor, and you are encouraged to examine the options. They are basically self-explanatory, and the following test method (which is a method in *DbOperations*) gives examples of the use of the methods on a *ResultSet*, where you can scroll back and forth:

```
private static void test03()
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  Statement stmt = conn.createStatement(
   ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
  ResultSet res = stmt.executeQuery("SELECT name, birth, death FROM history
   where title LIKE 'King' OR title LIKE 'Queen'");
  res.last();
  print(res);
  res.first();
  print(res);
  res.next();
  print(res);
  res.previous();
  print(res);
  res.absolute(50);
  print(res);
  res.relative(-5);
  print(res);
  res.afterLast();
  while (res.previous()) if (res.getString(1).startsWith("Gorm")) break;
  print(res);
  System.out.println(res.getRow());
  res.last();
  System.out.println(res.getRow() + 1);
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

If you performs the method, you get the result:

```
Margrethe d. 2.      1972         0
Gorm den Gamle       0          958
Harald Blåtand       958        987
Gorm den Gamle       0          958
Frederik d. 9.       1947      1972
Christian d. 8.      1839      1848
Gorm den Gamle       0          958
1
52
```

Each line are printed with the method

```
private static void print(ResultSet res)
{
 try
 {
  String name = res.getString(1);
  int birth = res.getInt(2);
  int death = res.getInt(3);
  System.out.printf("%-20s%5d%5d\n", name, birth, death);
 }
```

```
 catch (Exception ex)
 {
  System.out.println(ex);
 }
}
```

A resultset has a number of methods used to retrieve the values for the current row, and in the foregoing examples, I each time has referenced the values with the column name. Actually, it is also allowed to enter an index, starting with 1 for the first column. You should note that the index does not refer to the physical table, but the columns specified in the SELECT statement. In this case, there are three columns, and the indexes are therefore 1, 2 and 3.

4.1 UPDATE A RESULTSET

The following test method shows that one can update a resultset:

```
private static void test04()
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
   ResultSet.CONCUR_UPDATABLE);
  ResultSet res = stmt.executeQuery("SELECT * FROM history");
  insert(res);
  res.last();
  printRow(res);
  update(res, "King", "Probably lived between the years 400 and 500");
  printRow(res);
  update(res, "Legends king", res.getString("description") +
   " and his story is described in Rolf Krake's Saga");
  printRow(res);
  while (true)
  {
   res = stmt.executeQuery("SELECT id FROM history WHERE name LIKE 'Rolf%'");
   if (res.next())
   {
    int id;
    System.out.println(id = res.getInt("id"));
    print("SELECT * FROM history WHERE id = " + id);
    delete(res);
   } else break;
  }
 }
```

```
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

First, you notice that the creation of a *Statement* object that allows you to update a *ResultSet*. Next, all rows are extracted from the table *history*, and then the method *insert()* is called with the *ResultSet* as a parameter:

```
private static void insert(ResultSet res)
{
 try
 {
  res.moveToInsertRow();
  res.updateString("name", "Rolf Krake");
  res.updateString("country", "DK");
  res.insertRow();
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

This method adds a new row to the *ResultSet* object and assigns values to two of the columns. The last method writes the changes physically back to the database. If you do not want that, you can omit this statement, and the row is just inserted in the *ResultSet* object in memonry.

After the new row is inserted, the test method set the cursor to the last row and the prints the row by a method *printRow()*. The next statement calls a method *update()*, which updates the row that the cursor points to:

```
private static void update(ResultSet res, String title, String description)
{
 try
 {
  res.updateString("title", title);
  res.updateString("description", description);
  res.updateRow();
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

There is updated two columns, and again it's the last statement, which writes the changes back to the database. The test method prints then the row again, so you can see that the changes are made, after which the row is updated once again.

Next, the method performs an infinite *while* loop where it starts to execute a query on persons where the name starts with *Rolf*. There should only be 1. If there is such a row (*res.next()* is true) the row is removed with the method *delete()*. If the row is not found a break is performed, and the loop stops. The goals is to show that the updates are physically in the database, so that in the next iteration, no rows are found. The method *delete()* is quite simple and deletes the row that the cursor points to:

```
private static void delete(ResultSet res)
{
 try
 {
  res.deleteRow();
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

You should note that the test method's *while* loop in case there are found a row calls the method *print()* with a SQL expression as a parameter. The objective is to show that the new row is available on request associated with another connection, and thus the row is physically stored in the database. If you performs the test method, the result could be:

```
Rolf Krake            null            0   0 DK
null

Rolf Krake            King            0   0 DK
Probably lived between the years 400 and 500

Rolf Krake            Legends king    0   0 DK
Probably lived between the years 400 and 500 and his
story is described in Rolf Krake's Saga

70
Rolf Krake            Legends king    0   0 DK
Probably lived between the years 400 and 500 and his
story is described in Rolf Krake's Saga
```

The test method also uses a method *printRow()*, which prints a row on the screen (the row that the cursor points to). The method requires no special explanation and is not shown here, but it is has a statement that is commented out:

```
// System.out.println(res.rowInserted() + " " + res.rowUpdated() + " " +
   res.rowDeleted());
```

When it is commented out, it is simply because it does not work. The method should test whether a *ResultSet* has a row that is inserted, updated, or deleted, and where the changes are not yet written back to the database. Note specifically that when a row is deleted, there is nothing else, than the row is marked as deleted in the resultset, but it is not removed. These methods are described in the JDBC standard, but it is not the same as the supplier of the JDBC driver implements these methods and it does not apply to the JDBC driver for MySQL. The lesson is that whether the characteristics of the JDBC standard is implemented, depends on the specific driver, and thus of the database product used, but also the version of the driver.

4.2 MUNICIPALITIES AND ZIPCODES

Following the above, I will show a more realistic example of a database application. The database is still *padata*, but I will expands it with four new tables. The first should be a table of Danish regions:

regnr	**name**
1081	Region Nordjylland
1082	Region Midtjylland
1083	Region Syddanmark
1084	Region Hovedstaden
1085	Region Sjælland

and is thus a simple table with two columns and five rows. The next table is a table of Danish municipalities, where each municipality is recorded as

1. the municipality number (a unique number of 3 digits)
2. the municipality name
3. the region, that the municipalty belongs to
4. the municipality's area in square kilometers
5. the municipality's inhabitants
6. year of the municipality's inhabitants

The region is the region number, so there is a 1:n relation between municipalities and regions, such that each municipality is linked to a specific region.

The third table is a table of Danish zip codes and is a simple table with two columns that contain respectively the code and the city name.

A municipality uses several zip codes, and a specific zip code can actually be used by several municipalities, and such there is an m:n relation between municipalities and zip codes. Such a relationship is implemented with a table where a row connects a zip code and a municipality.

Accordingly, you can expand the database with four new tables using the following script:

```
use padata;
drop table if exists post;
drop table if exists municipality;
drop table if exists region;
drop table if exists zipcode;
```

```
create table region
(
 regnr int not null primary key,
 name varchar(30) not null
);

create table municipality
(
 munnr int not null primary key,
 name varchar(30) not null,
 regnr int not null,
 area decimal(10, 2),
 number int,
 year int,
 foreign key (regnr) references region(regnr)
);

create table zipcode
(
 code char(4) not null primary key,
 city varchar(30) not null
);
```

```
create table post
(
 code char(4) not null,
 munnr int not null,
 primary key (code, munnr),
 foreign key (code) references zipcode(code),
 foreign key (munnr) references municipality(munnr)
);

insert into region values (1081, 'Region Nordjylland');
insert into region values (1082, 'Region Midtjylland');
insert into region values (1083, 'Region Syddanmark');
insert into region values (1084, 'Region Hovedstaden');
insert into region values (1085, 'Region Sjælland');
```

There is not much to explain, but you should note that the script inserts the five rows in the table *region*. The script is called *CreateDenmark.sql*.

It you performs the script, it creates the four tables, but the tables other than *region* are all empty and contains no data. The books directory contains a file named *zipcodes*. It is a text file that contains 1106 lines with Danish zip codes, where each line contains a code and a city name separated by a semicolon. The file should be used to update the table *zipcode*, and you can do that with a SQL command *LOAD DATA INFILE*. It is a command that MySQL do not like, because it is not without risk, and you can, in principle, copy anything into the database. Therefore, you can not immediately execute the command from *MySQL Workbench*. To use the command to load the content of the file to a database, you has to place the file in a special directory that only root has access to:

```
/var/lib/mysql-files
```

I have (as *root*) in this directory created a subdirectory *data*. Next, I have copied the file *zipcodes* to this directory. Then the table can be updated using the following script (*LoadZipcodes.sql*):

```
use padata;
load data infile '/var/lib/mysql-files/data/zipcodes' into table zipcode
CHARACTER SET UTF8
fields terminated by ';' lines terminated by '\n'
(code, city);
```

The result is that the table *zipcode* now have 1106 rows.

Then there is the table *municipality*, and the book's directory has a file *municipalities* that contains lines with data on Danish municipalities. Below is shown the first three lines:

```
773;Morsø Kommune;1081;364.42;20815;2015;7900;7950;7960;7970;7980;7990
787;Thisted Kommune;1081;1095.63;44078;2015;7700;7730;7741;7742;7752;
  7755;7760;7770
810;Brønderslev Kommune;1081;633.38;35781;2015;9320;9330;9340;9352;9370;
  9382;9440;9480;9700;9740;9750;9760
```

A line contains separated by semicolons

1. municipality number
2. municipality name
3. region number
4. area
5. number of inhabitants
6. year for number of inhabitants

This is followed by a variable number of zip codes, which are the zip codes that this municipality uses. For each line in the file there must be inserted a row in the table *municipality*, and for each zip code a row in the table *post*. The easiest is to write a small program that reads the file and updates the database:

```
package createmunicipalities;

import java.sql.*;
import java.io.*;
import java.math.*;

public class CreateMunicipalities
{
 public static void main(String[] args)
 {
  try (Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234");
   BufferedReader reader = new BufferedReader(
    new FileReader(System.getProperty("user.home") + "/data/municipalities")))
  {
   Statement stmt = conn.createStatement();
   stmt.execute("DELETE FROM post");
   stmt.execute("DELETE FROM municipality");
   PreparedStatement stmt1 = conn.prepareStatement(
    "INSERT INTO municipality VALUES(?, ?, ?, ?, ?, ?)");
   PreparedStatement stmt2 = conn.prepareStatement(
    "INSERT INTO post VALUES(?, ?)");
```

```
for (String line = reader.readLine(); line != null; line = reader.readLine())
{
 String[] elem = line.split(";");
 int munnr = Integer.parseInt(elem[0]);
 stmt1.setInt(1, munnr);
 stmt1.setString(2, elem[1]);
 stmt1.setInt(3, Integer.parseInt(elem[2]));
 stmt1.setBigDecimal(4, new BigDecimal(elem[3]));
 stmt1.setInt(5, Integer.parseInt(elem[4]));
 stmt1.setInt(6, Integer.parseInt(elem[5]));
 stmt1.executeUpdate();
 System.out.println(elem[1]);
 for (int i = 6; i < elem.length; ++i)
 {
  try
  {
   stmt2.setString(1, zipcode(elem[i]));
   stmt2.setInt(2, munnr);
   stmt2.executeUpdate();
  }
  catch (Exception ex)
  {
   System.out.println(munnr + " " + elem[i]);
  }
```

```
     }
    }
   }
   catch (Exception ex)
   {
    System.out.println(ex);
   }
  }

  private static String zipcode(String str)
  {
   if (str.length() == 3) str = "0" + str;
   return str;
  }
}
```

In the parentheses after *try* are in the usual way opened a connection to the database and the file *municipalities* that I have copied to the folder *data* under my home directory. The first thing that happens is that the content of the two database tables are deleted and then are created two *PreparedStatement* objects with INSERT statements for the two tables. The next for *loop* reads the file line by line, and each line is split into tokens that are used to initialize the parameters of *stmt1*. After having inserted a row in the table *municipalities*, there is inner for loop that runs over all zip codes and insert rows in the table *post* for this municipality.

You should note that it actually takes a long time to execute the program. The program executes more than 2000 SQL INSERT commands, and it is a tedious procces to make it from Java, as shown in this example, and there are also as shown later better ways to accomplish the same.

As an example of a program that uses the new tables, determines the following test method (still a method in the program *DbOperations*) the number of residents in Region Midtjylland:

```
private static void test05()
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  Statement stmt = conn.createStatement();
  ResultSet res = stmt.executeQuery("SELECT number FROM municipality AS M,
   region AS R WHERE M.regnr = R.regnr AND R.name LIKE '%Midtjylland'");
  int count = 0;
  while (res.next()) count += res.getInt("number");
```

```
  System.out.println("Number of inhabitants in Region Midtjylland: " + count);
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

You should note that the only thing new is a little more complex SQL SELECT statement, but from Java there is nothing new.

EXERCISE 3

Write a program that from the numbers in the table *municipality* on the screen prints the population density in Jutland and Fyn.

PROBLEM 1

You must expand the database *padata* with two other tables. The first should be named *world* and must contains the names of all continents:

```
create table world
(
 code char(2) not null primary key,
 name varchar(15) not null
);
```

The table must contain the following data:

- AS Asia
- AF Africa
- NA North America
- SA South America
- AN Antarctica
- EU Europe
- OC Oceania

The second table should contain information on countries:

```
create table country
(
 code2 char(2) not null primary key,    # country code
 code3 varchar(3),                      # country code
 name varchar(50) not null,             # the country's name
```

```
 area int,                                  # the country's area
 number int,                                # number of inhabitants
 continent char(2),                         # continent
 currency char(3),                          # currency code
 foreign key (continent) references world(code),
 foreign key (currency) references currency(code)
);
```

Write a script that creates these two tables and inserts information about the continents in the first table.

The folder to this book contains a file called *countries*. It is a plain text file in which each line contains information about a country. Below is the first lines:

```
AF;AFG;Afghanistan
AL;ALB;Albanien
DZ;DZA;Algir
AS;ASM;Amerikansk Samoa
US;USA;Amerikas Forenede Stater (USA);US Dollar;USD;553.14
AD;AND;Andorra
AO;AGO;Angola
AI;AIA;Anguilla
```

```
AQ;ATA;Antarktis
AG;ATG;Antigua og Barbuda
AR;ARG;Argentina
AM;ARM;Armenien
AW;ABW;Aruba
AZ;AZE;Aserbajdsjan
AU;AUS;Australien;Australsk dollar;AUD;568.80
BS;BHS;Bahama
BH;BHR;Bahrain
```

The fields are separated by semicolons. For each country there is at least 3 fields: country code on 2 characters (primary key), country code on 3 characters (which may be blank) and the country's name. In addition, there may be three additional fields, that is the name of the currency which the country uses, the currency code and an exchange rate.

You must now write a program that can update the table country by reading the above file. If there is a currency, the currency code is inserted in the column for currency, otherwise the value should just be *null*. There is a particular problem when the file for a country indicates a currency, and that currency not exitst in the table *currency*. For these currencies, the program must also update the table *currency*.

4.3 STORED PROCEDURES

A stored procedure is a script that contains SQL statements that is executed when the procedure is performed. When one speaks of a stored procedure, it means that the procedure is translated and stored in the database, which means that it is highly effective to execute a stored procedure. In MySQL Workbench you can create a stored procedure by right-clicking *Stored Procedures* for the database under *Schemas*, and you get as a skeleton for a procedure:

```
CREATE PROCEDURE 'new_procedure' ()
BEGIN

END
```

As an example, I have written a very simple procedure which *OUT parameter* returns the number of persons in the table *history* where the year falls within a range:

```
CREATE PROCEDURE 'persons' (IN a int, IN b int, OUT c int)
BEGIN
 SELECT count(id) FROM history WHERE (a <= birth AND birth <= b) OR
    (a <= death AND death <= b) INTO c;
END
```

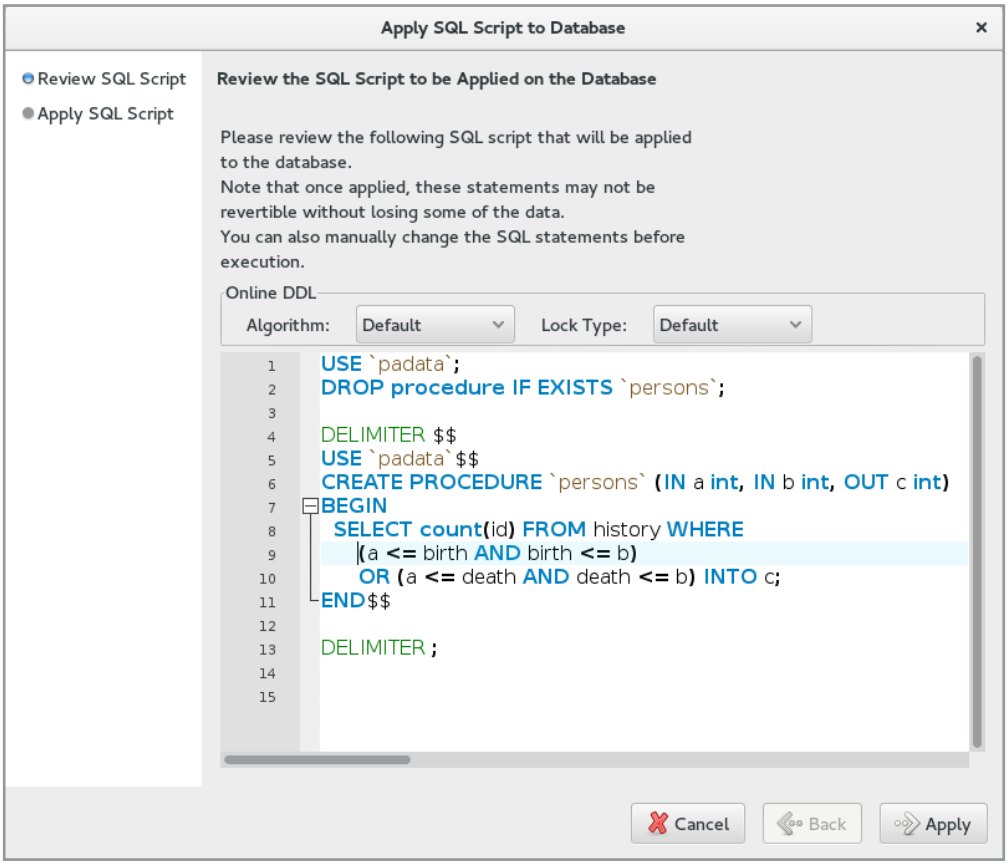
When you have written the procedure, you must click *Apply*, and you are then given a window as shown on the next page, and click *Apply* again the procedure is translated and store it in the database. Once the procedure is saved, it can be executed like like any other SQL command:

```
use padata;

call padata.persons(1100, 1600, @count);
select @count;
```

The following test method shows how to execute a stored procedure from a Java program:

```
private static void test06()
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  CallableStatement stmt = conn.prepareCall("CALL persons(?, ?, ?)");
  stmt.setInt(1, 1200);
  stmt.setInt(2, 1300);
  stmt.registerOutParameter(3, Types.INTEGER);
  stmt.execute();
  System.out.println(stmt.getInt(3));
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

JDBC defines in form of the types *Statement* and *PreparedStatement* types that represents a SQL statement, and there is also a type called *CallableStatement*, representing a stored procedure. It is parameterized in the same way as a *PreparedStatement*, and the input parameters are initialized in the same manner. By contrast, the output parameters must be registered with a special syntax including to assign a type. After this, the procedure is executed with the method *execute()*.



5 DATA TYPES

In a database table, each column has a data type and in Java data is stored in variables of a particular type. It is therefore necessary to know how JDBC are converting types in the database to the Java variables, and vice versa. The preceding programs have already shown many examples, for example when values are retrieved from a *ResultSet* with a *get* method, or when parameters in a *PreparedStatement* are initialized with a *set* method. The following table shows the relationship between these types:

SQL	Java	Methods
VARCHAR	String	getString(), setString()
CHAR	String	getString(), setString()
LONGVARCHAR	String	getString(), setString()
BIT	boolean	getBoolean(), setBoolean()
NUMERIC	java.math.BigDecimal	getBigDecimal(), setBigDecimal()
TINYINT	byte	getByte(), setByte()
SMALLINT	short	getShort(), setShort()
INTEGER	int	getInt(), setInt()
BIGINT	long	getLong(), setLong()
REAL	float	getFloat(), setFloat()
FLOAT	float	getFloat(), setFloat()
DOUBLE	double	getDouble(), setDouble()
VARBINARY	byte[]	getBytes(), setBytes()
BINARY	byte[]	getBytes(), setBytes()
DATE	java.sql.Date	getDate(), setDate()
TIME	java.sql.Time	getTime(), setTime()
TIMESTAMP	java.sql.Timestamp	getTimestamp(), setTimestamp()
CLOB	java.sql.Clob	getClob(), setClob()
BLOB	java.sql.Blob	getBlob(), setBlob()

SQL	Java	Methods
ARRAY	*java.sql.Array*	*getArray(), setArray()*
REF	*java.sql.Ref*	*getRef(), setRef()*
STRUCT	*java.sql.Struct*	*getStruct(), setStruct()*

Thus, it is important to note that the package *java.sql* defines special data types that correspond to some of the SQL data types. Converting between types to date and time often results in problems with database applications, and to facilitate the problem of converting these types, I have expanded my library *PaLib* with the following class:

```
package palib.util;

import java.util.*;

public class Db
{
 public static java.sql.Date toDate(Date date)
 {
  return new java.sql.Date(date.getTime());
 }

 public static java.sql.Time toTime(Date date)
 {
  return new java.sql.Time(date.getTime());
 }

 public static java.sql.Timestamp toTimestamp(Date date)
 {
  return new java.sql.Timestamp(date.getTime());
 }

 public static java.sql.Date toDate(Calendar date)
 {
  return new java.sql.Date(date.getTimeInMillis());
 }

 public static java.sql.Time toTime(Calendar date)
 {
  return new java.sql.Time(date.getTimeInMillis());
 }
```

```
public static java.sql.Timestamp toTimestamp(Calendar date)
{
 return new java.sql.Timestamp(date.getTimeInMillis());
}

public static Calendar toCalendar(java.sql.Date date)
{
 Calendar c = new GregorianCalendar();
 c.setTime(date);
 return c;
}

public static Calendar toCalendar(java.sql.Time time)
{
 Calendar c = new GregorianCalendar();
 c.setTime(time);
 return c;
}
```

```
public static Calendar toCalendar(java.sql.Date date, java.sql.Time time)
{
 Calendar c1 = new GregorianCalendar();
 Calendar c2 = new GregorianCalendar();
 c1.setTime(date);
 c2.setTime(time);
 c1.set(Calendar.HOUR_OF_DAY, c2.get(Calendar.HOUR_OF_DAY));
 c1.set(Calendar.MINUTE, c2.get(Calendar.MINUTE));
 c1.set(Calendar.SECOND, c2.get(Calendar.SECOND));
 return c1;
}

/**
 * Converts a string to a Calendar object. The string should be of the form
 *  DD-MM-YYYY HH:MM:SS
 * and must represent a valid date and time. The year must be at least
 * 1700, as the date must be from the Gregorian calendar.
 * It is permitted only to indicate the date and not a time indication.
 * @param str The string to be converted
 * @return The string is converted into a Calendar object
 * @throws UtilException If the string does not represent a valid date
 */ public static Calendar toCalendar(String str) throws UtilException
{
 try
 {
  int day = Integer.parseInt(str.substring(0, 2));
  int month = Integer.parseInt(str.substring(3, 5));
  int year = Integer.parseInt(str.substring(6, 10));
  int hour = 0;
  int minute = 0;
  int second = 0;
  if (str.length() > 10)
  {
   hour = Integer.parseInt(str.substring(11, 13));
   minute = Integer.parseInt(str.substring(14, 16));
   second = Integer.parseInt(str.substring(17).trim());
  }
  if (year >= 1700 && month >= 1 && month <= 12 && day >= 1 &&
   day <= days(year, month) && hour >= 0 && hour <= 23 && minute >= 0 &&
   minute <= 59 && second >= 0 && second <= 59)
   return new GregorianCalendar(year, month - 1, day, hour, minute, second);
 }
```

```
  catch (Exception ex)
  {
  }
  throw new UtilException("String can not be converted to a Calendar object");
 }

 public static String toStr(Calendar cal)
 {
  return String.format("%02d-%02d-%04d %02d:%02d:%02d",
   cal.get(Calendar.DAY_OF_MONTH), cal.get(Calendar.MONTH) + 1,
   cal.get(Calendar.YEAR), cal.get(Calendar.HOUR_OF_DAY),
   cal.get(Calendar.MINUTE), cal.get(Calendar.SECOND));
 }

 public static boolean leapYear(int aar)
 {
  return aar % 100 == 0 ? aar % 400 == 0 : aar % 4 == 0;
 }

 private static int days(int aar, int mdr)
 {
  if (mdr == 2) return leapYear(aar) ? 29 : 28;
  if (mdr == 4 || mdr == 6 || mdr == 9 || mdr == 11) return 30;
  return 31;
 }
}
```

It should be easy enough to figure out each method, and from practical programming, it is especially important to be able to convert to and from *Calendar* objects and the corresponding SQL data types. The following test method uses the above conversion methods:

```
private static void test07()
{
 System.out.println(Db.toDate(new java.util.Date()));
 System.out.println(Db.toTime(new java.util.Date()));
 System.out.println(Db.toDate(Calendar.getInstance()));
 System.out.println(Db.toTime(Calendar.getInstance()));
 System.out.println(Db.toStr(Db.toCalendar(Db.toDate(new java.util.Date()))));
 System.out.println(Db.toStr(Db.toCalendar(Db.toTime(new java.util.Date()))));
 System.out.println(Db.toStr(Db.toCalendar(Db.toDate(new java.util.Date()),
  Db.toTime(new java.util.Date()))));
}
```

and an example of running the method could be:

```
2017-01-01
16:41:08
2017-01-01
16:41:08
01-01-2017 16:41:08
01-01-2017 16:41:08
01-01-2017 16:41:08
```

Another problem concerning data type is *NULL* values as *null* in databases and Java is not the same. For example will a *ResultSet*'s *get* methods convert a NULL value for a primitive type to the default value (often 0), which is not always appropriate. In the test method *test02()*, I have shown how you from an application can write NULL values to the database. The following test method demonstrates an application of the method *wasNull()* where a *ResultSet* tests whether the last value, that is returned with a *get* method, was NULL:

```
private static void test08()
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  Statement stmt = conn.createStatement();
  ResultSet res = stmt.executeQuery("SELECT name, birth, death FROM history
   WHERE title Like 'Queen'");
  while (res.next())
  {
   String navn = res.getString(1);
   String birth = "";
   int t = res.getInt(2);
   if (!res.wasNull()) birth = "" + t;
   String death = "";
   t = res.getInt(3);
   if (!res.wasNull()) death = "" + t;
   System.out.printf("%-25s%4s - %4s\n", navn, birth, death);
  }
 }
 catch (SQLException ex)
 {
  System.out.println(ex);
 }
}
```

If the method is performed, you get the result:

```
Margrete d. 1.       1387 - 1412
Margrethe d. 2.      1972 -
```

6 TRANSACTIONS

When performing an SQL statement using either a *Statement* or *PreparedStatement* object, the database is immediately updated. Sometimes it is not appropriate, since you often have multiple SQL statements thart must be executed as a whole, where either all statements are executed correctly or everyone are ignored. In database contexts wee talks about a *transaction* that simply means that multiple SQL statements can be perceived as a whole. The following test method inserts three rows in the table *history*, where the three statements is performed as a transaction:

```
private static void test09()
{
 Connection conn = null;
 try
 {
  conn = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234");
  conn.setAutoCommit(false);
  ArrayList<Integer> list = new ArrayList();
  PreparedStatement stmt = conn.prepareStatement(
   "INSERT INTO history (name, title) VALUES (?, ?)");
  insert(stmt, "Egil Skallegrimson", null, list);
  insert(stmt, "Knud Larvard", "Duke", list);
  insert(stmt, null, "Rebel", list);
  conn.commit();
 }
 catch(SQLException ex)
 {
  if (conn != null) try { conn.rollback(); } catch (Exception ex2) {}
  System.out.println(ex);
 }
 finally
 {
  try { if (conn != null) conn.close(); } catch (SQLException ex) {}
 }
}
```

First as before is in the usual way created a connection to the database, and the next statement says that there should be no *auto commit*. This means that subsequent SQL statements are not immediately been performed on the database (they are marked as statements that can be rolled back), and the effect will not appear before performing a *commit()*. The statement

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

means that a transaction is starting. The next statement creates an *ArrayList* and is explained shortly. Next, is created a *PreparedStatement*, which represents a SQL command that inserts a row in the table *history*, but only insert values in the columns *name* and *title*. Then the method performs three calls of a method *insert()*, which inserts a row in the table. If it goes well, a *commit()* is performed, and the three insertions of rows are permanent inserted in the database. If, however, one of the three *insert()* raises a *SQLException*, the catch block is performed and executes a *rollback()*, which means that the database operations performed after the start of the transaction is rolled back and the database has the same condition as before the transaction was started.

Then there is the method *insert()*:

```
private static void insert(PreparedStatement
stmt, String navn, String titel,
  ArrayList<Integer> list) throws SQLException
{
 prepareStatement(stmt, navn, titel);
 stmt.executeUpdate();
 list.add(getId(stmt));
}
```

The method performs a call to a method that inserts the values for the parameters in *stmt*, and then the statement is executed. The primary key of the table *history* is an auto-generated *Integer*, and sometimes you need to know the id when a row is assigned. This can be done in several ways, but I've written a little method for this purpose:

```
private static int getId(Statement stmt) throws SQLException
{
 ResultSet res = stmt.executeQuery("SELECT LAST_INSERT_ID()");
 res.next();
 return res.getInt(1);
}
```

Here is *LAST_INSERT_ID()* a function in MySQL, which returns the last assigned auto-generated ID. These keys are in this example added to an *ArrayList* that is not used for anything, but it does in the next example, and here you must mainly observe how to grab an auto generated key.

If one executes the above test method, the last INSERT fails, since there is no value for the column *name*, and in the database is defined that this column must have a value. That is, there is a rool back. If you open *MySQL Workbench*, you can easily convince yourself that the rows are not inserted.

The example shows basically what there is to say about transactions, but there is also a possibility to define a *Savepoint*, which can be used to indicate that not all operations has to be rolled back. Consider the following test method, which is an extension of the above method:

```
private static void test10()
{
 Connection conn = null;
 Savepoint point = null;
 try
 {
  conn = DriverManager.getConnection(
   "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234");
  conn.setAutoCommit(false);
  ArrayList<Integer> list = new ArrayList();
  PreparedStatement stmt1 = conn.prepareStatement(
   "INSERT INTO history (name, title) VALUES (?, ?)");
  PreparedStatement stmt2 = conn.prepareStatement(
   "UPDATE history SET country = ? WHERE id = ?");
  insert(stmt1, "Egil Skallegrimson", null, list);
  insert(stmt1, "Knud Larvard", "Duke", list);
```

```
  insert(stmt1, "Skipper Clement", "Rebel", list);
  conn.commit();
  point = conn.setSavepoint();
  update(stmt2, "DK", list.get(1));
  update(stmt2, "DK", list.get(2));
  update(stmt2, "Island", list.get(0));
  conn.commit();
 }
 catch(SQLException ex)
 {
  if (conn != null)
   try
   {
    if (point == null) conn.rollback(); else conn.rollback(point);
   } catch (Exception ex2) {}
  System.out.println(ex);
 }
 finally
 {
  try { if (conn != null) conn.close(); } catch (SQLException ex) {}
 }
}
```

The first part is basically the same as the method *test09()*, except that there is defined an additional *PreparedStatement* to a SQL UPDATE. Furthermore, there is defined a name for the last INSERT, such it will not fails. After the three *inserts()* methods are performed, the test method performs a *commit()* and defines a *Savepoint*, indicating that up to this point of the transaction the statements are executed correct. Next, the method *update()* is executed for each of the three rows that are inserted in the table. The method updates the column *country* and it is here that I uses the keys that are stored in the list. Here are the first two *update()* done correctly, but the last one raises a SQLException when the column *country* only has room for two characters. The result is that control is transferred to the catch block, which perform a rollback, but when the *Savepoint* object is not *null*, one can conclude that the *Savepoint* is reached, and thus the three *insert()* methods are performed without error. Therefore is only rolled back from the *Savepoint*. If you examine the table in *MySQL Workbench* you will see that the three INSERT statements are executed, but the three UPDATE statements are not.

6.1 BACH UPDATES

As a last thing concerning JDBC and database operations, I will mention batch updates. If you have to perform many database operations – such as many INSERT statements – it may take a long time, if the program calls the database for each statement. You can then gather all operations in a batch, which is possible for a *Statement*, a *PreparedStatement* and a *CallableStatement*, and then performs all operations at once with a single connection to the database. The following test method shows the syntax:

```
private static void test11()
{
 List<Person> list = createKings();
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
 {
  PreparedStatement stmt1 =
   conn.prepareStatement("DELETE FROM history WHERE name LIKE ?");
  PreparedStatement stmt2 = conn.prepareStatement("INSERT INTO history
   (name, title, birth, death, country, description) VALUES (?, ?, ?, ?, ?, ?)");
```

```
   DatabaseMetaData metaData = conn.getMetaData();
   if (metaData.supportsBatchUpdates())
   {
    for (Person pers : list)
    {
     stmt1.setString(1, pers.getName());
     stmt1.addBatch();
    }
    for (Person pers : list)
    {
     prepareStatement(stmt2, pers);
     stmt2.addBatch();
    }
    stmt1.executeBatch();
    stmt2.executeBatch();
   }
   else System.out.println("This database does not support batch updates");
  }
  catch (SQLException ex)
  {
   System.out.println(ex);
  }
}
```

createKings() is a method that, in principle, is identical to *createPersons()* (it creates just some other *Person* objects). Inside the try block the method creates two *PreparedStatement* objects respectively a DELETE and an INSERT. Since it is not a requirement that JDBC supports batch updates, is is first tested whether that is the case, and if so a loop of all the objects in the list, add statements to delete the corresponding rows, if they exist, but instead of performing *excecuteUpdate()* is performed an *addBatch()*, which adds that database operations to a list of operations in the object *stmt1*. Subsequently, the same happens for all objects in the list, but this time as *stmt2* objects. So far, the database operations are not performed, but it happens eventually with the method *executeBatch()*.

EXERCISE 4

The following lines show the country code, code for the continent, the country's area and size of population of seven countries:

```
AF;AS;652225;28150000
AL;EU;28748;3100112
DZ;AF;2381740;32531853
AS;NA;199;57902
US;NA;9826675;310322000
AD;EU;467.63;85458
AO;AF;1246700;18565269
```

You must write a program that updates the table *country* with the information above when

1. it must be done with a batch update
2. you must use a *Statement* object instead of a *PreparedStatement* object

Note that the method *addBatch()* may have a SQL expression as a parameter.

After the table is updated, you must perform a SQL SELECT that extracts the name of the countries in which the area is not NULL, and prints the names on the screen.

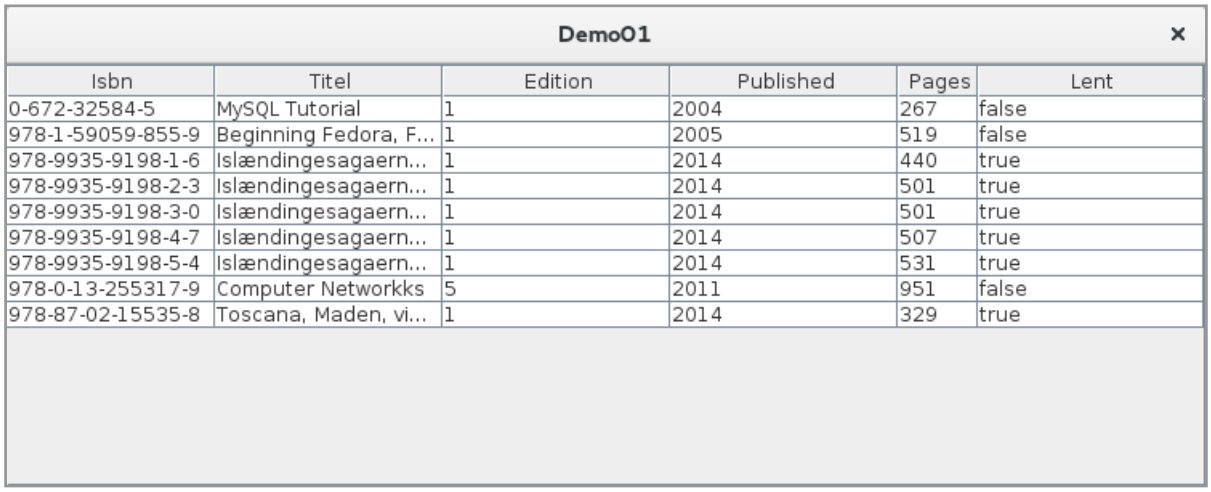
7 THE COMPONENT JTABLE

This chapter describes one of the most complex *Swing* components called *JTable*. In principle, it has not something with JDBC or databases to do, but the component is intended to display data organized in rows and columns, and therefore the component finds specific application to display the contents of database tables. This is the reason why it is referred to at this place.

It is a very important component, it is very flexible with many options and customizations, but it is equally complex and it is not always so easily to figure out how it works. The class name is as mentioned *JTable*, and first of all it is important to note, that a *JTable* object does not contains data, but it can display data defined by a data model. The class uses several (actually many) helper classes that are commonly found in the package *javax.swing.table*.

7.1 THE DEMO PROGRAM

I'll start simple, and below shows a window with a *JTabel* that shows some data laid out in a table with 6 columns and nine rows:



Isbn	Titel	Edition	Published	Pages	Lent
0-672-32584-5	MySQL Tutorial	1	2004	267	false
978-1-59059-855-9	Beginning Fedora, F...	1	2005	519	false
978-9935-9198-1-6	Islændingesagaern...	1	2014	440	true
978-9935-9198-2-3	Islændingesagaern...	1	2014	501	true
978-9935-9198-3-0	Islændingesagaern...	1	2014	501	true
978-9935-9198-4-7	Islændingesagaern...	1	2014	507	true
978-9935-9198-5-4	Islændingesagaern...	1	2014	531	true
978-0-13-255317-9	Computer Networkks	5	2011	951	false
978-87-02-15535-8	Toscana, Maden, vi...	1	2014	329	true

The individual data elements are not important, but they show data about 9 books. If you run the program, you must observe the following:

- You can change the window size. If you changes the height, so there is no room for the entire table, you get a scrollbar. If you changes the the width, the column widths are changed, but there is no scrollbar. However, the width of the two columns for ISBN and Pages do not change.
- It is possible to select multiple rows by clicking with the mouse, and there are the usual features that apply in combination with Ctrl and Shift keys.
- You can change the width of each column by dragging the divider between two column headers.
- You can edit the contents of the individual cells. You opens a cell for editing by double-clicking with the mouse.

There are so many attached functions to a *JTable*, and what is mentioned above is the default settings (except the two columns that have fixed width).

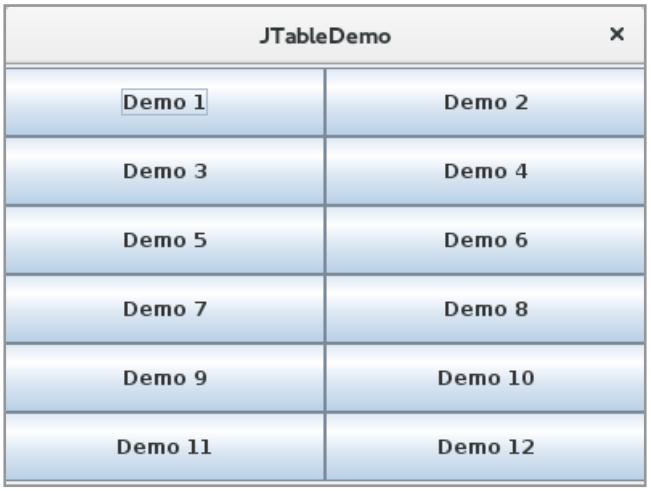
The code is as follows:

```
package jtabledemo;

import javax.swing.*;
import java.awt.*;
import javax.swing.table.*;
```

```
public class Demo01 extends JDialog
{
 private String[] colNames =
  { "Isbn", "Titel", "Edition", "Published", "Pages", "Lent" };
 private Object[][] data = {
  { "0-672-32584-5", "MySQL Tutorial", 1, 2004, 267, false },
  { "978-1-59059-855-9", "Beginning Fedora, From Novice to Professional", 1,
   2005, 519, false },
  { "978-9935-9198-1-6", "Islændingesagaerne bind I", 1, 2014, 440, true },
  { "978-9935-9198-2-3", "Islændingesagaerne bind II", 1, 2014, 501, true },
  { "978-9935-9198-3-0", "Islændingesagaerne bind III", 1, 2014, 501, true },
  { "978-9935-9198-4-7", "Islændingesagaerne bind IV", 1, 2014, 507, true },
  { "978-9935-9198-5-4", "Islændingesagaerne bind V", 1, 2014, 531, true },
  { "978-0-13-255317-9", "Computer Networkks", 5, 2011, 951, false },
  { "978-87-02-15535-8", "Toscana, Maden, vinen, kulturen & landskabet", 1, 2014,
   329, true }
 };

 public Demo01()
 {
  super(null, "Demo01", JDialog.ModalityType.APPLICATION_MODAL);
  setSize(750, 300);
  this.setLocationRelativeTo(null);
  setLayout(new BorderLayout());
  add(new JScrollPane(createTable()));
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  setVisible(true);
 }

 private JTable createTable()
 {
  JTable table = new JTable(data, colNames);
  setColumnWidth(table.getColumnModel().getColumn(0), 130);
  setColumnWidth(table.getColumnModel().getColumn(4), 50);
  return table;
 }

 private void setColumnWidth(TableColumn col, int width)
 {
  col.setPreferredWidth(width);
  col.setMinWidth(width);
  col.setMaxWidth(width);
 }
}
```

In the beginning of the code is laid out the data to be displayed. The first array is the column headings, while the next two-dimensional array is the table's data. You should note that the array's data is of the type *Object*, and since some of the data are simple data types there are used auto boxing for objects to type *Integer* and *Boolean*. The table is created in the method *createTable()* where the two data structures (arrays) are sent as parameters to the constructor in *JTabel* and the constructor will then based on these structures create a data model. Next is called the method *setColumnWidth()* for column 0 and column 4 to define a fixed width for these columns. When looking at the syntax, there is not much mystery in it, but it is far from a typical application, and take the many opportunities for user interaction into account is clear that there is a part that must be learned.

The class is part of a project called *JTableDemo* and when you run the program, you get the following window:



and thus a window with 12 buttons. Each button opens a dialog box, showing an example of a *JTable*, and the upper left is the example mentioned above.

THE DATA MODEL

I will now look at the data model as a *JTable* displays the objects in a data model. In the previous example it was the constructor of the class *JTable* that created the model, but typically it happens otherwise. A data model is a class that inherits *AbstractTableModel*, and a model for the same data as above could be:

```
package jtabledemo;

import javax.swing.table.*;

public class DemoDataModel extends AbstractTableModel
{
 private String[] colNames =
  { "Isbn", "Titel", "Edition", "Published", "Pages", "Lent" };
 private Object[][] data = {
  { "0-672-32584-5", "MySQL Tutorial", 1, 2004, 267, false },
  { "978-1-59059-855-9", "Beginning Fedora, From Novice to Professional", 1,
   2005, 519, false },
  { "978-9935-9198-1-6", "Islændingesagaerne bind I", 1, 2014, 440, true },
  { "978-9935-9198-2-3", "Islændingesagaerne bind II", 1, 2014, 501, true },
  { "978-9935-9198-3-0", "Islændingesagaerne bind III", 1, 2014, 501, true },
  { "978-9935-9198-4-7", "Islændingesagaerne bind IV", 1, 2014, 507, true },
  { "978-9935-9198-5-4", "Islændingesagaerne bind V", 1, 2014, 531, true },
  { "978-0-13-255317-9", "Computer Networkks", 5, 2011, 951, false },
  { "978-87-02-15535-8", "Toscana, Maden, vinen, kulturen & landskabet", 1, 2014,
   329, true }
 };
```

JAVA 6: JDBC AND DATABASE APPLICATIONS THE COMPONENT JTABLE

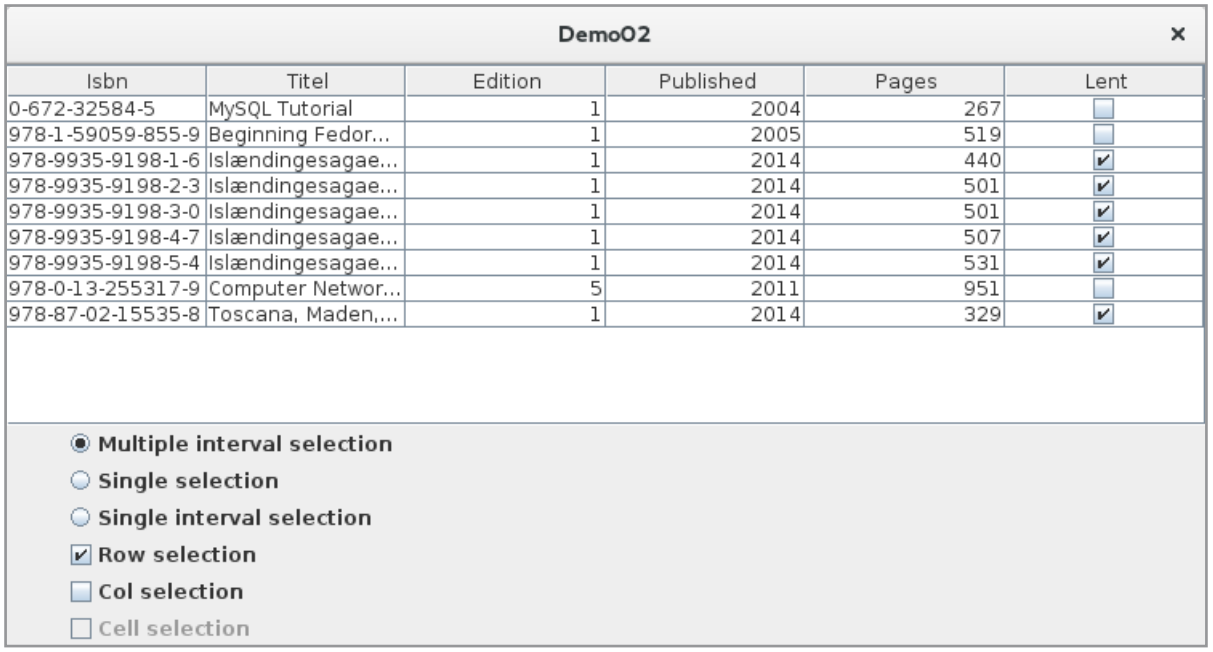
```
 public DemoDataModel()
 {
  super();
 }

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

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

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

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

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

 public boolean isCellEditable(int row, int col)
 {
  return col > 2;
 }

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

that in addition to the data definitions alone is overriden methods from the base class. In fact, only three of these methods are necessary (*getColumnCount(), getRowCount()* and *getValueAt()*) because the others have default implementations in *AbstractTableModel*. However *getColumnName()* is necessary to get the correct column names. The methods are generally simple, but you should note the method *isCellEditable()*, which in this case defines that only columns with index greater than 2 can be edited. Note also the method *getColumnClass()*, which returns the data type of each column. In general (as in the previous example), a JTable display the value of a datalement as a string in the form of *toString()*, but with the data type available, the cells will be rendered differently. For example the cells containing the numerical values show the content right justified, and as another example, a *Boolean* (the last column in this case) is displayed as a *JCheckBox* component.

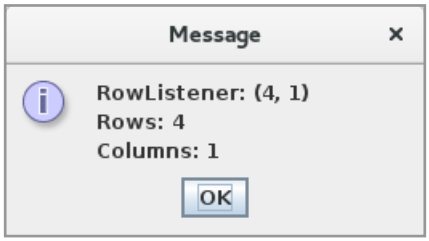
Below is a window showing a *JTable* on the basis of the above data model. The example should show two things:

1. how to apply a data model
2. how the selection of rows and columns in *JTabel* works

The first is very simple, while the selection of cells is a little more difficult to figure out.



In addition to the table, the window has some radio buttons and checkboxes, so the user can specify how the selection of cells works. If you click in the table, it displays a message box that tells what is clicked:



You are encouraged to run the program, so you get an idea of how the selection of cells works depending on the settings of the program's buttons.

The following method creates the table, and you must mainly observe how to map the data model, which is done with a parameter to the constructor:

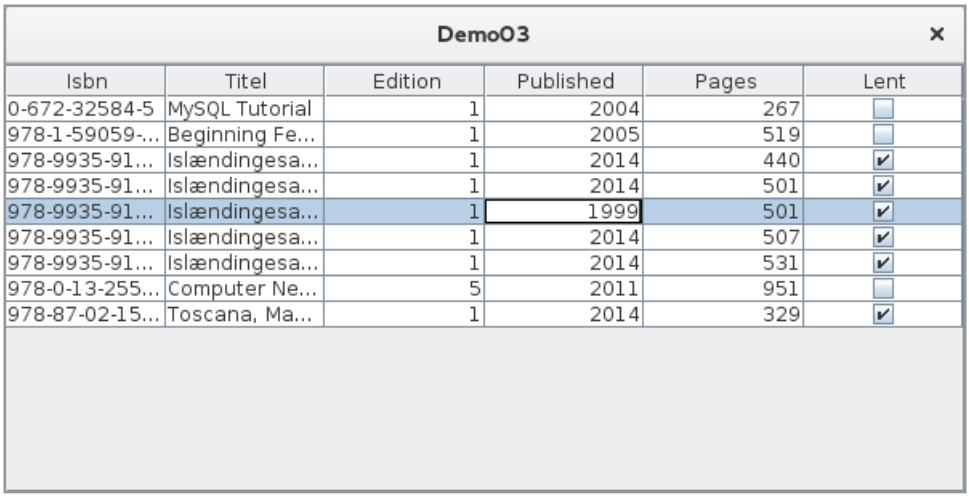
```
private JTable createTable()
{
 table = new JTable(new DemoDataModel());
 table.setFillsViewportHeight(true);
 table.getSelectionModel().addListSelectionListener(new RowListener());
 table.getColumnModel().getSelectionModel().addListSelectionListener(
  new ColumnListener());
 return table;
}
```

The next statement is not really necessary, but it means that a *JTable* component fills the part of the container that is not used for other components, and thus the white space between the table and the bottom components are part of the *JTable* component. For example, it could be important if the program uses drag and drop. The last two statements are related to event handlers, respectively selection of rows and columns, that among other things, opens a message box as shown above. The goal of this event handling is to show how to catch events concerning selection of rows and columns and including which cell is clicked.

The rest of the program code fills a part, but contains nothing new. A part of the code concerning the design of the user interface, where the components are laid out with a BoxLayout. Moreover, there are event handlers for each of the 6 buttons, and when the three check boxes is not independent of each other, the program must implement a logic that controls when a check box must be selected.

EDIT CELLS

As mentioned in the first example, you can immediately edit the contents of the cells (if permitted according to the data model, which means that in this case you can not edit the contents of the first three columns). It is an example of a window with a *JTabel* showing the same data model as in the previous example. Here you can edit a cell by double-clicking on it:

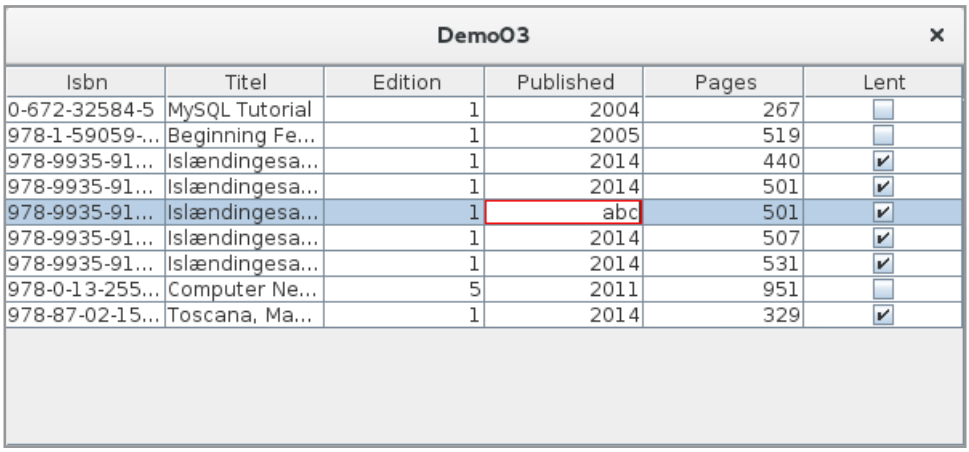


and if you then leave the cell (for example by pressing *Enter*), you get a message box that tells that the cell is changed:

The content of the cell should in this case (the model) be an integer, and if you enter something that is not an integer and press *Enter*, the cell will be selected, which shows that the content is illegal (see below).

This is the standard error handling, but I show later, how you can control the editing of cells, and what should happens if you enter something illegal.

With regard to the code there is not much to explain, and the most important is to show how to assign an event handler that captures changes to the table:



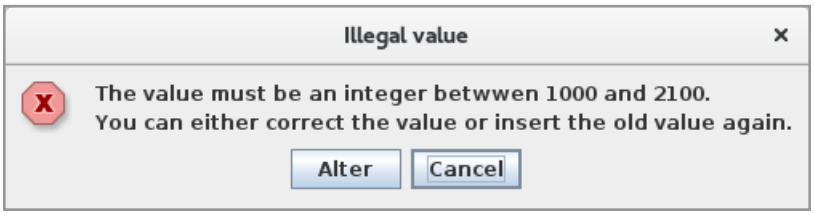

```
private JTable createTable()
{
 JTable table = new JTable(new DemoDataModel());
 table.getModel().addTableModelListener(new TableChangedListener());
 return table;
}

class TableChangedListener implements TableModelListener
{
 public void tableChanged(TableModelEvent e)
 {
  int row = e.getFirstRow();
  int col = e.getColumn();
  DemoDataModel model = (DemoDataModel)e.getSource();
  String name = model.getColumnName(col);
  Object value = model.getValueAt(row, col);
  JOptionPane.showMessageDialog(Demo03.this, name + ": " + value);
 }
}
```

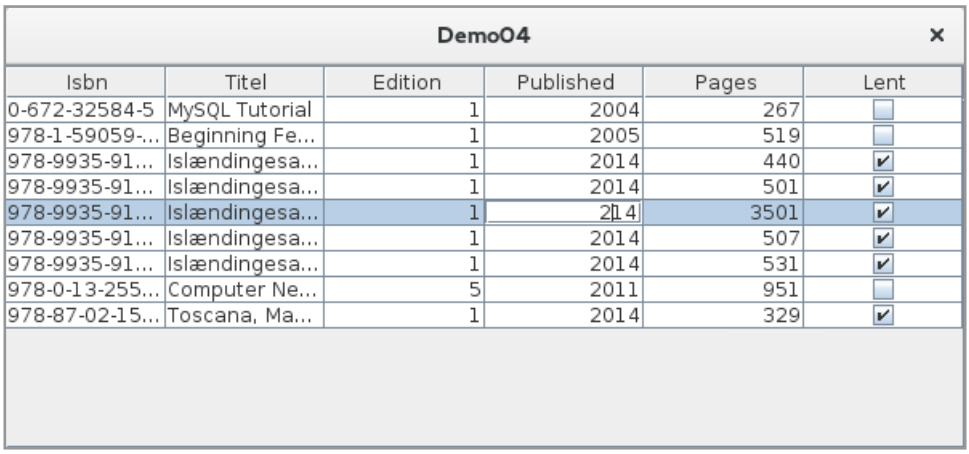
Here you particularly should note how to references the cell that is modified, including its content.

As mentioned, you can associates an editor for editing the individual cells, as you do in the following way:

```
table.getColumnModel().getColumn(3).setCellEditor(new IntegerEditor(1000, 2100));
```

where is assigned a *CellEditor* to the third column (the column of the year), which requires that the value must be an integer between 1000 and 2100. When you edit the content of a cell, the content of the cell are copied into a corresponding *JTextField* where you can edit it. In this case there is only attached a *CellEditor* to column 3, and so you can edit the content of the column 4 (the number of pages) as default. Below is a window where the page number is changed and the cell for a year is editing. Now if you press Enter, you get the following message box that says that you have entered an illegal value:





Isbn	Titel	Edition	Published	Pages	Lent
0-672-32584-5	MySQL Tutorial	1	2004	267	☐
978-1-59059-...	Beginning Fe...	1	2005	519	☐
978-9935-91...	Islændingesa...	1	2014	440	☑
978-9935-91...	Islændingesa...	1	2014	501	☑
978-9935-91...	Islændingesa...	1	214	3501	☑
978-9935-91...	Islændingesa...	1	2014	507	☑
978-9935-91...	Islændingesa...	1	2014	531	☑
978-0-13-255...	Computer Ne...	5	2011	951	☐
978-87-02-15...	Toscana, Ma...	1	2014	329	☑

If you here click *Alter*, you can continue to edit the cell, and if you click *Cancel*, the old value is inserted again.

A *CellEditor* is a class that inherits *DefaultCellEditor*, which in turn inherits *JTextField*. In this example, the editor is written as follows:

```
class IntegerEditor extends DefaultCellEditor
{
 JFormattedTextField field;
 NumberFormat format;
 private Integer min, max;

 public IntegerEditor(int min, int max)
 {
  super(new JFormattedTextField());
  field = (JFormattedTextField)getComponent();
```

```
 this.min = min;
 this.max = max;
 format = NumberFormat.getIntegerInstance();
 NumberFormatter formatter = new NumberFormatter(format);
 formatter.setFormat(format);
 formatter.setMinimum(min);
 formatter.setMaximum(max);
 field.setFormatterFactory(new DefaultFormatterFactory(formatter));
 field.setValue(min);
 field.setHorizontalAlignment(JTextField.TRAILING);
 field.setFocusLostBehavior(JFormattedTextField.PERSIST);
 field.getInputMap().put(KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0), "check");
 field.getActionMap().put("check", new CheckAction());
}

public Component getTableCellEditorComponent(JTable table, Object value,
 boolean isSelected, int row, int column)
{
 JFormattedTextField field =
   (JFormattedTextField)super.getTableCellEditorComponent(
    table, value, isSelected, row, column);
 field.setValue(value);
 return field;
}
```

```
public Object getCellEditorValue()
{
 JFormattedTextField field = (JFormattedTextField)getComponent();
 Object value = field.getValue();
 if (value instanceof Integer) return value;
 else if (value instanceof Number)
  return new Integer(((Number)value).intValue());
 else
 {
  try
  {
   return format.parseObject(value.toString());
  }
  catch (ParseException exc)
  {
   return null;
  }
 }
}

public boolean stopCellEditing()
{
 JFormattedTextField field = (JFormattedTextField)getComponent();
 if (field.isEditValid())
 {
  try
  {
   field.commitEdit();
  }
  catch (java.text.ParseException exc)
  {
  }
 }
 else
 {
  if (!revert()) return false;
 }
 return super.stopCellEditing();
}

protected boolean revert()
{
 Toolkit.getDefaultToolkit().beep();
 field.selectAll();
 Object[] options = {"Alter", "Cancel" };
 if (JOptionPane.showOptionDialog(SwingUtilities.getWindowAncestor(field),
  "The value .... ", "Illegal value", JOptionPane.YES_NO_OPTION,
```

```
    JOptionPane.ERROR_MESSAGE, null, options, options[1]) == 1)
   {
    field.setValue(field.getValue());
    return true;
   }
   return false;
  }

  class CheckAction extends AbstractAction
  {
   public void actionPerformed(ActionEvent e)
   {
    if (!field.isEditValid())
    {
     if (revert()) field.postActionEvent();
    }
    else
     try
     {
      field.commitEdit();
      field.postActionEvent();
     }
     catch (java.text.ParseException exc)
     {
     }
    }
  }
}
```

It is a comprehensive class, simply because it is complex to edit the content of a field. Essentially the following happens:

The class inherits *DefaultCellEditor* and thus specifically a *JTextField*. The constructor has two parameters, similar to that you have to edit a number within a range. The constructor starts to replace its editing component (which is a *JTextField*) with another *Swing* component called a *JFormattedTextField*, that is a *JTextField* that has an associated object, which can format the text entered. In this case it is a *NumberFormatter* object that is initialized with a *NumberFormat* object for formatting integers and the range within which the number must lie. Then is defined that the field should be right justified, and an event handler to the *Enter* key is added, such you returns from the field when you hit the *Enter* key.

The event handler is defined as an inner class. It begins by testing whether the content of the field is legal in relation to the formatter, and is it not the case, the method *revert()* is called. It is the method that displays the above message box where you can choose whether you want to change the entered value, or cancel and return to the old content. If you choose the last, the handler is terminated with a *postActionEvent()*. Is the content of the field legally, the result is marked as *commited*, and again finish with a *postActionEvent()*.

The class has three other methods. The first

```
getTableCellEditorComponent(JTable table, Object value, boolean isSelected,
  int row, int column)
```

returns the component (the input field) that is assigned to a particular cell and assign it a value, while the next

```
Object getCellEditorValue()
```

returns the object which is currently edited as an *Integer* object. Finally the last method that is performed when editing ends.

RENDERING CELLS

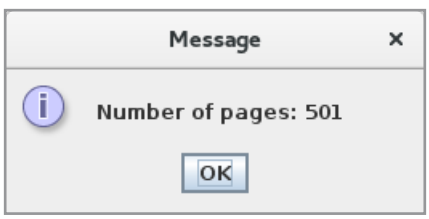
A *CellEditor* allows you to define how the content of cells in a *JTable* can be edited, but a column can also be associated with a *CellRenderer* that indicates how the content of the column's cells should appear. As already mentioned the default is as a string in which the cell shows an object as the result of its *toString()* method, but if a JTable is defined on the basis of a *TableModel* and implements the method *getColumnClass()*, it is instead the objects' data type, which determines how the values appear. For example becomes a *Boolean* type shown – or rendered – as a *JCheckBox*. How each column should render their cells values can also be defined with a *CellRenderer*, which is a class that defines how.

Demo05

Isbn	Titel	Edition	Published	Pages	Lent
0-672-32584-5	MySQL Tutorial	1	2004	Click	Available: ☐
978-1-59059-855-9	Beginning Fedora, From Novice to Professional	1	2005	Click	Available: ☐
978-9935-9198-1-6	Islændingesagaerne bind I	1	2014	Click	Available: ☑
978-9935-9198-2-3	Islændingesagaerne bind II	1	2014	Click	Available: ☑
978-9935-9198-3-0	Islændingesagaerne bind III	1	2014	Click	Available: ☑
978-9935-9198-4-7	Islændingesagaerne bind IV	1	2014	Click	Available: ☑
978-9935-9198-5-4	Islændingesagaerne bind V	1	2014	Click	Available: ☑
978-0-13-255317-9	Computer Networkks	5	2011	Click	Available: ☐
978-87-02-15535-8	Toscana, Maden, vinen, kulturen & landskabet	1	2014	Click	Available: ☑

Consider the above window, which has a *JTable* showing the same data model as above. Here you should note

- if you clicks on a cell in the column *Published*, you get a combo box from which you can select a date
- a cell in the column *Pages* appears as a button, and if you clicks the button, you get a message box as shown below
- the last column shows not only a check box, but also a text



The reason is that there is associated a *CellRenderer* for each of the last three columns. For the column *Published* the renderer is defined it as follows:

```
public void setYearColumn(JTable table, TableColumn column)
{
 JComboBox comboBox = new JComboBox();
 for (int i = 1950; i <= 2025; ++i) comboBox.addItem(i);
 column.setCellEditor(new DefaultCellEditor(comboBox));
 DefaultTableCellRenderer renderer = new DefaultTableCellRenderer();
 renderer.setToolTipText("Klik for at åbne dropdown boks");
 renderer.setHorizontalAlignment(JLabel.RIGHT);
 column.setCellRenderer(renderer);
}
```

This creates a usual *JComboBox* component that is initialized with a year between 1950 and 2025. It can then be directly assigned to the column as a *CellEditor*, which means that if you double-click a cell, the combo box is opend, so you can choose a date. In addition, there is created a *DefaultTableCellRenderer*, that has attached a tooltip. The *DefaultTableCellRenderer* is then associated the column as a *CellRenderer*.

Then there is the column *Pages* where the content should be displayed as a button. Here a renderer is defined as follows:

```
public void setPageColumn(JTable table, TableColumn column)
{
 column.setCellRenderer(new PageCells());
 column.setCellEditor(new PageCells());
}

class PageCells extends AbstractCellEditor
 implements TableCellEditor, TableCellRenderer
{
 public Component getTableCellEditorComponent
  (JTable table, Object value, boolean isSelected, int row, int column)
 {
  return table.getCellRenderer(row, column).ge
tTableCellRendererComponent(table,
   value, isSelected, isSelected, row, column);
 }

 public Object getCellEditorValue()
 {
  return null;
 }
```

```
 public Component getTableCellRendererComponent(JTable table, Object value,
    boolean isSelected, boolean hasFocus, int row, int column)
 {
  AbstractAction action = new AbstractAction("Klik")
  {
   public void actionPerformed(ActionEvent e)
   {
    JOptionPane.showMessageDialog(Demo05.this, "Antal sider: " + value);
   }
  };
  JPanel panel = new JPanel(new BorderLayout());
  panel.setBorder(new EmptyBorder(3, 3, 3, 3));
  panel.add(new JButton(action));
  return panel;
 }
}
```

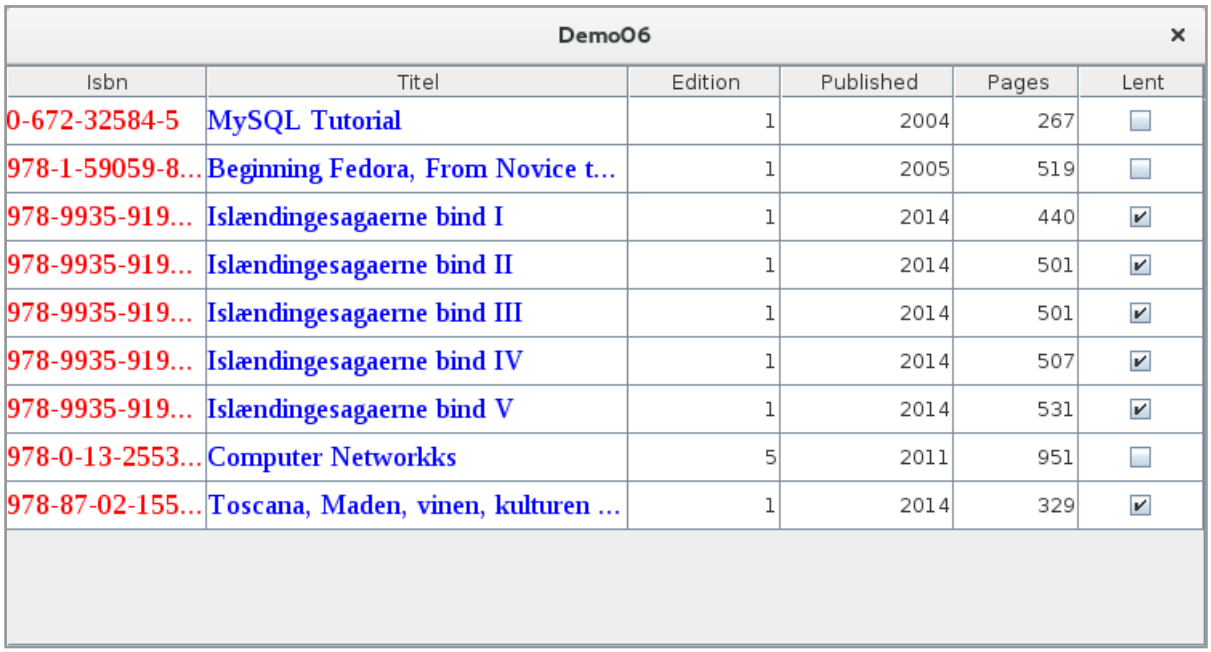
This time is defined an inner class. The class inherits *AbstratCellEditor* and implements interfaces to both a *CellEditor* and a *CellRenderer*. The method *getTableCellEditorComponent()* is not as important in this context and does nothing more than to return the current *CellEditor*, but the important things are happening in the method *getTableCellRendererComponent()*, which returns the component that should render the contents of the cell. It starts with defining an *Action*, which displays a message box with the value for number of pages. Next is created a *JPanel* with a button for this action, and the result is that the cell will contain a button, so you also can click the button. You should note that to be possible to click the button, the column must in the model be defined as *editable*.

The last column with the check box works the same way, and the difference is mainly that the panel created in *getTableCellRendererComponent()* this time contains two components. Since it should be possible to edit the content – put a check mark – the method *getCellEditorValue()* must this time return a value that must be initialized in the method *getTableCellEditorComponent()*.

The table is created as follows, with all the columns except the column for the title are assigned a fixed width:

```
private JTable createTable()
{
 JTable table = new JTable(new DemoDataModel());
 table.setRowHeight(35);
 setColumnWidth(table.getColumnModel().getColumn(0), 130);
 setColumnWidth(table.getColumnModel().getColumn(2), 60);
 setColumnWidth(table.getColumnModel().getColumn(3), 70);
 setColumnWidth(table.getColumnModel().getColumn(4), 80);
 setColumnWidth(table.getColumnModel().getColumn(5), 110);
 setYearColumn(table, table.getColumnModel().getColumn(3));
 setPageColumn(table, table.getColumnModel().getColumn(4));
 setCheckColumn(table, table.getColumnModel().getColumn(5));
 return table;
}

private void setColumnWidth(TableColumn col, int width)
{
 col.setPreferredWidth(width);
 col.setMinWidth(width);
 col.setMaxWidth(width);
}
```

As another example of a renderer displays the following window the first two columns with a different color:



The code is the following:

```
package jtabledemo;

import javax.swing.*;
import java.awt.*;
import javax.swing.table.*;

public class Demo06 extends JDialog
{
 public Demo06()
 {
  super(null, "Demo06", JDialog.ModalityType.APPLICATION_MODAL);
  setSize(750, 400);
  this.setLocationRelativeTo(null);
  setLayout(new BorderLayout());
  add(new JScrollPane(createTable()));
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  setVisible(true);
 }
```

```
 private JTable createTable()
 {
  JTable table = new JTable(new DemoDataModel());
  table.setDefaultRenderer(String.class, new ColorRenderer());
  table.setRowHeight(30);
  return table;
 }
}

class ColorRenderer extends JLabel implements TableCellRenderer
{
 public Component getTableCellRendererComponent(JTable table, Object value,
    boolean isSelected, boolean hasFocus, int row, int column)
 {
  setText(value.toString());
  setForeground(column == 0 ? Color.red : Color.blue);
  setFont(new Font("Liberation Serif", Font.PLAIN, 18));
  setToolTipText(column == 0 ? "ISBN is red" : "Title is blue");
  return this;
 }
}
```

The render object is defined by the class *ColorRenderer* and there is not much to explain, but you will notice how the render object is assigned in *createTable()*:

```
table.setDefaultRenderer(String.class, new ColorRenderer());
```

This means that all columns that contains objects of type *String* uses this renderer and thus the first two columns.

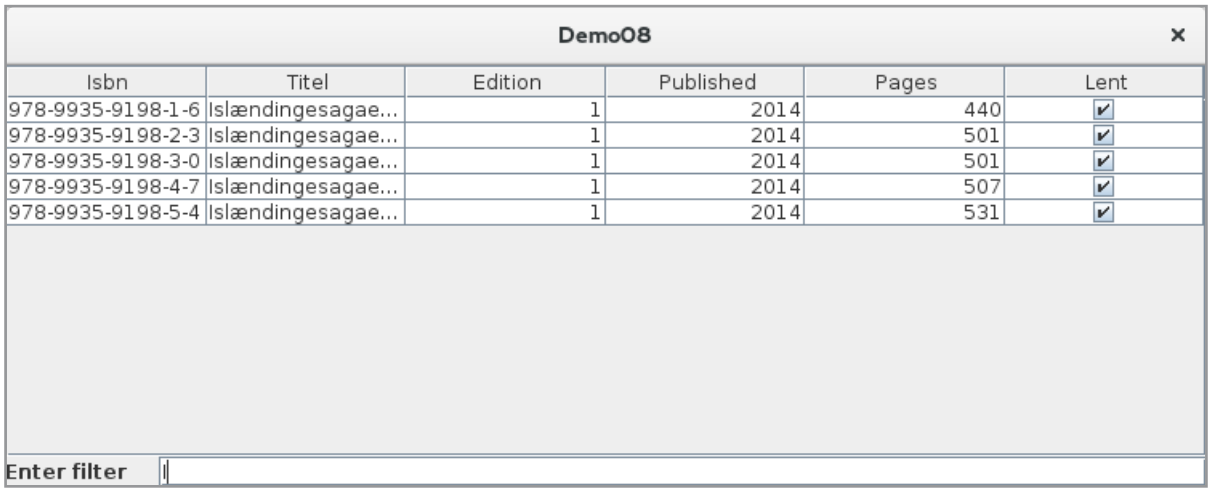
SORTING ROWS

A *JTable* supports sorting of rows by clicking the mouse on a column. Clicking once sorted rows of the column's values in ascending order, and click it again to sort in descending order. The example *Demo07* demonstrates how it works. The data model is the same as in the above examples, and the only thing that must happen is that you have to specify that it must be possible when the table is created:

```
private JTable createTable()
{
 JTable table = new JTable(new DemoDataModel());
 table.setAutoCreateRowSorter(true);
 return table;
}
```

FILTERS

This example shows how to assign a filter. The data model is again the same with 9 rows, but if you enter something in the input field below the table, only the rows where the title starts with what is entered are shown. Below I has entered a large I:



The code is as follows:

```
public class Demo08 extends JDialog
{
 private JTextField filterText;
 private TableRowSorter<DemoDataModel> sorter;

 public Demo08()
 {
  super(null, "Demo08", JDialog.ModalityType.APPLICATION_MODAL);
  setSize(750, 300);
  this.setLocationRelativeTo(null);
  setLayout(new BorderLayout());
  add(new JScrollPane(createTable()));
  add(createBottom(), BorderLayout.SOUTH);
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  setVisible(true);
 }

 private JTable createTable()
 {
  DemoDataModel model = new DemoDataModel();
  sorter = new TableRowSorter(model);
  JTable table = new JTable(model);
  table.setRowSorter(sorter);
  return table;
 }

 private JPanel createBottom()
 {
  JPanel panel = new JPanel(new BorderLayout(20, 0));
  panel.add(new JLabel("Enter filter"), BorderLayout.WEST);
  panel.add(filterText = new JTextField());
  filterText.getDocument().addDocumentListener(new Filter());
  return panel;
 }

 private void newFilter()
 {
  RowFilter<DemoDataModel, Object> filter = null;
  try
  {
   filter = RowFilter.regexFilter(filterText.getText(), 1);
  }
  catch (java.util.regex.PatternSyntaxException ex)
  {
```

```
    return;
   }
   sorter.setRowFilter(filter);
  }

  class Filter implements DocumentListener
  {
   public void changedUpdate(DocumentEvent e)
   {
    newFilter();
   }

   public void insertUpdate(DocumentEvent e)
   {
    newFilter();
   }

   public void removeUpdate(DocumentEvent e)
   {
    newFilter();
   }
  }
}
```

The first thing you should notice is that there are defined two instance variables, the first refers to the input field, while the other has the type *TableRowSorter* and is for the used data model (that is *DemoDataModel*). In the constructor is nothing new, but the method *createTable()* creates the *sorter* object and attached it to the table. The method *createBottom()* creates the input field, and associates the filter to the field the following statement:

```
filterText.getDocument().addDocumentListener(new Filter());
```

where *Filter* is an inner class that implements an interface called *DocumentListener*. This interface defines three methods that are executed when the content of the input field is changed, and therefore if the user enters or deletes anything. In all three cases, the methods called *newFilter()*, and it is the method it is all about. It creates a filter as a regular expression, and note especially that by an index indicating which column the filter must apply to – in this case, index 1 and thus the title column. Finally the *sorter* object is used to the filter the table rows.

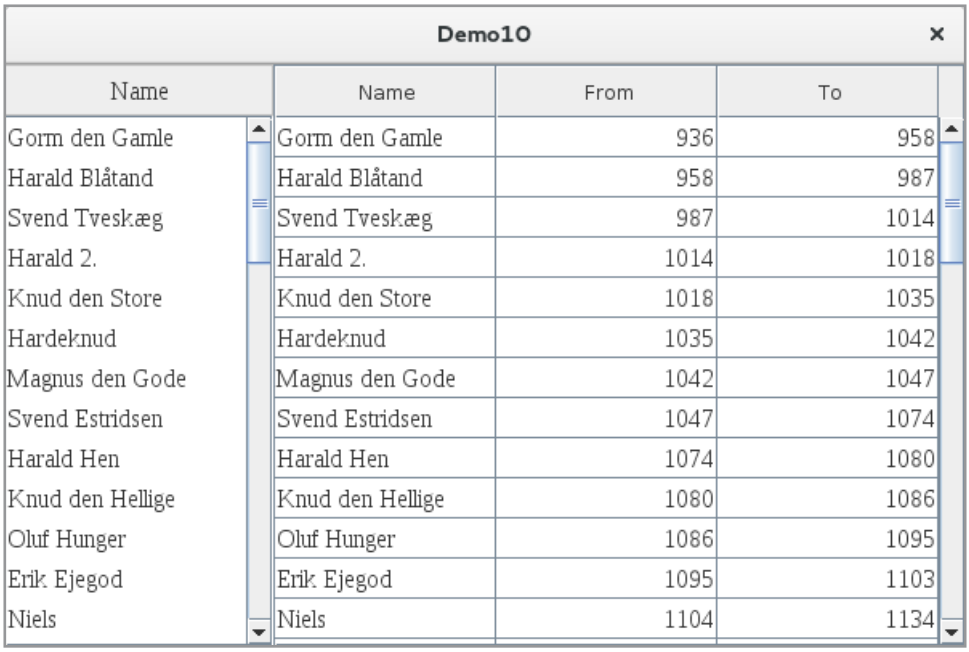
PRINT A JTABLE

The next example *Demo09* is simple and shows the same table as above, but the example demonstrates how to print the table of the printer, and it could hardly be simpler. The window has a *Print* button, and all that is needed is done in the event handler for this button:

```
private void print(ActionEvent e)
{
 MessageFormat header = new MessageFormat("Side {0, number, integer}");
 try
 {
  table.print(JTable.PrintMode.NORMAL, header, null);
 }
 catch (java.awt.print.PrinterException ex)
 {
  JOptionPane.showMessageDialog(this, ex.getMessage());
 }
}
```

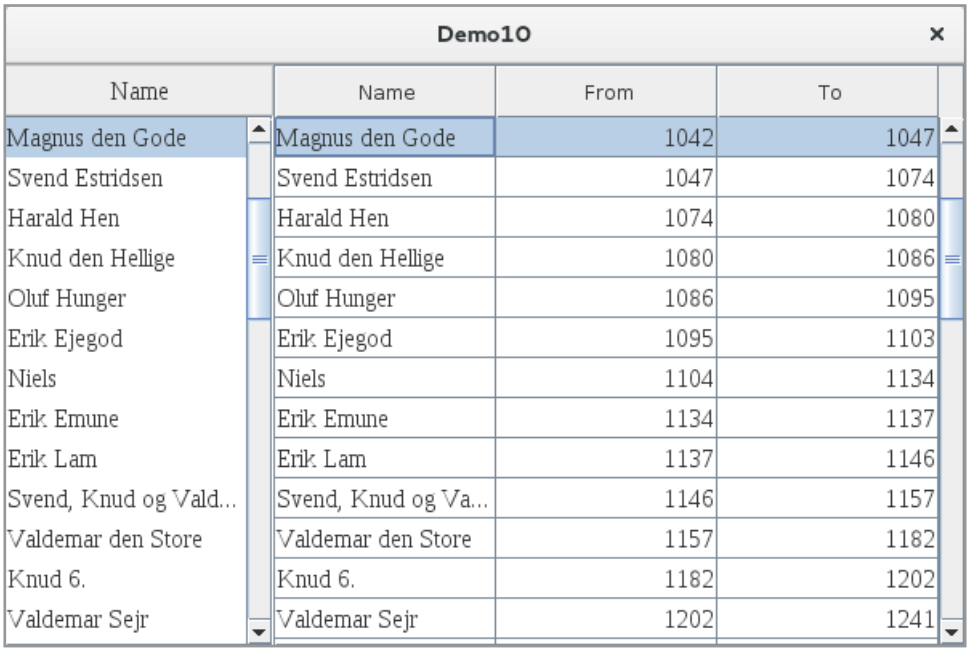
SYNCHRONIZING TWO TABLES

The next example shows a *JTable* and *JLis*t, and both using the the same data model, but this time a different data model, there is a list of Danish kings:



Both components thus uses the same data model, and the most important thing about the model is that there are so many rows, that you needs to scroll the table. The one component shows only one of the model's columns, while the second shows all columns. The example should show that if you click on a row in one of the two components they both scrolls, so the row that is selected appears at the top (se below).

The data model is called *Kings* and is relatively complex. It is defined as a class derived from *DefaultListModel*, but really it should be an *AbstractTableModel* (see examples in front). However, it is not necessary, but it is sufficient to implement the interface *TableModel* (what *AbstractTableModel* do). In return, the class *Kings* must implement all methods defined by *TableModel*. This is possible by defining an object of the type *KingsModel* that is an inner class that inherits *AbstractTableModel*, and the class *Kings* may then delegates the methods of the class *KingsModel*. You are encouraged to study the in principle simple, but a little unusual implementation of the data model.


Demo10
Name
Name
From
To
Magnus den Gode
Svend Estridsen
Harald Hen
Knud den Hellige
Oluf Hunger
Erik Ejegod
Niels
Erik Emune
Erik Lam
Svend, Knud og Vald...
Valdemar den Store
Knud 6.
Valdemar Sejr
Magnus den Gode 1042 1047
Svend Estridsen 1047 1074
Harald Hen 1074 1080
Knud den Hellige 1080 1086
Oluf Hunger 1086 1095
Erik Ejegod 1095 1103
Niels 1104 1134
Erik Emune 1134 1137
Erik Lam 1137 1146
Svend, Knud og Va... 1146 1157
Valdemar den Store 1157 1182
Knud 6. 1182 1202
Valdemar Sejr 1202 1241

Then there is the code of the user interface:

```
public class Demo10 extends JDialog
{
 private Font font = new Font("Liberation Serif", Font.PLAIN, 16);
 private Kings dataModel = new Kings();
 private JList list;
 private JTable table;
 private ListSelectionModel listSelectionModel;

 public Demo10()
 {
  super(null, "Demo10", JDialog.ModalityType.APPLICATION_MODAL);
  setSize(600, 400);
  this.setLocationRelativeTo(null);
  setLayout(new BorderLayout());
  add(createList(), BorderLayout.WEST);
  add(new JScrollPane(createTable()));
  setDefaultCloseOperation(DISPOSE_ON_CLOSE);
  setVisible(true);
 }

 private JPanel createList()
 {
  JPanel panel = new JPanel(new BorderLayout());
  JLabel label = new JLabel(dataModel.getColumnName(0), JLabel.CENTER);
  label.setPreferredSize(new Dimension(0, 32));
  label.setBorder(new LineBorder(Color.LIGHT_GRAY));
  label.setFont(font);
  panel.add(label, BorderLayout.NORTH);
  list = new JList(dataModel);
  list.setFont(font);
  list.setCellRenderer(new DefaultListCellRenderer()
  {
   public Component getListCellRendererComponent(JList list, Object value,
    int index, boolean selected, boolean focus)
   {
    King king = (King)value;
    Component component = super.getListCellRendererComponent(
     list, king.getName(), index, selected, focus);
    component.setPreferredSize(new Dimension(150, 25));
    return component;
   }
  });
  listSelectionModel = list.getSelectionModel();
  listSelectionModel.addListSelectionListener(new ListSelectionHandler());
  panel.add(new JScrollPane(list));
  return panel;
 }
```

```
  private JTable createTable()
 {
  table = new JTable(dataModel);
  table.setSelectionModel(listSelectionModel);
  JTableHeader header = table.getTableHeader();
  header.setPreferredSize(new Dimension(0, 32));
  table.setFont(font);
  table.setRowHeight(25);
  return table;
 }

 class ListSelectionHandler implements ListSelectionListener
 {
  public void valueChanged(ListSelectionEvent e)
  {
   int index = e.getSource() ==
    table ? table.getSelectedRow() : list.getSelectedIndex();
   list.ensureIndexIsVisible(index);
   Rectangle rect = table.getCellRect(index, 0, true);
   table.scrollRectToVisible(rect);
   table.scrollRectToVisible(list.getCellBounds(
    index, dataModel.getSize() - 1));
   list.scrollRectToVisible(list.getCellBounds(index, dataModel.getSize() - 1));
  }
 }
}
```

The components are placed in the window in the constructor. The *JList* component is created in the method *createList()*, where it is placed in a panel with a label above. Next the list is assigned a new *CellRenderer* object created on the basis of an anonymous class that inherits *DefaultListCellRenderer*, and does not much, but overrides the method *getListCellRendererComponent()*. The list's data model is *kings*, and the method has to ensure that only the name is shown and the cell has a preferred size to fit the height of the rows in the *JTable* component. Next the variable *listSelectionModel* is set to refer to list box's *SelectionModel*, and to this is attached an event handler of the type *ListSelectionHandler* (that is an inner class).

Then there is the method *createTabel()*, and here you mainly notice how to defines the height of the table's header and rows. Finally, note that the table is assigned to the same *SelectionModel* as the list box. This means that it is the same event handler that is executed if you click on one of the two components, and it is this handler that scrolls the two components so that the element that is clicked, is at the top.

JTABLE AND DATABASE TABLES

The example Demo11 shows how to display the content of a database table with a *JTable*. The example opens the window below. The window shows a *JTable* with the content of the table *zipcode*. At the bottom of the window, there is a button and two input fields. The input fields act as filters for the two columns, while button clears the fields. Regarding the definition of the table and window and including the filters there is nothing new, so it is the data model you need to be interested, and here is actually not very new:

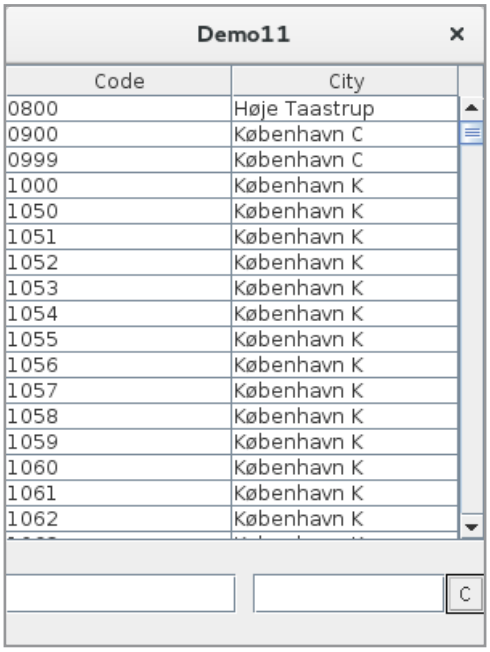
```
package jtabledemo;

import java.sql.*;
import java.util.*;
import javax.swing.table.*;

public class Zipcodes extends AbstractTableModel
{
 private List<Zipcode> list = new ArrayList();

 public Zipcodes()
 {
  try (Connection conn = DriverManager.getConnection(
```

```
   "jdbc:mysql://localhost:3306/padata?useSSL=false", "pa", "Volmer_1234"))
  {
   Statement stmt = conn.createStatement();
   ResultSet res = stmt.executeQuery("SELECT * FROM zipcode");
   while(res.next())
    list.add(new Zipcode(res.getString("code"), res.getString("city")));
  }
  catch (SQLException ex)
  {
   System.out.println(ex);
  }
}

public int getColumnCount()
{
 return 2;
}

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

public String getColumnName(int col)
{
 return col == 0 ? "Code" : "City";
}

public Object getValueAt(int row, int col)
{
 return col == 0 ? list.get(row).getCode() : list.get(row).getCity();
}

public Class getColumnClass(int c)
{
 return String.class;
}

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

```
 public void setValueAt(Object value, int row, int col)
 {
  if (col == 0)
   list.get(row).setCode((String)value);
  else
   list.get(row).setCity((String)value);
  fireTableCellUpdated(row, col);
 }
}
```

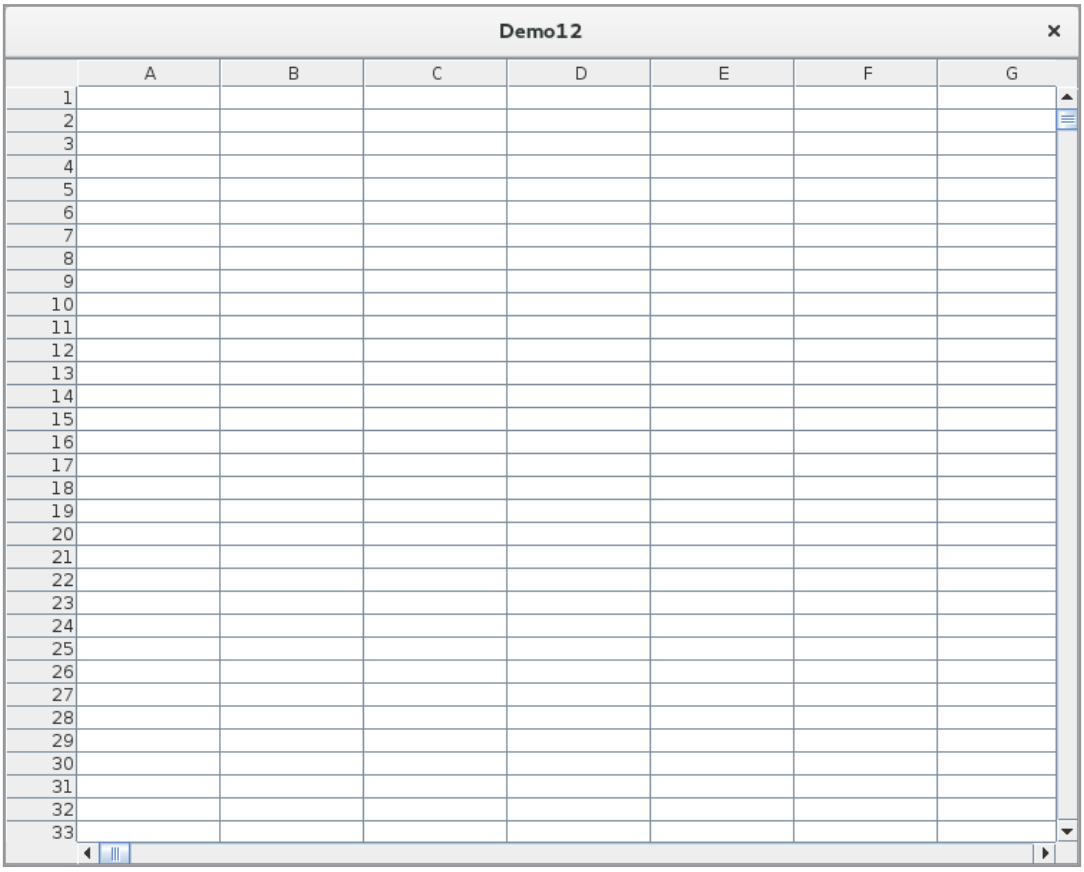
The only place where is something new is in the constructor, and it is nothing more than the initial examples in this book has illustrated, that is how to read a database table.

The conclusion is that it is quite simple to display the content of a database table using a *JTable* – you just need to initialize the data model with the content of the database table.



BIG TABLES

The last example will opens the following window:



The window shows a *JTable*, and there are 10000 rows and 200 columns and thus 2 million cells. The example will primarily show that a *JTabel* can have many cells, and it is still effective. You can also edit the individual cells except the last row and the last column. In the cells you can enter numbers, and the bottom row shows all the time the column sum, while the last column shows all the time the row sum.

The data model is simple and adds nothing new:

```
class SpreadsheetModel extends AbstractTableModel
{
 private Object[][] data;

 public SpreadsheetModel(int rows, int cols)
 {
  data = new Object[rows + 1][cols + 1];
 }
```

```
public int getColumnCount()
{
 return data[0].length;
}

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

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

public Class getColumnClass(int c)
{
 return Double.class;
}
```

```
 public boolean isCellEditable(int row, int col)
 {
  return col < data[0].length - 1 && row < data.length - 1;
 }

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

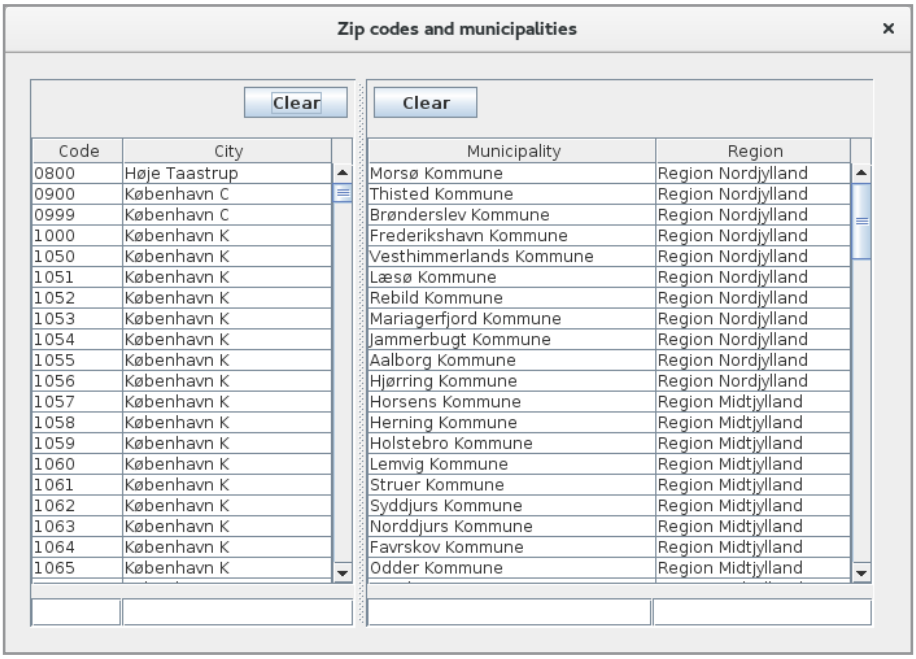
You should notice how it is defined that to the last row and last column must not be edited, while all other cells must. You should also notice that each column contains objects of the type *Double* and thus numbers.

Then there is the dialog box, where there is more to note. I will not show the code here, partly it fills a lot, and secondly, it is mainly something as discussed earlier in this chapter, so I'll just mention a few things that you need to notice when you study the code:

1. There are two defined constants which define the table size. You could try to experiment and change these constants.
2. How to define a *RowHeader* that is encapsulated in a *JViewPort* that is inserted in the *JScrollPane* that contains the *JTable* component.
3. An inner class *CellEditorHandler* that is event handler for editing a cell
4. A class *DoubleEditor*, that is a *CellEditor*.
5. A class *RowNumberHeader*, that is a *JTable* and defines the type of table's *RowHeader*.

PROBLEM 2

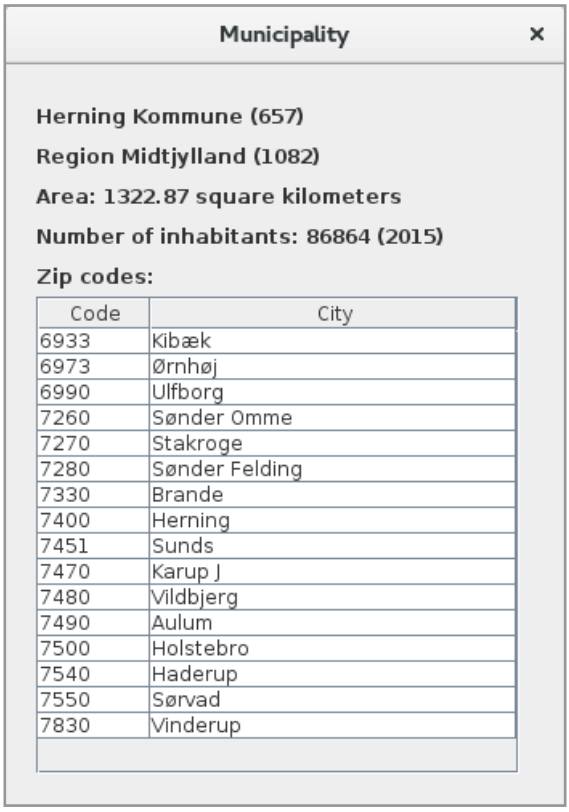
You must write a program that you can call the *Denmark*, which opens the following window:



The window shows two *JTable* components located in a *JSplitPane*. The table on the left shows all rows in the table *zipcode* while the table on the right shows all rows in the table *municipality*, but only the municipality's name and the name of the region. During the two tables, there are four input fields used for filters, to the above column. The two buttons at the top is used to clear the filters.

If you double-click on a line in the table with the zip codes, you should get the following window:

that for the zip code displays the names of the municipalities that uses this zip code. If you double-click a line in the table with municipalities, you must get a window as shown below, where you partly receive the municipality's area and number of inhabitants, and partly a table with the zip codes that this municipality uses. This table (the table with local zip codes) must be a *JTable* component.


Municipality
Herning Kommune (657)
Region Midtjylland (1082)
Area: 1322.87 square kilometers
Number of inhabitants: 86864 (2015)
Zip codes:
Code
City
6933 Kibæk
6973 Ørnhøj
6990 Ulfborg
7260 Sønder Omme
7270 Stakroge
7280 Sønder Felding
7330 Brande
7400 Herning
7451 Sunds
7470 Karup J
7480 Vildbjerg
7490 Aulum
7500 Holstebro
7540 Haderup
7550 Sørvad
7830 Vinderup

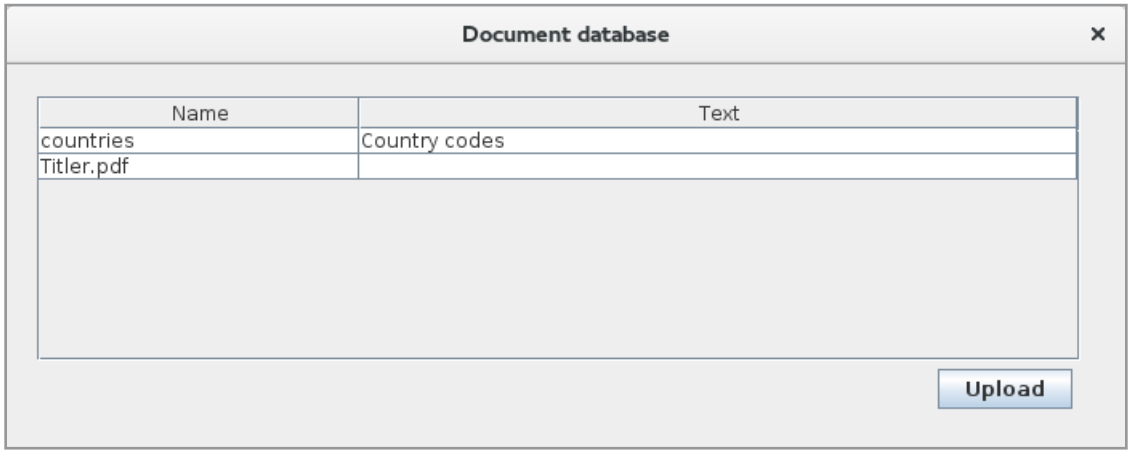
8 FILES IN DATABASES

It is also possible to save files in databases, for example PDF documents, XML documents or images. It has limited use, but with web applications one sometimes sees examples where images are stored in databases, and if these are small files, there may also be situations where it is ok. In this section I will show how to do that.

A column that must contain a file must be of the type TEXT (for documents) or BLOB (binary files like images). The following script (*CreateFileDb.sql*) creates a database with two tables, one table to documents, while the other is for images:

```
use sys;
drop database if exists filedb;
create database filedb;
use filedb;

create table documents (
 filename            varchar(100),       # the documents name
 description         varchar(100),       # a short description
 content             longtext,           # the documents content
 primary key(filename)
);

create table pictures (
 filename            varchar(100),       # the filename
 description         varchar(100),       # a short description
 content             longblob,           # picture data
 primary key(filename)
);
```

The following program is called *UploadDocs* and opens the following window:



When you click the *Upload* button, you get a standard dialog where you can browse file systems and find the file that you want to store in the database. If you double-click at a line in the table, you get a similar dialog box where you can browse to a folder, and the program must then save the content of the database row as a file in that folder.

The event handler for the button is:

```
private void upload(ActionEvent e)
{
 JFileChooser fc = new JFileChooser();
 fc.setCurrentDirectory(new File(System.getProperty("user.home")));
 if (fc.showOpenDialog(this) == JFileChooser.APPROVE_OPTION)
 {
  File file = fc.getSelectedFile();
  String description = JOptionPane.showInputDialog(this, "Description:",
   "Enter text", JOptionPane.PLAIN_MESSAGE);
  streamFile(file, description);
 }
}

private void streamFile(File file, String description)
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/filedb?useSSL=false", "pa", "Volmer_1234"))
 {
  long length = file.length();
  FileInputStream in = new FileInputStream(file);
  PreparedStatement stmt =
   conn.prepareStatement("INSERT INTO documents VALUES(?, ?, ?)");
  stmt.setString(1, file.getName());
  if (description == null) stmt.setNull(2, java.sql.Types.VARCHAR);
  else stmt.setString(2, description);
  stmt.setAsciiStream(3, in, length);
  stmt.executeUpdate();
  model.add(new Document(file.getName(), description));
 }
 catch (Exception ex)
 {
  JOptionPane.showMessageDialog(this, file.getAbsolutePath() +
   " could not be streamed to the database\n" + ex,
   "Error message", JOptionPane.ERROR_MESSAGE);
 }
}
```

The event handler creates a *JFileChooser* object and open the dialog box so that you are placed in the user's home directory. If the user selects a file, the user should enter a description of the document (the file). Then the method *streamFile()* is called, that streams the file to the database. This method opens a *FileInputStream* to the file and then a connection to the database and initialize a *PreparedStatement*. The only thing to note is the fact that the column 3 (the third parameter in *stmt*) is initialized with the method *setAsciiStream()* with the *InputStream* as a parameter.

You should note that the program does not try to validate the file, and in fact the program can upload any file and store it in the database – even a picture.

If you double-click a line in the table, the following event handler is performed:

```
class MouseHandler extends MouseAdapter
{
 public void mousePressed(MouseEvent e)
 {
  try
  {
   if (e.getClickCount() == 2)
   {
```

```
     int row = table.getSelectedRow();
     Document doc = model.getDocument((Integer)table.getValueAt(row, 0));
     JFileChooser fc = new JFileChooser();
     fc.setCurrentDirectory(new File(System.getProperty("user.home")));
     fc.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);
     if (fc.showSaveDialog(MainView.this) == JFileChooser.APPROVE_OPTION)
      streamFile(fc.getSelectedFile(), doc);
    }
   }
   catch (Exception ex)
   {
    JOptionPane.showMessageDialog(MainView.this,
     "could not be streamed trom the database\n" + ex,
     "Error message", JOptionPane.ERROR_MESSAGE);
   }
  }
}
```

There is not so much new, but the user will be able to select the directory to where the file should be saved. If a directory is selected the method *streamFile()* is called, which creates the file, and retrieves the content from the database:

```
private void streamFile(File file, Document doc) throws Exception
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306/filedb?useSSL=false", "pa", "Volmer_1234"))
 {
  PreparedStatement stmt =
   conn.prepareStatement("SELECT content FROM documents WHERE filename = ?")
  stmt.setString(1, doc.getFilename());
  ResultSet res = stmt.executeQuery();
  if (res.next())
  {
   InputStream stream = res.getAsciiStream (1);
   BufferedInputStream in = new BufferedInputStream(stream);
   String filename = file.getAbsolutePath() + "/" + doc.getFilename();
   BufferedOutputStream out =
    new BufferedOutputStream(new FileOutputStream(filename));
   int c;
   while ((c = in.read ()) != -1) out.write(c);
   out.close();
  }
 }
}
```

Again, there is not much new to explain. The method creates a *PreparedStatement* with one parameter, which is value to the primary key. The SELECT statement defines only a single column, as the column of file's content. If the row is found the content in the column is referenced to with the method *getAsciiStream()*, which is an *InputStream*. It is encapsulated in a *BufferedInputStream* (of performance reasons), and then is created a *BufferedOutputStream* that represents the file to which the database content must be streamed.

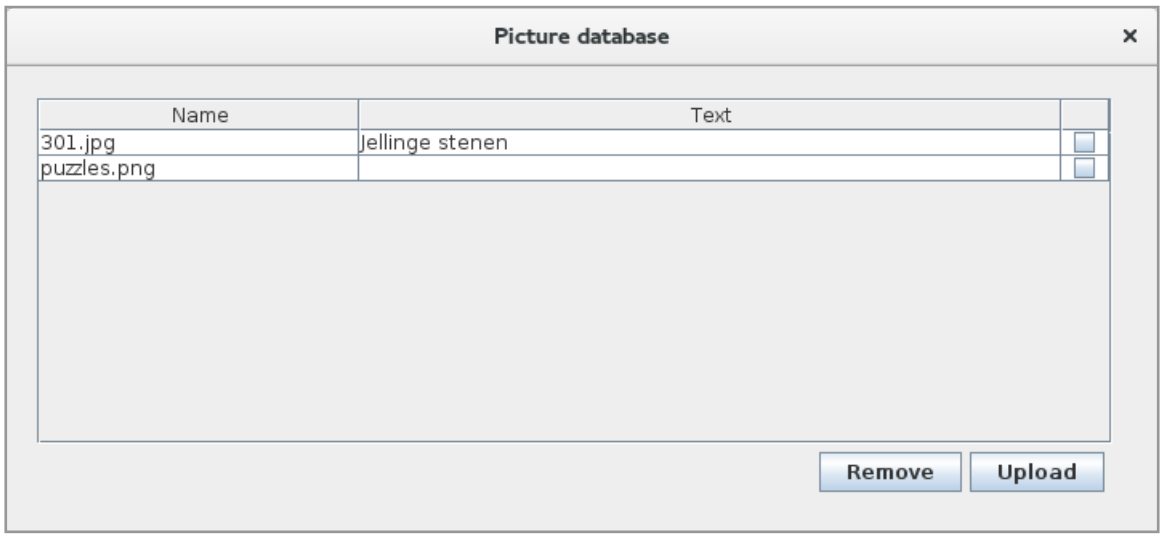
In the example, the data streaming to and from the database is performed with an *AsciiStream* that treat data as ASCII bytes encoded as ISO-Latin-1. There are other options:

- *UnicodeStream*, where the data is treated as 16-bits unicode
- *BinaryStream*, where the data is treated as raw bytes

and the difference is, what services these types provides.

EXERCISE 5

You must write a program similar to the above, and the program should open a window, as shown below:



The program must upload pictures to the *filedb* database and store the pictures in the table *pictures*, but in contrast to the above program it must use a *BinaryStream*. It should only be possible to upload *jpg*, *gif* and *png* files. If you double-click a line in the table, the corresponding file should in the same manner as in the above program be copied to a directory in the file system.

The table's right column consists of check boxes. If you click on the *Remove* button, the images selected must be removed from the database.

9 DDL COMMANDS

If you have to create a database and including its tables, you will in practice usually write a SQL script. When a script consists of SQL commands, these commands can of course also be performed using JDBC, and I close this explanation of JDBC with an example that shows how to create a database and a table in the database. Actually, it's just to, and the following test method creates a database:

```
private static void test12()
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306?useSSL=false", "pa", "Volmer_1234"))
 {
  Statement stmt = conn.createStatement();
  stmt.executeUpdate("CREATE DATABASE contacts");
 }
 catch(SQLException ex)
 {
  System.out.println(ex);
 }
}
```

The next method creates a table in this database:

```
private static void test13()
{
 try (Connection conn = DriverManager.getConnection(
  "jdbc:mysql://localhost:3306?useSSL=false", "pa", "Volmer_1234"))
 {
  Statement stmt = conn.createStatement();
  stmt.execute("use contacts");
  stmt.executeUpdate("CREATE TABLE persons (
   id int auto_increment not null primary key,
   name varchar(50) not null, phone varchar(20), email varchar(50) )");
 }
 catch(SQLException ex)
 {
  System.out.println(ex);
 }
}
```

and then it should be clear that all DDL commands immediately can be executable from a Java program – if you have the rights to perform the commands.

10 FINAL EXAMPLES

I will finish this book about JDBC and databases with two examples of database applications. The two examples are similar and are classic examples of database applications and in addition to the size in which the first is a small program, while the other is slightly larger, so the difference is that the first use an existing database, while the latter shows the development of a database program right from the start, which also includes the design and implementation of the database.

10.1 WORLD

In the exercises and problems in this book you have expanded the database *padata* with three tables

- *world*
- *country*
- *currency*

The task now is to write a program that can maintain these tables, and thus a program where you can search the tables, as well as edit and delete data. The program must have a graphical user interface.

THE REQUIREMENTS

Before I will address the development, the requirements must be defined and therefore which functions the program must be able to perform.

1. The program should open a window (the main view) with two tables, one table provides an overview of all countries, while the second shows an overview of all currencies.
2. It should be possible to create a new country.
3. It should be possible to edit the information about a country.
4. It must be possible to delete a country.
5. It must be possible to create a new currency.
6. It should be possible to edit information about a currency and here especially the exchange rate.
7. It must be possible to delete a currency, if there is no country that uses this currency.
8. It should be possible to update exchange rates with data from a CSV file.

It has been decided that you should not be able to maintain the table *world* with information about the world's continents.

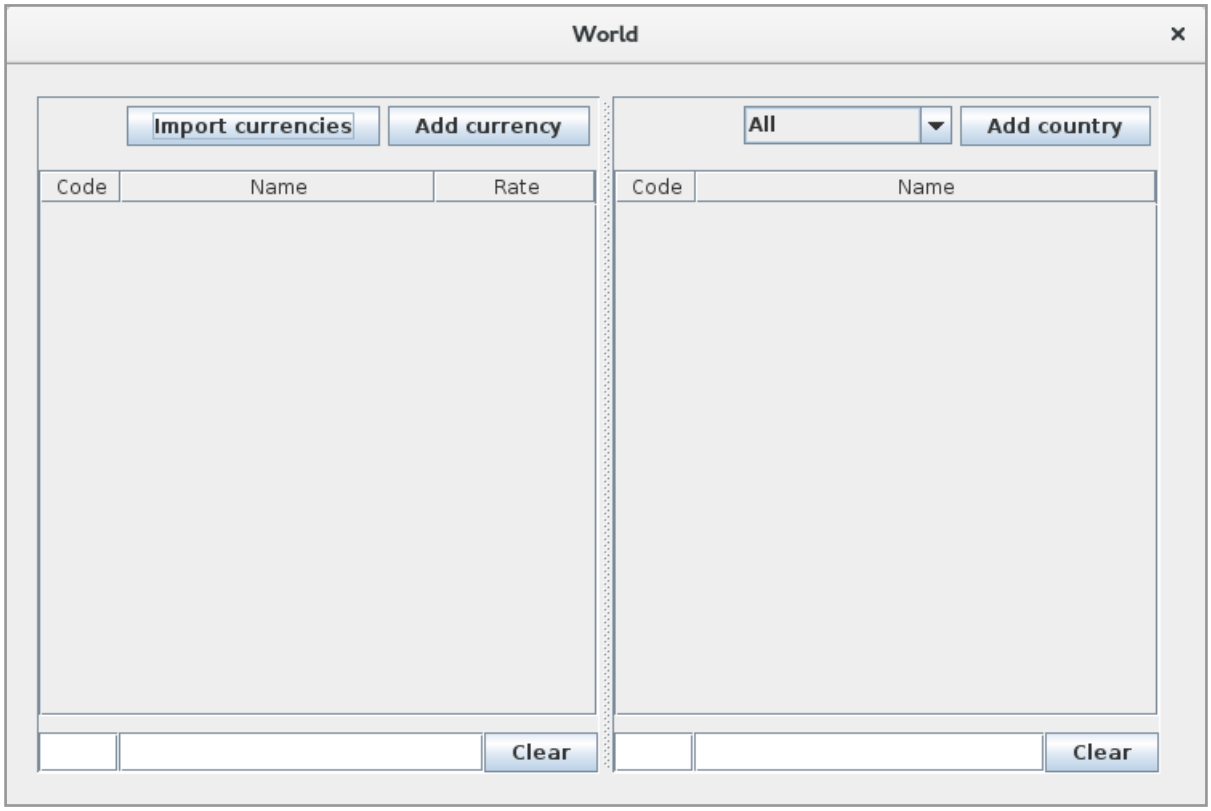
Except for the first and the last function the requires does not needs further explanation. With regard to the main view I will define it a prototype. As regards the last function, a CSV file must consist of lines of the form

```
code;rate[;name]
```

and there must at least be a currency code and a currency exchange rate om each line. If the currency is available i the database, the currency's rate is updated. Otherwise, a new currency should be created, if the line contains a name for the currency. The function should show lines that contains errors and can not update the table.

THE PROTOTYPE

The prototype is a program called *World*, which opens the following window:



The program should show the design of the main view. The window shows two *JTable* components in a *JSplitPane* and the tables shows an overview for respectively currencies and countries. Search is implemented with filters below each table, and the *Clear* buttons are used to remove the filters. The two buttons above the currency table are used for

- import of currency exchange rates from a CSV file
- add a new currency to the table (and the database)

The button above the countries table is used to add a new country, while the combo box contains a list of all continnents in the world and offers the opportunity to filter the countries table by continent.

To edit and possible delete either a currency or a country is decided that you have to double click the current object in one of the two tables, and then you get a dialog box that displays the object's data.

To write the prototype I have created a new project *World* in NetBeans, and besides the main-class, are added the following classes (files)

- *MainView*, that is the above window
- *Currencies*, that is a data model for the currency table
- *Countries*, that is a data model for the country table

The last two are trivial, but they are necessary for the prototype can create the two *JTable* components. As an example is shown the one below:

```
package world;

import javax.swing.table.*;

public class Countries extends AbstractTableModel
{
 public int getColumnCount()
 {
  return 2;
 }

 public int getRowCount()
 {
  return 0;
 }

 public String getColumnName(int col)
 {
  return col == 0 ? "Code" : "Name";
 }

 public Object getValueAt(int row, int col)
 {
  return null;
 }

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

After the prototype is finished, I've created a copy of the project, which I have called *World0*. I will now continue to work on the project, but with the copy I can always return to the prototype.

THE DATA MODEL

As a next step I have written some model classes, and thus the classes to represent the program's data.

There's added three very simple model classes, representing respectively a continent, a country and currency:

- *Continent*
- *Country*
- *Currency*

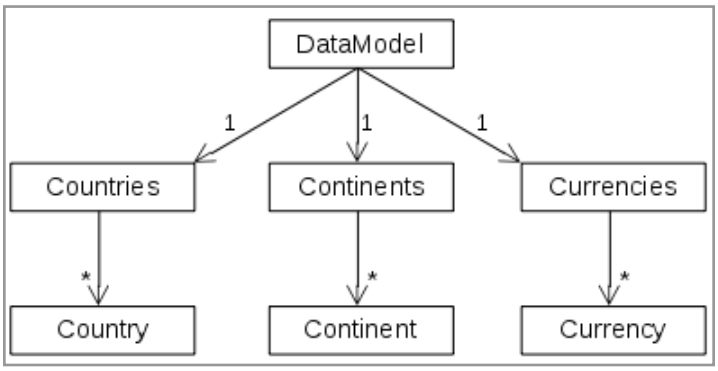
The classes are directly modeling the rows in the corresponding database tables, however, the class *Currency* extends with a list containing *Country* objects for the countries using this currency.

There are also defined a collection for each of the above types:

- *Continents*
- *Countries*
- *Currencies*

each of which includes an object for each row in the corresponding database table. Here the first of the classes is trivial, while the other two are reprogrammings of the corresponding classes from the prototype. The two classes are thus at the same time model for the program's *JTable* components.

The total model is represented by the class *DataModel*, which is a simple class that is made up of an object by each of the above three collections. The program's model can be illustrated as follows:



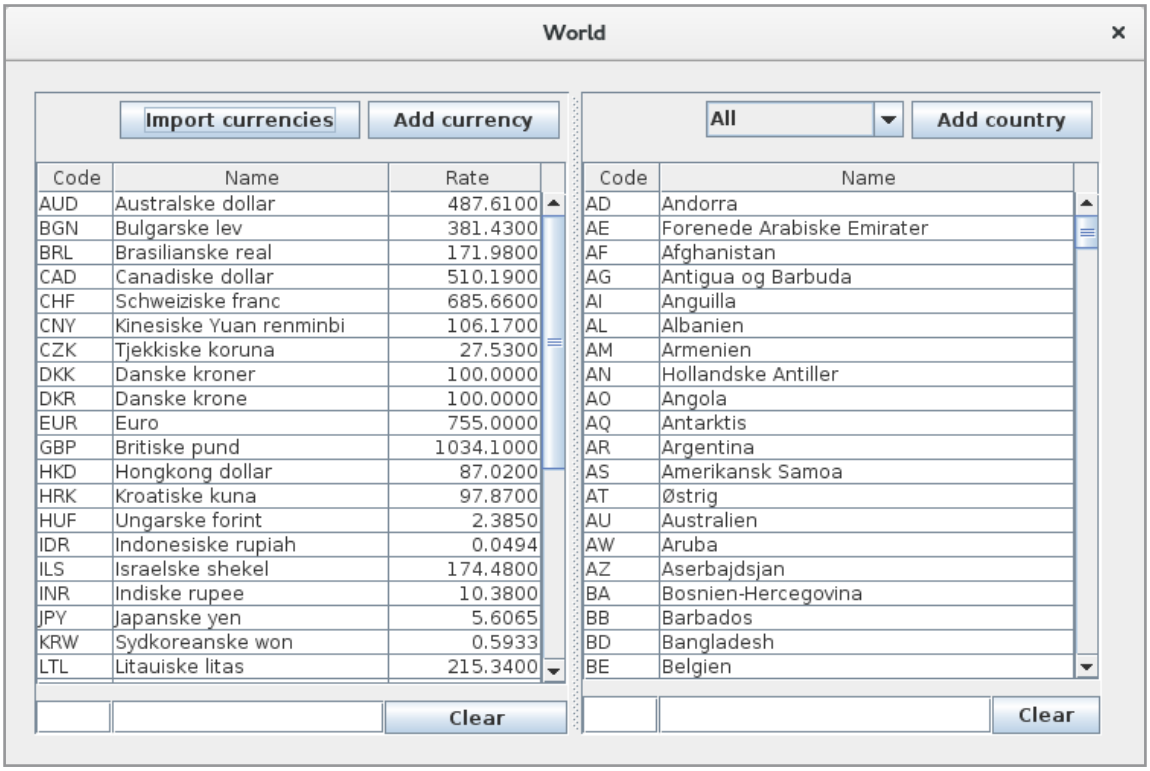
The three collections must be initialized, which is done by reading the corresponding database tables. To this end, it is written a class *Repository*, which only has static methods, including among others, methods that creates the three collection classes. The class *Repository* also has update methods for the tables *currency* and *country*, and that means that everything that has to do with the database is collected in this class. If the application must use a different database, it means that only this class has to be changed. The class *Repository* is a relatively complex class, as everything concerning the database and SQL are gathered here, and thus everything that is introduced in this book.

THE USER INTERFACE

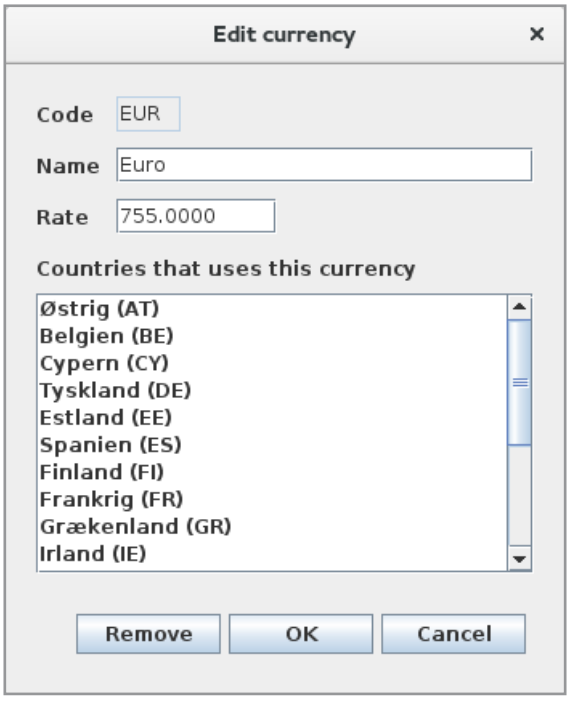
The program's user interface consists besides *MainView* of

- *CurrencyView* which is a dialog box for maintenance of currencies
- *CountryView* which is a dialog box for maintaining the countries

Furthermore, there is a secondary dialog box called *ErrorView* and is used to display a list of errors with the import of exchange rates. Below is the *MainView*, which is the same view as in the prototype, but this time initialized with data:

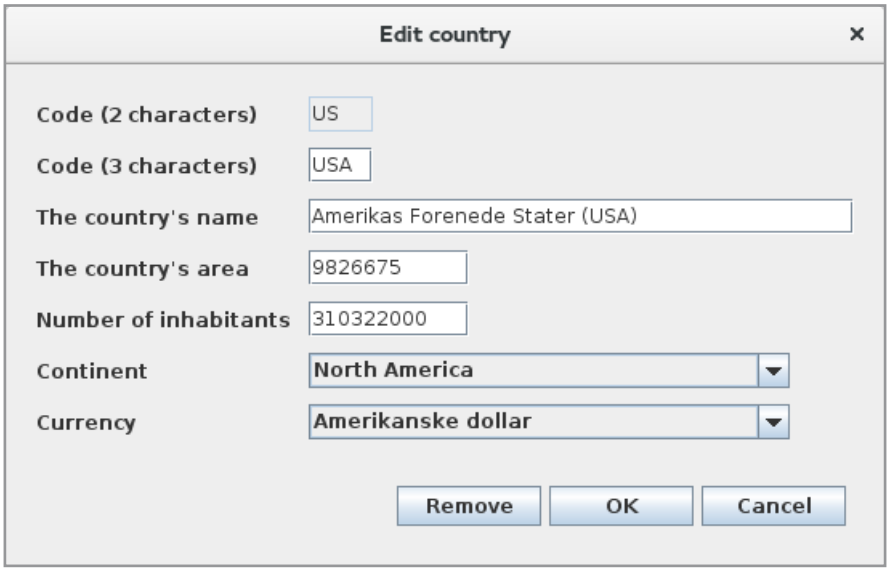


For each view (except *ErrorView*) there is attached a controller. The goal is that the controller class should validate the user input and choices, and it is also, where appropriate the controller classes that calls the methods in the class *Repository* to update the model. The controller classes for the two dialog boxes for maintaining respectively currencies and countries works in principle the same. Below is the dialog for maintaining currency:



It is the same dialog that is used regardless of whether to create a new currency, or to edit an existing currency. In this case, there is double clicked EUR, in the main window.

Similarly is below shown the dialog box for the maintenance of countries and where there are double clicked on US:



By thus associating a controller for a dialog you achieves two things:

1. everything regarding algorithms and business logic is moved into a separate class
2. the code of dialog box becomes simpler and easier to understand

Now one should not at any price attach a controller class for a dialog/window. A controller class must contain something interesting for it to makes sense.

IMPORT AF CSV FILE

The last feature that is missing is updating the exchange rates from a CSV file. This includes being able to open the file, parse it and use the result of the pasing to update the database table *currency*. With regards to browse the file, it is an activity that belongs in an event handler in the *MainView*. When the file is selected, it is sent to the controller class for *MainView*, which then parses the file and validate the individual lines. On the basis of the legal lines it creates *Currency* objects, which are sent to a method in the class *Repository*, which then updates the database.

10.2 MYWINES

The following project is to write a classic database application. This means that you must create a database and write a program that can maintain this database. The program is an ordinary PC application, that should maintain information about a private wine cellar, which is not entirely realistic. Should such a program be written in practice, you would probably write a web application. It should be ignored, and the purpose of the project is to test most of the substance that is treated in this book.

Compared with the programs that I have shown in this and the previous books, it is a relatively large program, and it is also a program that can be used in practice if one has a wine cellar and feel a need to register his wine consumption. The program consists of many classes (about 80), and there are more than 25 windows (dialogs). Now it's all not as violent as it sounds. Most of the dialog boxes are simple, and the same goes for the corresponding classes. All dialog boxes are implemented in the same manner with a view class that implements everything concerning the user interface, and then one or two model classes. There will always be a model class, which models the object which the dialog box maintains, and in the case where the dialog box has to display a collection of objects (rows in a database table), a *JTabel* is used in the user interface, and the dialog box has also assigned a data model for a *JTable*. In addition to model classes most dialog boxes also have a controller class that validates the user input and possible calls methods in the application's repository to update the database. You should therefore note that many dialog boxes are implemented in the same way that makes it easier to understand the program and thus maintain the code.

When you have to write a program which, that as in this example, will consist of many dialog boxes (the program has many features) which are similar and almost works the same way, you can then try to reduce the code and the number of dialog boxes by parameter control the dialog boxes and apply them to multiple functions depending on the parameters that are transferred to the constructors. If you do that, you get less code and fewer classes and thus, in principle, a program that is easier to maintain, but however more complex classes, which can be difficult to understand, because you have dialog boxes that perform several functions. Conversely, a dialog box for every function mean that the number of classes becomes very large, which in turn may mean that you have to change in many places for maintains the program. It is a choice, and in the current solution I have tried to some extent to parameter control dialogs without letting it go beyond clarity.

You are encouraged to spend the time reading the following description of how the program is designed and also to study the final result and including primary the program's code and to test the program.

THE TASK

Many private wine lovers has a substantially wine storage – perhaps several hundred bottles or more. It requires control, including to ensure that the wine is not too old, but also for the sake of historical data with information about prices, ratings, etc. These historical data are vital when buying the same wine again, but also to learn from experience and in general as documentation of purchase and consumption. There is thus not only a need to keep track of the current stock, but at least as much a need to store information about wines that are drunk and removed from the wine cellar.

The wines are acquired from different Danish suppliers – including grocery stores, but there may also be stock entries either in the form of direct import (that is from holidays or trips to wine countries) or gifts. The wines are purchased at greatly varying prices, and especially in connection with gifts, the price can be 0 (or lack of). It is common in the business, the granting of large discounts (and, arguably, even unrealistically large similar to that the list prices is not real). The price, and thus the discount granted is often determined by whether you buy single bottles or boxes (of 6 or 12 bottles).

The task is to write a program called *MyWines* that should be used to manage a private wine cellar. The program should be written in accordance with the following requirements/wishes.

In general, it should be possible to record the following information about a wine:

Information on the supplier (where the wine is purchased or obtained):

The supplier's phone number

- The supplier's name (company name or other)
- The supplier's address
- The supplier's zip code
- The supplier's city
- The supplier's email address

Which supplier information that actual should be recorded may vary, because a wine can be a gift or be imported from abroad.

Information about the wine (that is facts about the product):

- The wine's name
- The producer's name
- The wine's type (white, red, rosé, sparkling, dessert, port)
- The country where the wine is produced
- The country's code
- Wine district (for example Alsace, Barolo etc.)
- The grapes used for the production of the wine and in what quantities
- Classification if there is a classification (for example Chianti Classico Riserva DOCG)
- The wine's vintage
- Alcohol percent
- Size of the bootle/packing (in cl)

Information about the purchase:

- The date when the wine is purchased
- Expiry date (date when the wine latest should be drunk)
- The wine's price without discount
- Discount price
- The number of bottles purchased
- Quality (D = daily wine, H = house wine, Q = quality wine, C = cellar wine, X = cult wine)

The stock levels:

- Stock level (the current number of bottles)
- Revised quality (D, H, Q, C, X)
- Rating – an assessment consists of a date and a character from 0 to 100
- A description of the wine as a text

Every time there is used a bottle from the store, you have to register, which wine it is, how many bottles are taking and when.

For both the supplier and the wine itself one can not expect that all the information is present, and the same goes for how long the wine can be saved, discount and classification when a new wine must be added to the store, and in general you should be aware that sometimes you only have the knowledge that you can get from the label on the bottle.

The program should basically have the following features:

1. Register/create new wines (purchase)
2. Updates (consumption from the store and classifications)
3. Search functions

Because the work of records information about wines can be comprehensive (there may be many details), you should aim for a solution where you have to enter as little as possible, and where it is easy to get the data available, which is already in the system. On the whole, you should priority that the program is easy to use and it is important that the program is robust and react sensibly by improper operation.

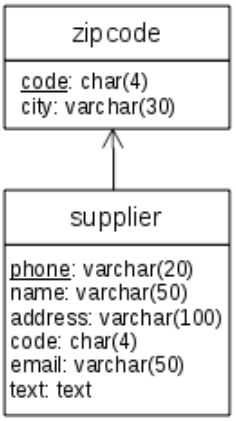
The program must have a search function where one can easily search wines from several criteria. You should aim for enhanced flexibility when searching, but conversely, the function should not be too complex. In connection with the search for wines, one should get:

- an overview of the total wine purchases within a given period and if necessary with the possibility of demarcation on country or by other criteria
- an overview of the total consumption within a given period and if necessary with the possibility of demarcation on country or by other criteria
- an overview of a wine's rating assessments over time (points)

It is also a desire that the program has a feature where one quickly can get a list of the wines that have reached the expiration date, and possibly has exceeded the expiration date, but also wines that should be drunk within the next time.

DATABASE DESIGN

I will start the development with design and implentation of the database. If I look at the above description of the task, the database should contain infomation about suppliers, and the analysis of the description results in two tables (se below). Basically, it is an address book, and here you will usually place the zip code and the city name in a table for zip codes, and then let *code* be a foreign key in the table *supplier*. Sometimes one can illustrate database tables with a figure as below, here indicates that there are two tables, and what columns these tables must have. If the name of a column is underlined it means that the column is a primary key. Which columns there should be, comes from the description of the task, but in practice the software developer through an analysis must contact the person who has proposed the task to clarify uncertainties, but also exactly to define the data types to be used. In this case, information about the supplier has been expanded with a new value, where it is possible to register a description of the supplier. On a figure as below the arrow shows, that the table *supplier* must have a foreign key to the table *zipcode*.



The primary database table is a table *wine*, which should contain information about a particular wine, and thus a wine to be included in the cellar. This table would have among others the following columns:

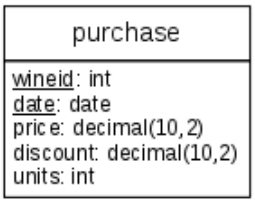
```
wineid      int               # surrogate key
name        varchar(50),      # the wine's name
year        int,              # the wine's production year
time        int,              # expiration time in years
percent     decimal(5,1),     # alcohol percent
store       varchar(100),     # storing on barrel / tank / bottle
amount      int,              # current amount
text        text,             # description of the wine
```

and here it is also agreed that it should be possible to register a description of a wine. A wine has to be identified by a surrogate key, and it is agreed that two wines are considered different, if they do not have the same production year. Furthermore, for each wine it must be possible to define

- supplier
- packing (bottle, bib, other)
- producer
- production country
- district
- classification
- wine type
- wine category

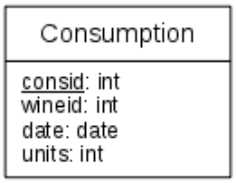
Here are suppliers already described and the table wine must have a foreign key to the *supplier*. The 7 other informations is in principle text, but it has been decided to register the individual information in their own table. The argument is partly that several wines must be registered with the same value, and second, that it can be difficult to ensure that the same text is spelled the same way each time, and finally you have for packing and district to register an addional information which is respectively the packing's volume and the district's country. For the type of wine and wine categories the description of the task defines a natural primary key (one letter), but for the other 5 I has to choose a surrogate key. The design should therefore be extended with 7 other but simple tables (these tables are sometimes called dimension tables) and the table wine must have additional 8 foreign keys.

There is also a need for a table to purchases:



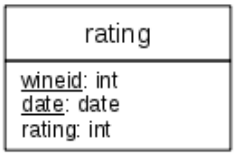
where a purchase is identified by the primary key of the table *wine* as well as a date. Based on the analysis it has been decided that the same wine can not be purchased twice the same day. Should the same wine could be purchased several times on the same day, it would be necessary to expand the table with a surrogate key.

Similarly, there must be a table with information on consumption of wines:



Here however, it is decided to use a surrogate key corresponding to the possibility of consuming the same wine several times on the same day.

Finally, there must be a table to record the user's ratings by the 100 points scale:

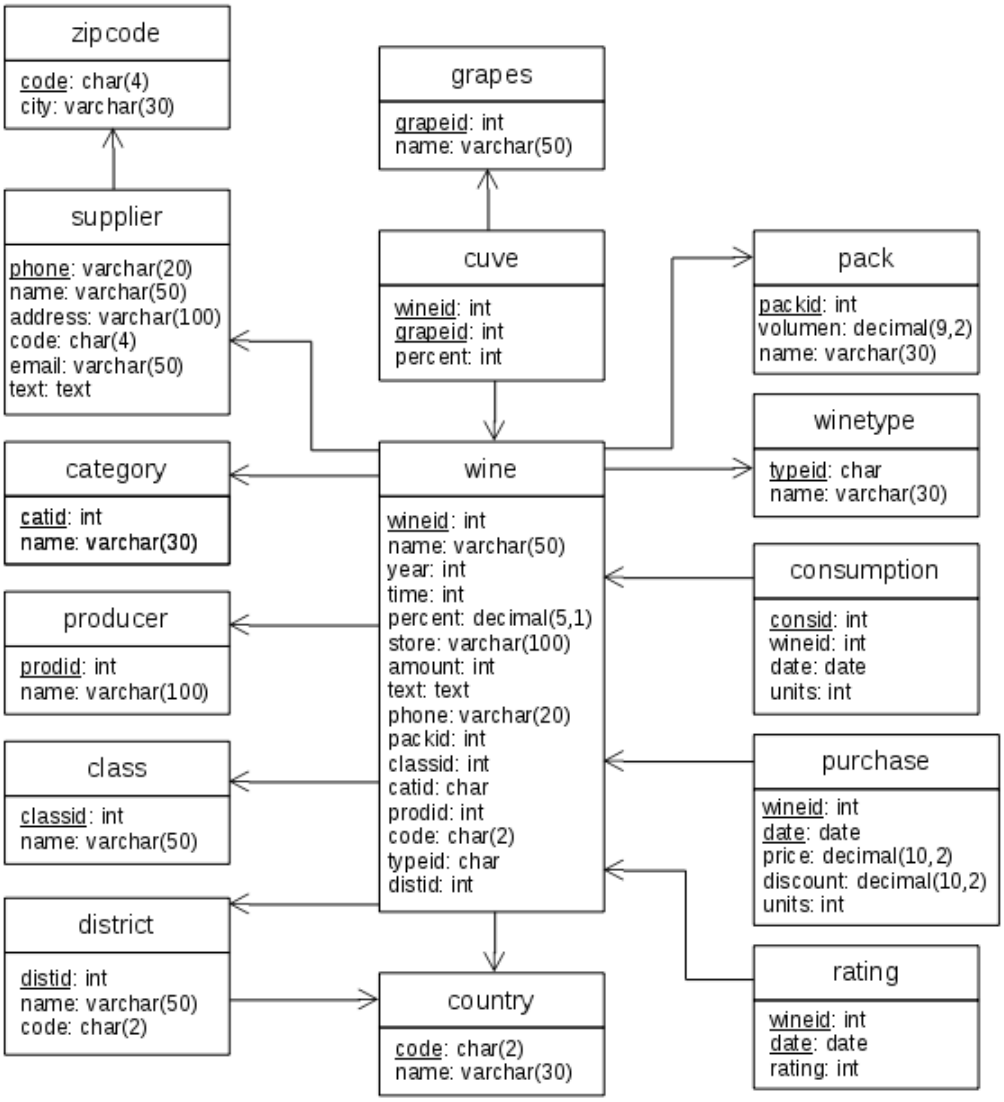


and here it is again assumed that the individual wine has only one registration per day (only one rating for a wine the same day).

After these considerations, the design of the database is illustrated as shown below.

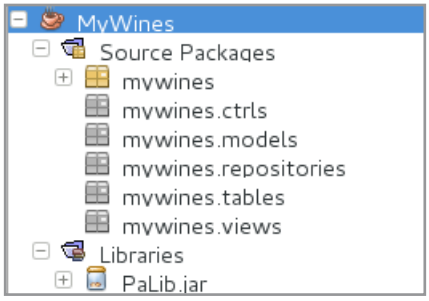
Along with the project is a script called *Wine.sql* that creates the database. I will not show the script here, but in addition to the database – called *cellar* – the script initializes the four tables:

- *zipcode*
- *winetype*
- *category*
- *pack*



PROGRAM ARCHITECTURE

The program should in principle be able to maintain the above database, and to that purpose the program must use some windows and dialog boxes, and the program will consist of many files (types). As a start, I have therefore created a project *MyWines* and created the following packages:



- *mywines* is the default package created by NetBeans, and should just contain the class with the *main()* method
- *mywines.ctrls* should contain all controller classes to dialog boxes
- *mywines.models* should be used for model classes, which is essentially a model class for every database table
- *mywines.repositories* that should contain classes with the SQL code, and then the code that extracts data from tables and update the tabels
- *mywines.tables* that belongs to the user interface layer and consists at classes thatt act as data models for *JTable* components
- *mywines.views* that shouls contain dialog boxes

In addition is added a reference to my class library *PaLib*, when I will use tools from this library.

THE MODEL LAYER

I am now ready to start on the development of the program and I will begin with the model layer and writes the first model classes, when a model class should modeling a row in a database table. So far, I will concentrate on the table *wine* and the tables which it refers, and I will write the following model classes:

- *Zipcode*
- *Supplier*
- *Producer*
- *Category*
- *Classification*
- *Winetyper*
- *Grape*
- *Country*
- *District*
- *Packing*
- *Wine*

Except the last they are all simple and vary only in terms of the fields they contains. As an example is shown the class *District* that is modeling the table *distrct*:

```
package mywines.models;

public class District
{
 public static final int NAME = 50;
 private final int id;
 private String name;
 private String code;

 public District(int id, String name, String code)
 {
  this.id = id;
  this.name = name;
  this.code = code;
 }

 public int getId()
 {
  return id;
 }
```

```
  public String getName()
 {
  return name;
 }

 public String getCode()
 {
  return code;
 }

 public void setName(String name)
 {
  this.name = name;
 }

 public void setCode(String code)
 {
  this.code = code;
 }

 @Override
 public boolean equals(Object obj)
 {
  if (obj == null) return false;
  if (getClass() == obj.getClass()) return id == ((District)obj).id;
  return false;
 }

 @Override
 public int hashCode()
 {
  return id;
 }

 @Override
 public String toString()
 {
  return name;
 }
}
```

The class is simple since it only contains get and set methods for the class's variables, but note that the class overrides *equals()* so that two objects are equal, if the corresponding rows in the database have the same primary key. Also note that there are initially defined a *public* constant *NAME*, that indicates how many characters are reserved in the database for the column *name* (in the database the type is *VARCHAR(50)*).

The development of a database application will typically start to write that kind of model classes. There may of course be variations, but basically they will model the database table's columns, and often the classes will also be expanded later in the development process.

The class *wine* is basically designed in the same way, but it takes up a lot more, as the table *wine* has many columns. The beginning of the class is:

```
public class Wine
{
 private int id;
 private String name;
 private Integer year;
 private Integer time;
 private BigDecimal pct;
 private String barrel;
 private int units;
 private String text;
 private Packing packing;
 private Supplier supplier;
 private Producer producer;
 private Country country;
 private District district;
 private Classification classification;
 private Category category;
 private Winetype winetype;
 private List<Cuve> cuve;
 private boolean mark = false;
```

Here you should note how the columns for the foreign keys are defined as a reference to an object of that model type. That is, if you, for example, consider the supplier, it is not a *String* (for the primary key *phone*), but a reference to a *Supplier* object.

You should also note the variable *cuve*, that is a collection with objects of the type *Cuve*. The database has a table *cuve* that is a relation table between *wine* and *grapes* and defines which grapes are part of a wine. The class *Cuve* modelings this table, but is expanded with a variable for the grape's name:

```
public class Cuve
{
 private int wineid;
 private int grapeid;
 private String name;
 private Integer pct;
 private boolean mark = false;
```

There is also a variable *mark*, which allows to highlight (select) objects in a collection. The class *wine* has a corresponding variable. You will se, how this variables are used in the user interface.

THE MAINVIEW

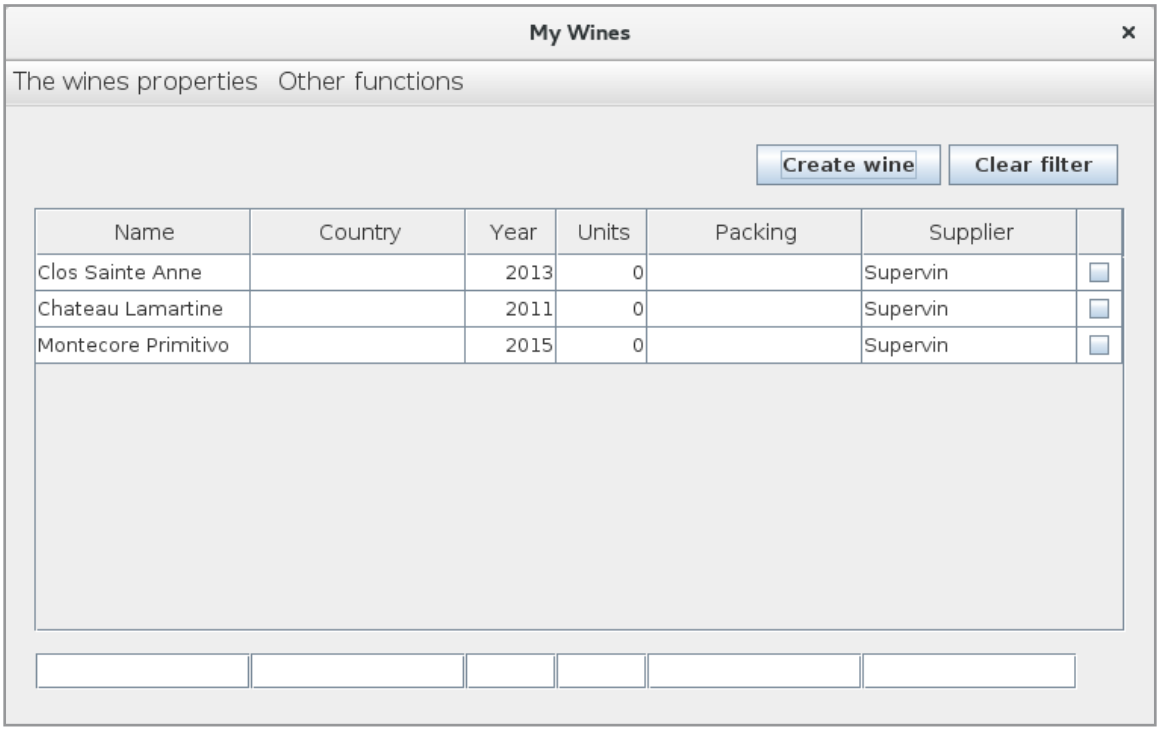
I will then write the program's *MainView*, but first I have in *MySQL Workbench* performed the following script that creates a supplier and three wines:

```
use cellar;

INSERT INTO supplier (phone, name, address, code, email) VALUES
('98921853', 'Supervin', 'Skagensvej 201', '9800', 'info@supervin.dk');
INSERT INTO wine (name, year, phone) VALUES
('Clos Sainte Anne', 2013, '98921853');
INSERT INTO wine (name, year, phone) VALUES
('Chateau Lamartine', 2011, '98921853');
INSERT INTO wine (name, year, phone) VALUES
('Montecore Primitivo', 2015, '98921853');
```

The script is called *Threewines.sql*, and the aim is, that the database should contain data, which can be displayed on the front window.

When I start with the main window, it is because I as quickly as possible wants to get a running program, so the program's user interface can be presented and finally defined with the future users.

When the program is executed, it should show the follwing window:



Center is a *JTable*, that shows an overview over all wines, but later it should show only be the wines, where the number of bottles in stock are positive. The last column with the check boxes should be used to select rows, that shows wines to be drunk. The *JTextField*'s at the bottom are for filters. If the user double-click on a wine in the table, the program should open a dialog box with all the details about that wine. The menu has the following functions:

- The wines properties
  1. Maintenance of suppliers
  2. Maintenance of producers
  3. Maintenance of countries
  4. Maintenance of districts
  5. Maintenance of classifications
  6. Maintenance of grapes
  7. Maintenance of wine types
  8. Maintenance of wine categories
  9. Maintenance of bottles/packings

- Other functions
  1. Advanced Search
  2. Purchase
  3. Consumption

To write the class *Mainiew* I have also written the following classes

- *mywines.tables.Wines*
- *mywines.repositories.DB*
- *mywines.repositories.Repository*

The class *Wines* is a simple model class to a *JTable*, and contains nothing new. The class *DB* should represents a connection to a database and is written as a singleton:

```
package mywines.repositories;

import java.sql.*;

public class DB
{
 private static DB instance = null;
 private String host = "localhost";
 private String port = "3306";
 private String data = "cellar";
 private String user = "pa";
 private String code = "Volmer_1234";

 private DB()
 {
 }

 public static DB getInstance()
 {
  if (instance == null)
  {
   synchronized (Repository.class)
   {
    if (instance == null) instance = new DB();
   }
  }
  return instance;
 }
```

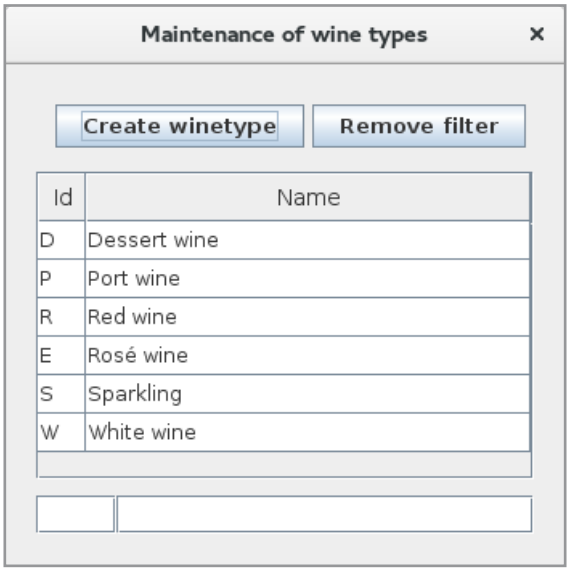
```
  public Connection getConnection() throws SQLException
  {
   return DriverManager.getConnection(String.format(
    "jdbc:mysql://%s:%s/%s?useSSL=false", host, port, data), user, code);
  }
}
```

Immediately the class makes little sense, since it merely consists of variables that contains parameters for the database, but the class must be used to solve the problem that the database parameters are hard coded so they can be lifted out of the program to a configuration file.

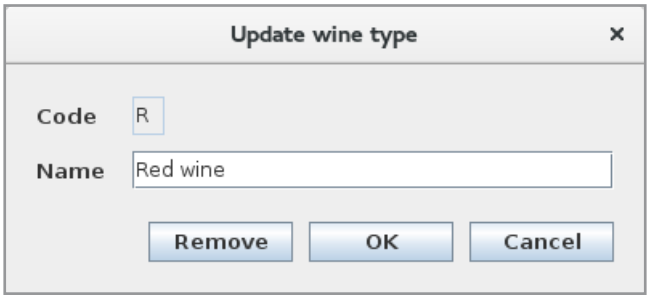
The class *Repository* is also written as a singleton, and so far it has only a single method that can return a *Wine* object for all rows in the table *wine*. However, it is the program's central class, and will includes all methods for database operations. It thus becomes a very extensive class. Al SQL is hidden away in this class.

THE DIMENSION TABLES

As a next step I want to write the code for the maintenance the dimension tables, and thus all functions under the menu *The wines properties*. All functions works in principle the same way, and I will as an eksemple look at the *Maintenance of wine types*. If you click the menu item, the program opens the following dialog box:



The dialog box shows a *JTable* with all wine types and such the content of the table *winetype*, and at the bottom is defined a filter to the table. The two buttons are used to create a new wine type and to remove a filter. Double-Clicking on a line in the table, you get a dialog box where you can edit (and delete) a wine type, for example:



It is the same dialog used if you create a new wine type.

The menu item *The wines properties* has 9 functions, and for each of the 9 functions the following should happens:

1. The class *Repository* must be extended with 4 new methods, a method that returns a list of objects corresponding to the content of the database table, a method that creates a new row in the database, a method that updates an existing row and a method that deletes an existing row.
2. A class that is a data model for the *JTable* component.
3. A class that is the dialog box with the *JTable* component.
4. A class that is a controller for the dialog box to edit an object.
5. A class that is the last dialog box.

When the menu has 9 functions it means that the program may be expanded with 36 new classes like the class *Repositoty* must be expanded with 36 new methods. It sounds like a lot and it is, but conversely note, that there is also talk about the program code to maintain 9 database tables, and finally you should note that the classes are generally simple, and the same applies to the methods in the class *Repository*.

You should also note that the implementation is a form of a pattern for a program to maintain a database table, where you have a dialog box with a *JTable* and filter that displays the content of the database table, and where you can open a dialog to add or edit a row in the table. I like to do the job in this way, because it is clear when the table is changed, but there are other ways, and here it is particularly important to note that one can edit the contents of a *JTable* directly.

THE WINE TABLE

Maintenance of the table *wine* follow the same pattern. That is, the *MainView* has a button to create a new wine, and if you double-click on a line in the table, you can edit the wine. In both cases the same dialog box is used, and in principle it means as above

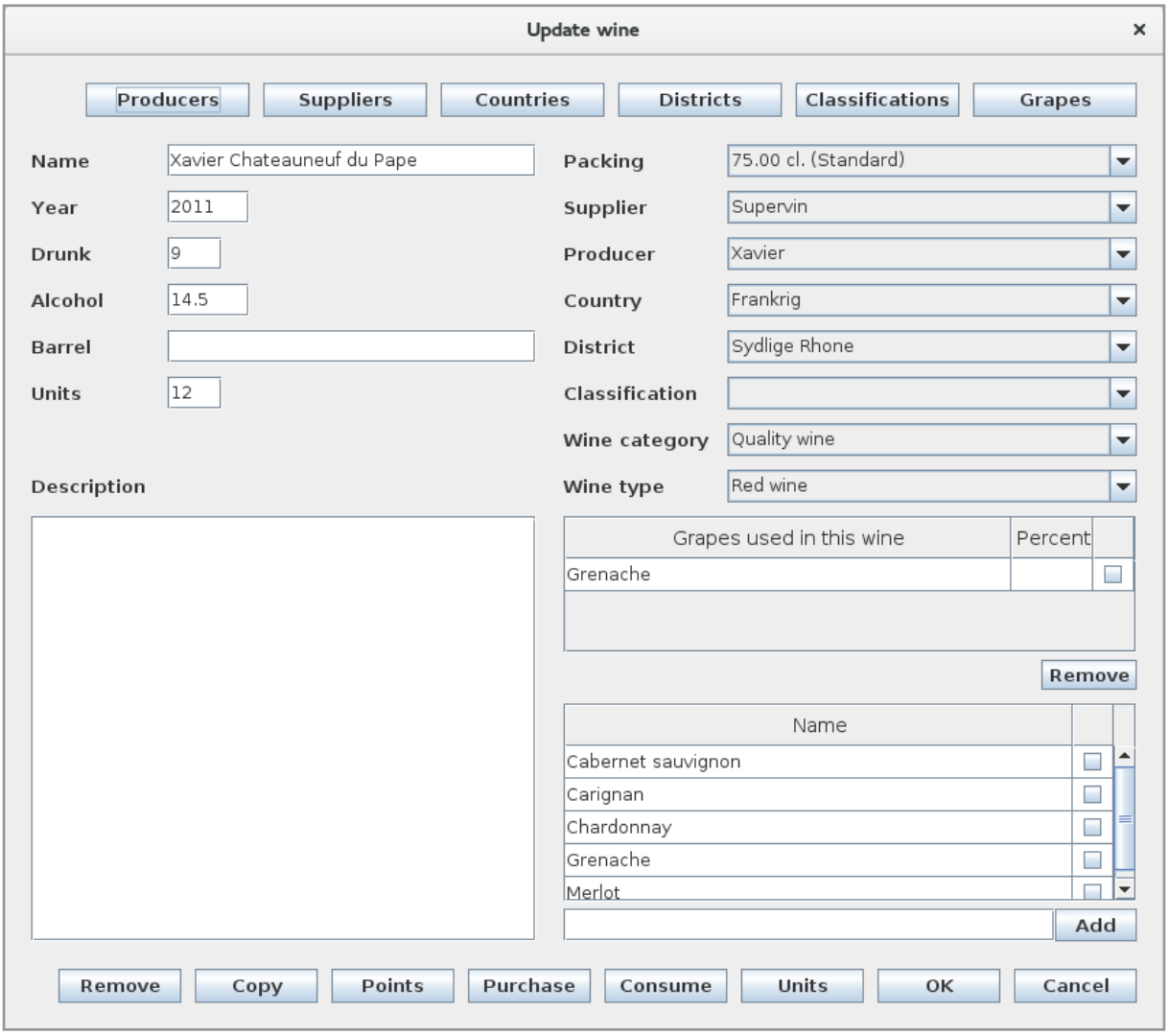
- the class *Repository* must be expanded with 3 new features (the function that returns all wines have already been implemented)
- to write a controller for the dialog
- the dialog box itself must be written

The difference is that this time it is a very complex dialog that has a number of other functions. If you double-click a wine in the main window, the result could be the window as shown below.

The dialog box has to the left input fields for the information, that can be entered, while the right side is combo boxes to select a value from a dimension table. Everything should be self-explanatory except for the two *JTable* components for the grapes. The bottom shows a table with all the grapes in the database with an associated filter. The top is to the grapes used in this wine. If you in the lower *JTable* choose one or more grapes by placing a checkmark in the check box and click the *Add* button, these grapes are added to the upper *JTable* and therefore grapes that are part of the wine. In the upper *JTable* you can similarly set a checkmark in one or more check boxes, and if you clicks on *Remove*, the grapes are removed from the wine. The top *JTable* also has a column that is to the grape's part in this wine if it is composed of several grapes. This column is edited by directly entering a value (in percent).

At the top of the window there are 6 buttons. They are used to create an object of that kind and thus has the same function as the functions that can be selected from the menu. The meaning of these shortcuts is, that if you are about to create a new wine (or edit an existing wine), and you as such discovers that the wine is from a district which was not been created, that you can instantly create the district. These 6 functions are using the same dialog boxes that are created earlier for maintenance of dimension tables.

At the bottom of the window are a further 8 buttons, and the meaning of the buttons *OK* and *Cancel* gives itself. The 6 other can only be activated if you are editing a wine, but not when you create a new wine. There function are explained below.

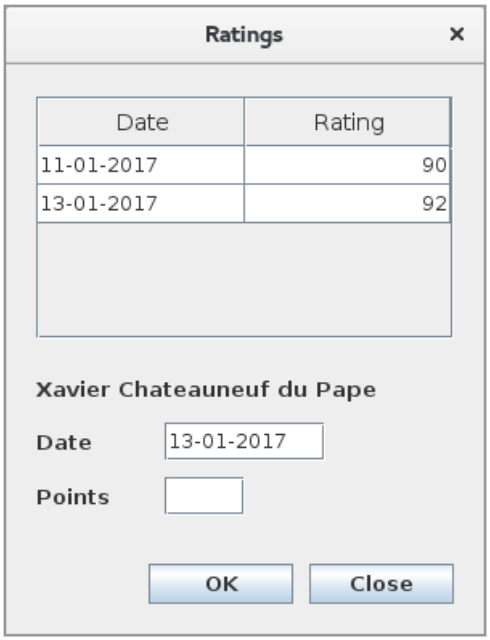


The button *Remove* is used til remove a wine. The function also removes all other information related to this wine.

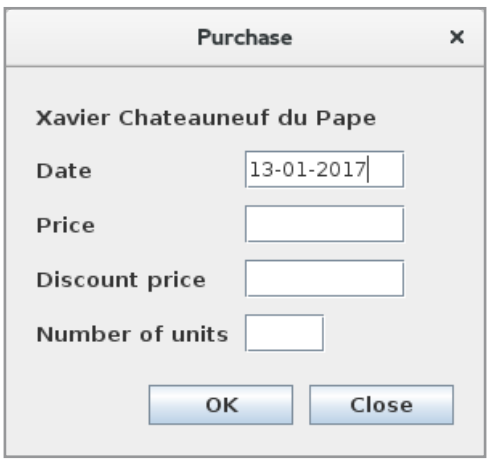
The button *Copy* is used to create a new wine with virtually the same data, but where the fields *year*, *drunk* and *units* is *null*. The reason is that the same wine in another year is considered as another wine, and then a new year can be created without to enter all the information again.

The button *Point* opens a dialog box called *PointView* as shown below. The function is used to assign a rating to this wine. The dialog box inserts today (that can be changed), and you should enter the point as a number between 0 and 100. A rating can not directly be changed, but if you double-click on a rating, you can remove it, and you can create all the ratings you like, but you can only create one rating for the same date. If you enter a rating for an existing date, the old value is overriden.

You should note, that dialog box has a *JTable*, and as so need a data model for the table (in the package *mywines.tables*). The same applies to the two *JTable* components above for the grapes. It means that every *JTable* add a class, and for the three mentioned here it is *Ratings*, *Cuves* and *Grapes*. *PointView* is a simple dialog box, but should update the database, and for that is assigned a controller with the name *PointCtrl*.

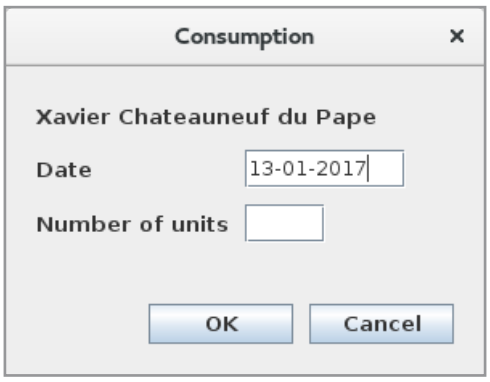


The buton *Purchase* opens the following dialog box:



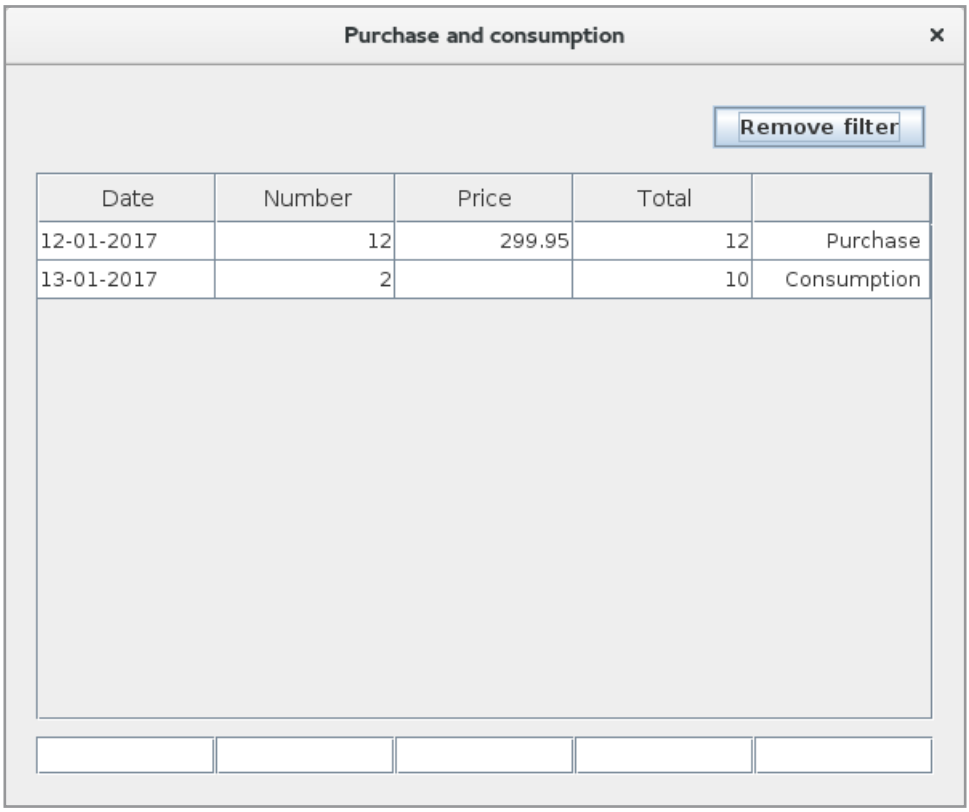
that is used to enter information about a purchase of a wine, and that is the price and number og units. The dialog box is called *PurchaseView* and the controller *PurchaseCtrl*.

The button *Consume* opens a dialog box to enten information about a consumption:



The dialog box is called *ConsumptionView* and the controller *CunsumptionCtrl*.

The last button *Units* opens a dialog box that shows what has happened with the store for this wine, that is alle purchases and consumptions:



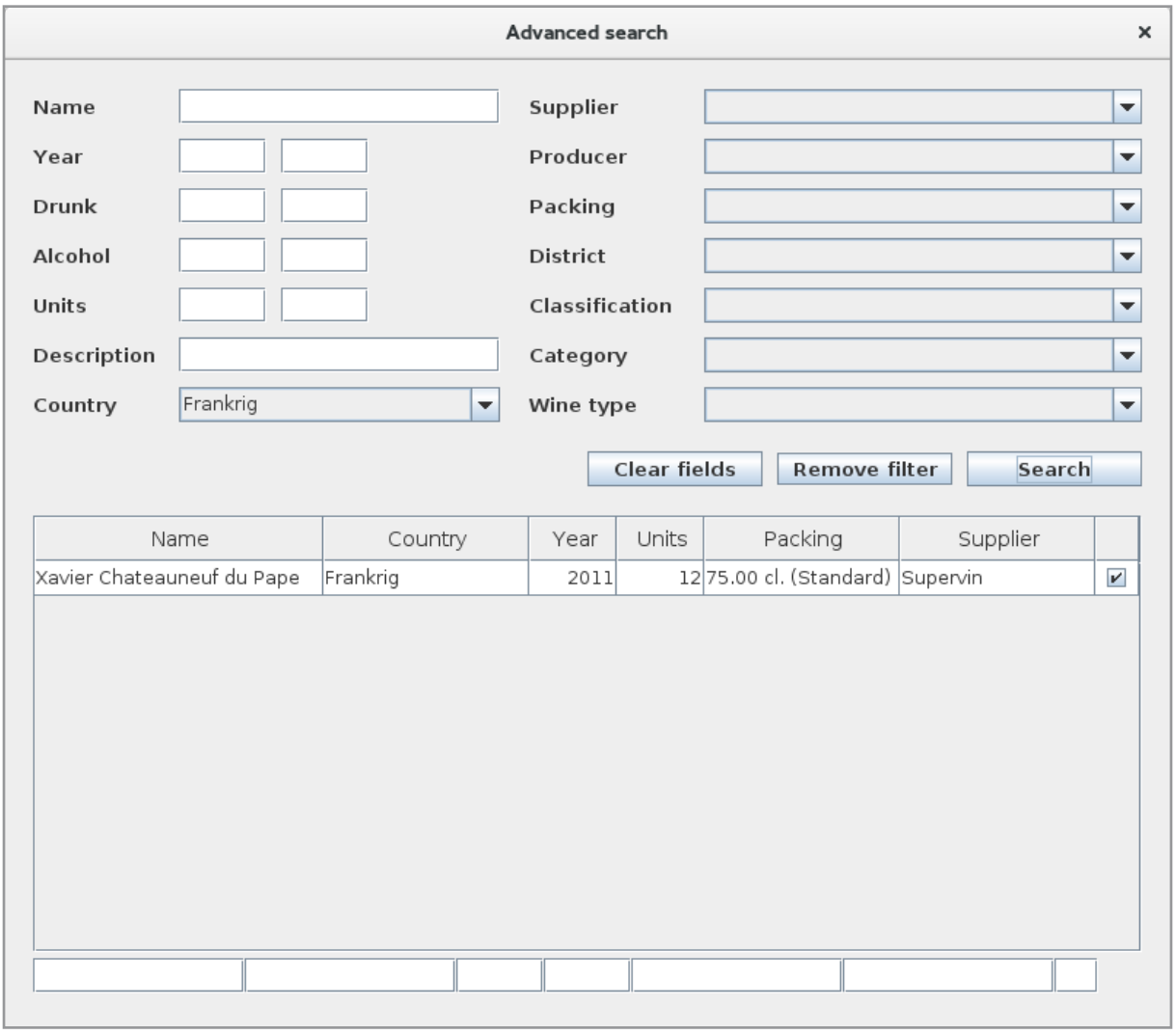
The dialog box shows primarily a *JTable* with a filter. If you double-click on a row in the table you are allowed to delete the actual purchase or consumption. The dialog box is called *UnitView*, and the data model to the *JTable* is called *Units*. The dialog also has a controller called *UnitCtrl*.

As described above, maintenance of wines is a function which extends the program with several new classes. Furthermore the function results in an extension of the class *Repository* with many new methods, and here you should special note, that several of these methods are implemented as database transactions.

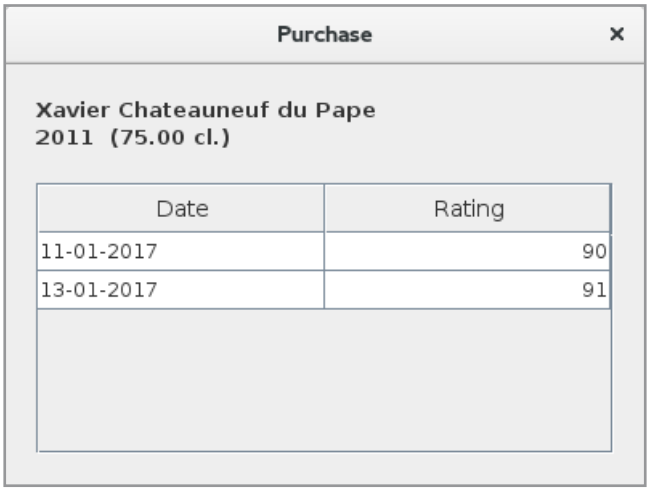
THE SEARCH FUNCTIONS

Then there is the three functions in the last menu. The three functions are very similar and opens dialogs with almost the same design. The first opens a dialog box as shown below and is used to search for wines based on several criteria. The window shows an example where there has been searched on all French wines. If a field is empty (has not been selected a search criteria), it is ignored in the search. If you enter a value in the fields *Name* and *Description* it means that a row in the database to match must contain the value in the actual column. For a criteria where you can enter two values, it means a from and to value, and if you only enter one value if means either from or to. If you select a value in a combo box a row in the *wine* table must have that value in the actual column (in the example belov a row must have the value *FR* for country.

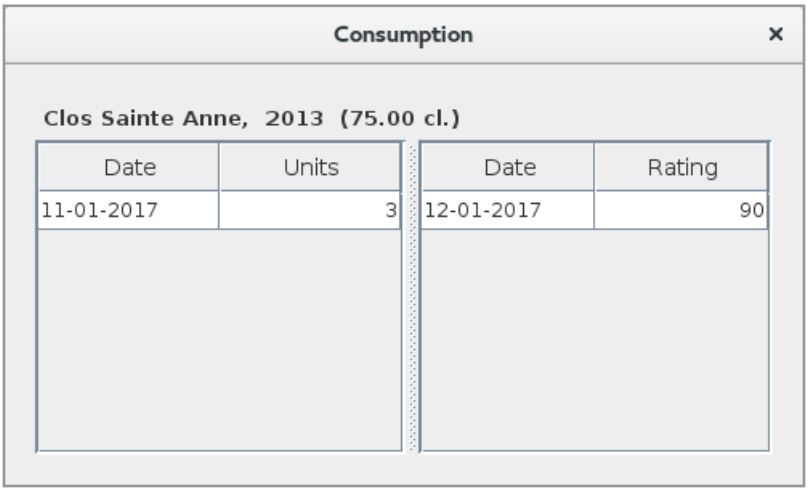
If you double-click a row in the table, the program opens til dialog box *WineView* for that wine, and you kan maintain the wine.



The two other search functions opens corresponding dialogs, and I will not show these dialogs here. The difference is that for both dialogs you also can choose a period as search criteria, and the *JTable* component has some other columns. Moreover, both dialogs shows some totals at the bottom. The first dialog is called *PurchasesView* and displays a list of all the purchases that matches the search criteria.

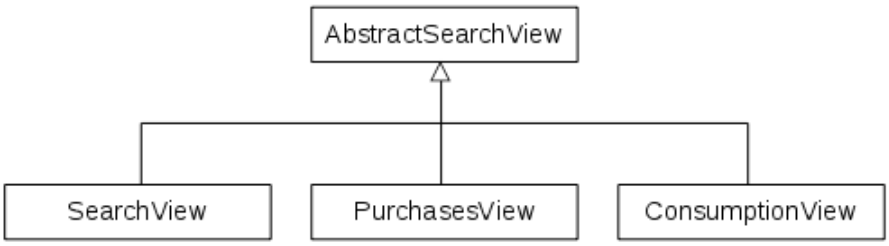


If you double-click a row in the table, you get the above dialog box showing all ratings for the wine. The last function works similar, but shows an overview of consumptions that matches the search criteria. If you here double-click a row, you get the following dialog:



The dialog box displays consumption by dates and again an overview of all ratings.

The three functions in turn results in a number of new classes, and I will not mention them here, but the three dialogs with search criteria are extensive but similar each other and therefore everything they have in common are moved to a base class:



There should be written some code to implement these three functions, but there is not much new compared to what already is mentioned. The most complex is to create the SQL expression to be used, partly because there are JOIN operations on many tables, and partly because the WHERE part depends on the search criteria selected. Everything happens in class *Repository*, and you are encouraged to study the code and the many details.

THE LAST THINGS

Back there are two things:

1. moving the database parameters from the program code to a configuration file
2. write an installation script

Regarding the first problem, I will use the following configuration file called *mywines.config*

```
host:localhost
port:3306
data:cellar
user:pa
code:Volmer_1234
```

Next, the class *DB* is changed:

```
package mywines.repositories;

import java.io.*;
import javax.swing.*;
import java.sql.*;

public class DB
{
 private static String path = System.getProperty("user.home") + "/";
 private static DB instance = null;
 private String host = "";
 private String port = "";
 private String data = "";
 private String user = "";
 private String code = "";

 private DB()
 {
  try (BufferedReader reader =
   new BufferedReader(new FileReader(path + "mywines.config")))
  {
   for (String line = reader.readLine(); line != null; line = reader.readLine())
   {
    String[] item = line.trim().split(":");
    if (item.length == 2)
    {
     String key = item[0].trim();
     if (key.equals("host")) host = item[1].trim();
```

```
     else if (key.equals("port")) port = item[1].trim();
     else if (key.equals("data")) data = item[1].trim();
     else if (key.equals("user")) user = item[1].trim();
     else if (key.equals("code")) code = item[1].trim();
    }
   }
  }
  catch (IOException ex)
  {
   JOptionPane.showMessageDialog(null,
    "The program can not connect to the database.",
    "ErrorMessage", JOptionPane.ERROR_MESSAGE);
   System.exit(0);
  }
}

public static DB getInstance()
{
 if (instance == null)
 {
  synchronized (Repository.class)
  {
   if (instance == null) instance = new DB();
  }
```

```
   }
   return instance;
  }

  public static void setPath(String pathname)
  {
   if (!pathname.endsWith("/")) pathname += "/";
   path = pathname;
  }

  public Connection getConnection() throws SQLException
  {
   return DriverManager.getConnection(String.format(
    "jdbc:mysql://%s:%s/%s?useSSL=false", host, port, data), user, code);
  }
}
```

The database parameters are now blank, and in return, the private constructor is changed to read the configuration file and initializes the parameters. Is it not possible, and here you must remember that the constructor is performed the first time the method *getInstance()* is called, it shows a message box, and the program terminates.

By default the constructor assumes that the configuration file exists in the user's home directory. It does of course not have to be the case, and therefore the class has a static method that can be used to define the directory that contains the configuration file.

After this class with *main()* method is changed:

```
package mywines;

public class MyWines
{
 public static void main(String[] args)
 {
  if (args != null && args.length > 0) mywines.repositories.DB.setPath(args[0]);
  javax.swing.SwingUtilities.invokeLater(() -> new mywines.views.MainView());
 }
}
```

If *main()* has an argument on the command line, use the first argument as a directory for the configuration file. You should also note that the *MainView* is opened in a different way. It is explained in the book Java 8 and can be ignored in this place, but it is the right way to start the program.

Then there's the final challenge with an installation script. The following files should be installed:

MyWines.jar
lib/PaLib.jar
mywines.png
mywines.config

The script is called *mywines.sh*. I will not show the script here, because in principle it works the same way, as I have shown before.

To run the program, the database must be created and therefore the database script must be executed before the program can be used. To distribute the program, I packs it all together in a tar file. First I copied all files to the directory *data/mywines* in my home directry:

```
[pa@localhost ~]$ ls -lR data/mywines
data/mywines:
totalt 488
drwxrwxr-x. 2 pa pa      4096   12 jan 21:38 lib
-rwxrwxrwx. 1 pa pa        63   12 jan 21:13 mywines.config
-rw-rw-r--. 1 pa pa    456246   12 jan 23:06 MyWines.jar
-rwxrwxrwx. 1 pa pa     20297   12 jan 21:37 mywines.png
-rwxrwxrwx. 1 pa pa       988   13 jan 17:30 mywines.sh
-rw-rw-r--. 1 pa pa      6886   7 jan 12:09 Wine.sql

data/mywines/lib:
totalt 44
-rw-rw-r--. 1 pa pa 43682 12 jan 21:38 PaLib.jar
```

Next, I created the tar file:

```
[pa@localhost ~]$ tar -tvf mywines.tar
drwxr-xr-x pa/pa         0  2017-01-13 17:49 data/mywines/
drwxrwxr-x pa/pa         0  2017-01-12 21:38 data/mywines/lib/
-rw-rw-r-- pa/pa     43682  2017-01-12 21:38 data/mywines/lib/PaLib.jar
-rwxrwxrwx pa/pa       988  2017-01-13 17:30 data/mywines/mywines.sh
-rwxrwxrwx pa/pa        63  2017-01-12 21:13 data/mywines/mywines.config
-rwxrwxrwx pa/pa     20297  2017-01-12 21:37 data/mywines/mywines.png
-rw-rw-r-- pa/pa    456246  2017-01-12 23:06 data/mywines/MyWines.jar
-rw-rw-r-- pa/pa      6886  2017-01-07 12:09 data/mywines/Wine.sql
```

and the result is that my home directory contains the file *mywines.tar*, and hence the file which can be distributed to users of the program. Here the user must extract the file:

```
$ tar -xvf mywines.tar
```

If the user is in the directory where the files are extracted and perform the following:

1. edit the configuration file
2. opens *MySQL Workbench* and performed the script *Wine.sql*
3. performs the installation script

and then the program should be running.

But there may be a problem with Java and the JDBC driver. If appropriate, you can find the solution in appendix A.

APPENDIX A: INSTALL MYSQL

The following is a brief description of how to install the database server *MySQL*, and as mentioned at the beginning of this book, you must install three products:

1. the database server
2. a client tool for managing and maintaining databases
3. a JDBC driver

Generally the three products are installed without major problems, and the following shows how I installed the products on a regular PC running Fedora 23.

THE DATABASE SERVER

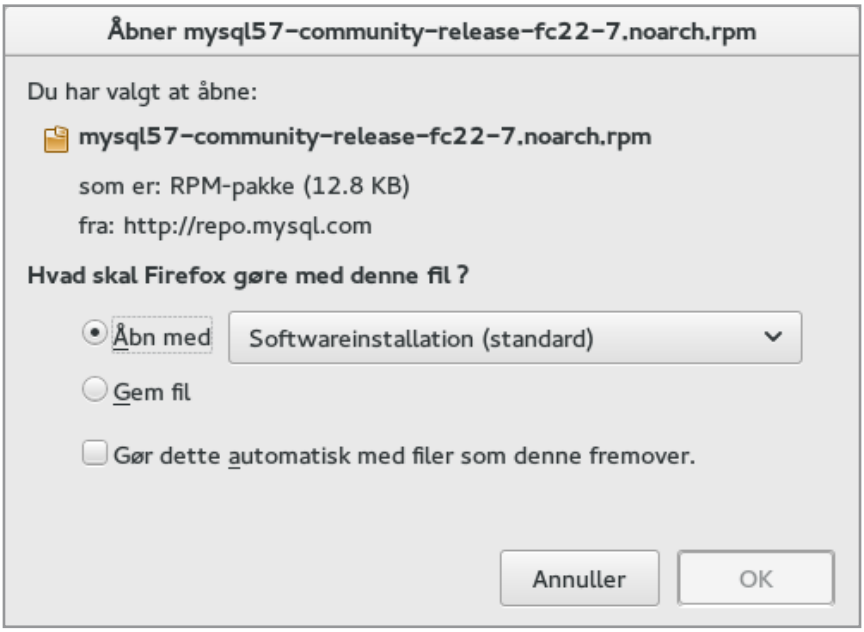
The server can be downloaded from the page

```
http://dev.mysql.com/downloads/mysql/
```

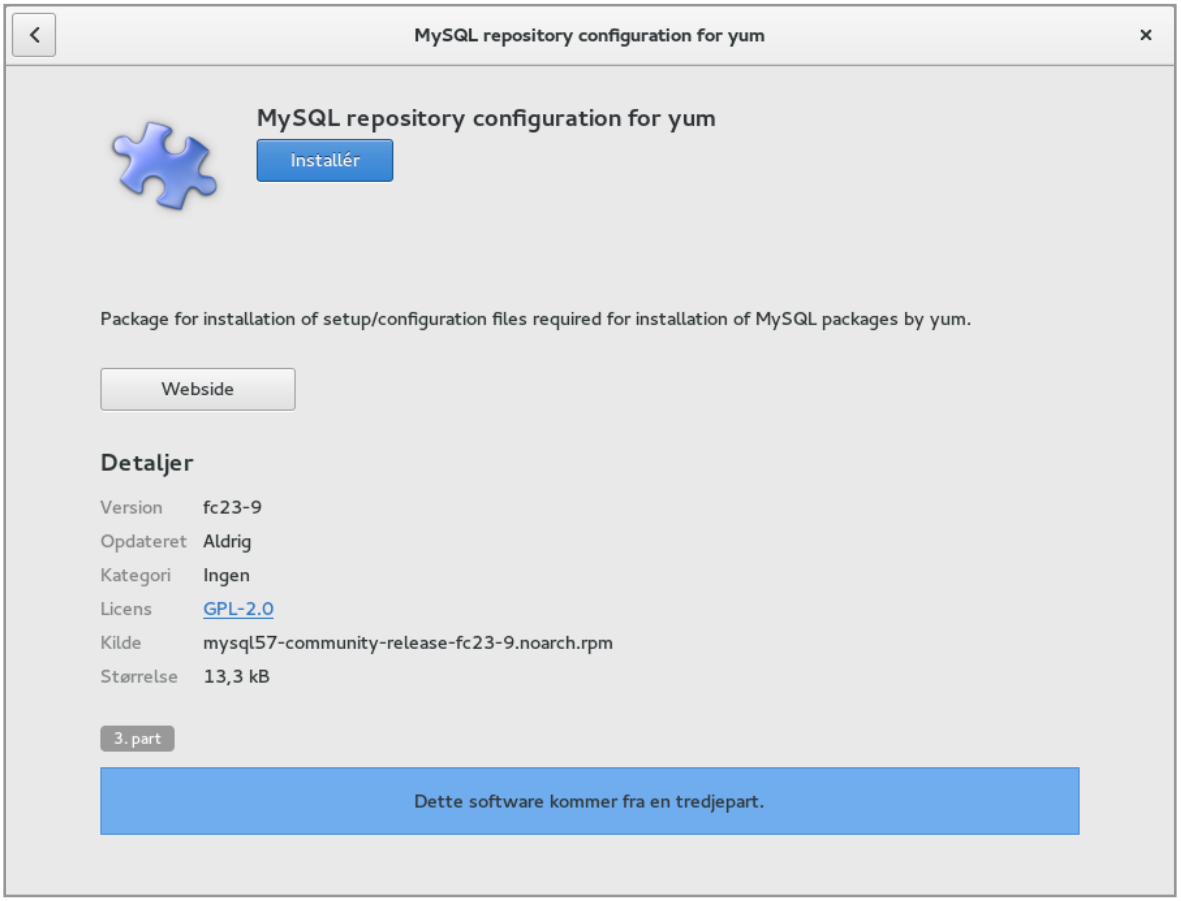
Here, you should select platform (for example Fedora), and when you then click *Download* you will be asked to log in, and is it the first time you download *MySQL*, you must create an user logon. After you have logged in, select the product, and I have chosen:

```
Fedora 23 (Architecture Independent), RPM Package
```

Then you get the usual window for *Download* / *Installation*, which in this case is a *rpm* package:



I have chosen Software installation, so the product will be installed in my software repository. It does not take many moments, and then the product must be installed, which means that the individual packages (there are 6 in all) must be downloaded and installed. It happens quite by itself with the program Software:



Once installation is complete, you can start the server with the following command:

```
sudo service mysqld start
```

and then the server will automatically start every time you start the machine. You can test that the server is running with the following command:

```
sudo service mysqld status
```

After that *MySQL* in principle is been installed and running. What is lacking is a kind of configuration which is done using a script. After *MySQL* is installed, there is created a database administrator called *root* and is the only database user. As part of the installation the script has auto-generated a password for this user that is stored in the file */var/log/mysqld.log*. You can for example find this password with the command

```
sudo grep 'temporary password' /var/log/mysqld.log
```

Once you have found this password, you must run a script:

```
mysql_secure_installation
```

Here you will be prompted to enter the password, and then select a new password for *root*. The script will then ask you about different things, all of which have to do with security (and also a little clean-up), and you just answer *Yes* to all.

After the script is completed, the database is installed and ready for use.

THE CLIENT TOOL

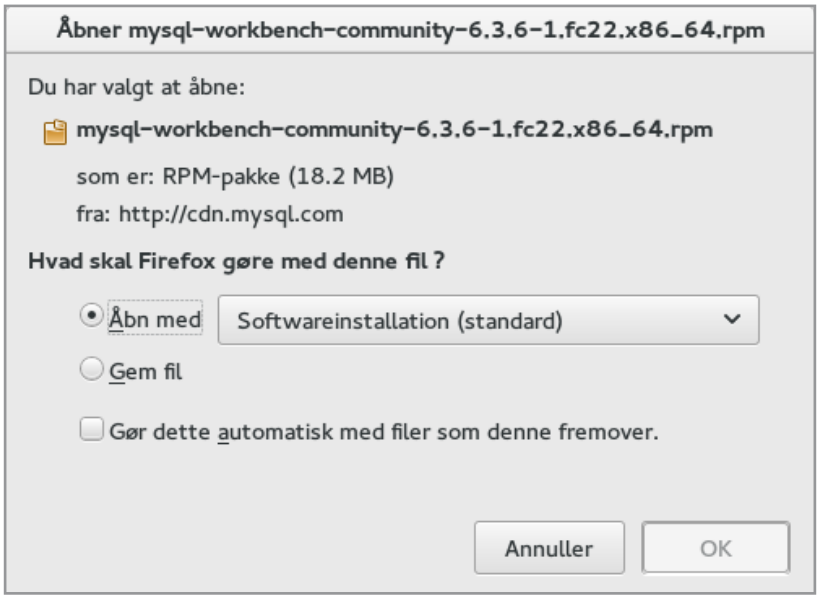
Together with the database server is installed a command-oriented client tool, but there is also a GUI program called *MySQL Workbench*, which should be installed. The program can be downloaded from

http://dev.mysql.com/downloads/workbench/

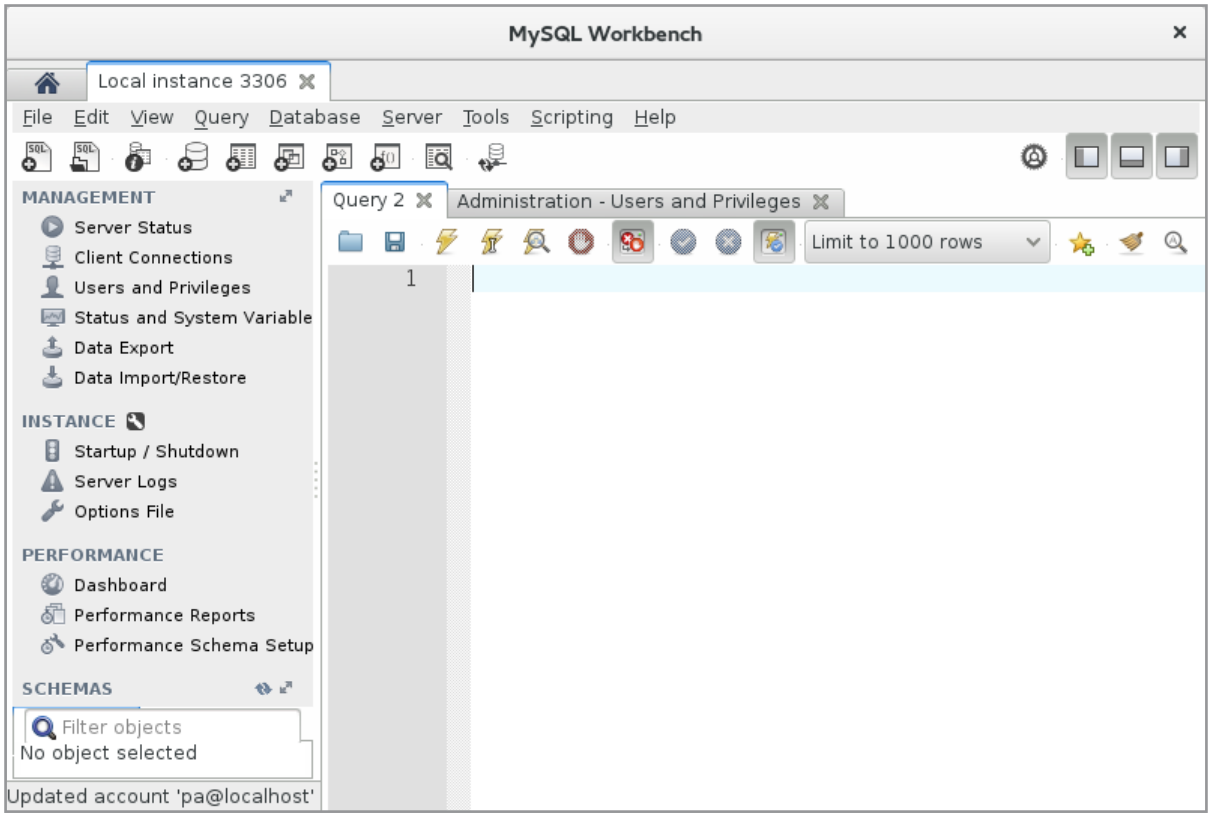
and here one must again choose platform and the product that you want to download. I have chosen

Fedora 23 (x86, 64-bit), RPM Package

and again I get the download window:



If you choose *Software Installation* and click OK, go it all by itself. It takes some time, but then the tool is installed and you've got a desktop icon. If you open the program, you must log in as root with the password that you have just chosen. Then you can work with *MySQL*, administrate the server, create databases, etc. As the first thing you should create a new user and only use *root* if the need arises. I created a user called *pa*, which I have given DBA privileges. It is of course to high for everyday use, but until you learn to work with *MySQL*, it can be quite reasonable.



JDBC

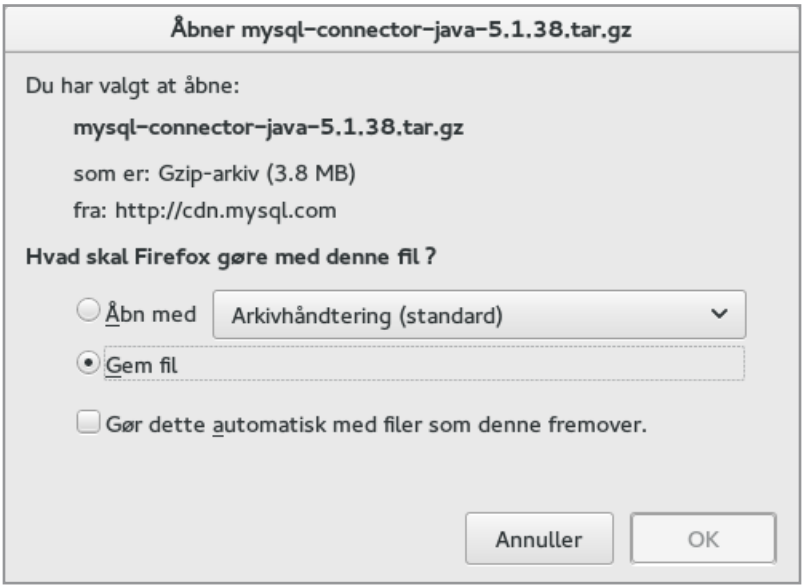
The JDBC driver can be downloaded from:

https://dev.mysql.com/downloads/connector/j/

Here you must choose

Platform Independent (Architecture Independent), Compressed TAR Archive

and in the subsequent download window, this time select *Save File*



The program is downloaded to the folder *Downloads*. Open a terminal window and change the current directory to this directory:

```
cd Downloads
```

The file downloaded is called *mysql-connector-java-5.1.38.tar.gz* (when the number is determined by the version). I've copied the file to another directory (it is not necessary) and made it to my current directory:

```
sudo cp mysql-connector-java-5.1.38.tar.gz /usr/local
cd /usr/local
```

As a next step, the content of the file is unpacked, which is done with the command:

```
tar -zxvf mysql-jdbc.tar.gz
```

The result is a directory *mysql-connector-java-5.1.38*, which contains the driver:

```
cd mysql-connector-java-5.1.38
```

You now must know where your Java runtime system is installed, and it is probably

/usr/java/jdk1.8.0_102

possibly with a different version number, and back is only to copy the driver to the right place:

```
sudo cp mysql-connector-java-5.1.38-bin.jar
  /usr/java/jdk1.8.0_102/jre/lib/ext
```

Then you are ready to write database applications in Java.

This applies at least as long as you execute the programs from NetBeans, but if you are performing programs from a terminal, they may not run, and you are told that Java can not find the JDBC driver. The reason is that Fedora (and other Linux distributions) by default uses *openjdk* and not Oracle's Java. They are two solutions

1. To ensure that the JDBC driver is available for openjdk
2. To ensure that Oracle java is used as default

I will show how to select the second solution. First I entered the folowing command:

```
$ java -version
openjdk version "1.8.0_111"
OpenJDK Runtime Environment (build 1.8.0_111-b16)
OpenJDK 64-Bit Server VM (build 25.111-b16, mixed mode)
```

that shows that the current runtime system is openjdk. Then I add an alternative for runtime system:

```
$ sudo alternatives --install /usr/bin/
java java /usr/java/latest/bin/java 1
[sudo] adgangskode for pa:
```

Then I select the new runtime system (Oracle's java) as default:

```
$ sudo update-alternatives --config java

There is 3 programms that supplies "java".

Select  Command
-----------------------------------------------
*+ 1   /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.111-1.
b16.fc23.x86_64/jre/bin/java
  2   /usr/java/jdk1.8.0_111/jre/bin/java
  3   /usr/java/latest/bin/java

Press enter for the current selection[+], or
enter the number for the selection: 3
```

APPENDIX B

The following is a short introduction to SQL, and the purpose is to show the syntax for the most important statements. Now SQL is not just SQL, and the different database vendors have their variants of the language, and although the following examples are tested against a MySQL database server, they will practically all could be used by any database server. Thus, it will be a number of extensions to SQL, which is not dealt with, and the following should not be percieved as a complete manual for SQL.

Before addressing the actual SQL language, I will mention a few basic database concepts.

A database is a collection of related tables, and a table contains data organized into columns and rows. As an example is shown a part of a database table for books, where there are seven columns:

#	isbn	title	edition	year	pages	pubnr	catnr
1	0-07-054225-2	Functional Analysis	1	1973	397	3	2
2	0-07-222897-0	C++ from the Ground Up	3	2003	602	3	1
3	0-12-296050-5	Homotopy Theory, An Introduction to Algebraic Topology	1	1975	368	10	2
4	0-13-019859-5	Object-Oriented Programming in Java	1	2001	641	2	1
5	0-13-028418-1	Perl, How to Program	1	2001	1057	2	1
6	0-13-032377-2	Java for Students	3	2002	634	2	1
7	0-13-033370-0	Object-Oriented Problem Solving Java, Java, Java	2	2003	862	2	1
8	0-13-034151-7	Java, How to Program	4	2000	1546	2	1
9	0-13-044929-6	Objects First With Java	1	2003	286	2	1

Each column has a name, and the name should be unique within each table. Each column has a data type that determines which data the column can contain. A row contains data about a particular book, and the row's value in a column is called a field and must match the column's data type. A row is also called for a record. The value of a particular field can be *null*, which simply means that the field has no value, and it is important to note that it is not the same as 0 or blank.

A column can be assigned constraint's that one can think of as conditions, the values in that column must meet. If they do not, the database management system is rejecting the database operation. The most important constraints are:

- *NOT NULL*, which indicates that a column can not contain null values, and the row thus must have a value in that column.
- *DEFAULT*, where you can specify a default value that is used, if there is not given a value for the column.
- *UNIQUE*, which means that the column's values must be unique.
- *PRIMARY KEY*, indicating that all rows in this column should have a unique value that identifies the record.
- *FOREIGN KEY*, which defines a reference to a primary key in another table (possibly the same table). The column's value must be the value of a primary key in the other table or *null*.
- *CHECK*, where it is possible to define that the column's values must satisfy a condition.
- *INDEX*, defines an index (table of content) for the column's values.

The database management system is the family of programs that maintains the database's databases, and is a daemon that constantly runs in the background and waiting for requests from users in form of SQL commands. In addition to performing these commands, it is also the database management system that validates that the commands can be executed properly and in this context tests whether the above constraints are respected. The database management system has to ensure the integrity of the database, and you talk about the following integrity rules:

1. *Entity integritet*, which mean that no rows must have a NULL value in the primary key column and all values in this column are different.
2. *Domain integritet*, which ensures that the values in a column are in accordance with the type of the definition and the constraints.
3. *Referential integritet*, which guarantees that a row can not be deleted if the database contains a different row (typically in another table), which refers to this row.
4. *User defined integritet*, where it is possible to define special rules that do not fall under the three above rules.

These rules help to ensure that data in the database always contains legal values, but they do not ensure efficiency, which is a matter of how the database is designed and created. There are several guidelines for good database design, and one of them is called normalization. Very briefly, it is a variety of conditions a database design must meet. Usually one uses only the first three (but there are several), and speaking about that the database is in third normal form. I will not mention normalization of databases here, but refer to books or articles on database theory or the book Java 7.

A BOOK DATABASE

In order to show specific SQL commands I needs a database to be able to perform them. I want to use a database which consists of 5 tables. The primary table contains information about books:

- the books ISBN
- the books title
- the books edition
- the books publisher year
- the number of pages in the book

Books are published by a publishing house, and there is also a publishing table, that a book can refer to by a foreign key. Similarly the books are divided into categories, and there is a category table as a book can refer to. Final has a book one or more authors whose names are stored in a table author, but as the same author may have written several books, it is necessary with a table that can relate books and authors. Correspondingly, the database can be created with the following script called *Books.sql*:

```
use sys;
drop database if exists books;
create database books;
use books;

drop table if exists written;
drop table if exists author;
drop table if exists book;
drop table if exists publisher;
drop table if exists category;

create table category (
catnr       int     not null,
name        varchar(30) not null,
primary key (catnr));
```

```
create table publisher (
pubnr       int      not null,
name        varchar(40) not null,
primary key (pubnr));

create table author (
autnr       int      not null,
firstname   varchar(50),
lastname    varchar(20) not null,
primary key (autnr));

create table book (
isbn        char(13)    not null,
title       varchar(100) not null,
edition     int,
year  char(4),
pages       int,
pubnr       int       not null,
catnr       int,
primary key (isbn),
foreign key (pubnr) references publisher (pubnr),
foreign key (catnr) references category (catnr));

create table written (
isbn        char(13)    not null,
autnr       int     not null,
primary key (isbn, autnr),
foreign key (isbn) references book (isbn),
foreign key (autnr) references author (autnr));
```

A script is simply a text file containing SQL commands. If the file is opened in *MySQL Workbench*, and performed the script creates the database. Preliminary ignore the individual commands because they are all dealt with below.

After the script is done, you have an empty database, but you must have something to do with, and you must add data to the database. In the folder for this book there are 5 text files with data for the 5 tables:

Category.txt
Publisher.txt
Author.txt
Book.txt
Written.txt

If you copy these files to the */var/lib/mysql-files/*data (requires you are *root*) so the database can be loaded with data using the following script (*LoadBooks.sql*):

```
use books;

load data infile '/var/lib/mysql-files/data/Category.txt'
into table category
fields terminated by '\t'
lines terminated by '\r\n';

load data infile '/var/lib/mysql-files/data/Publisher.txt'
into table publisher
fields terminated by '\t'
lines terminated by '\r\n';

load data infile '/var/lib/mysql-files/data/Author.txt'
into table author
fields terminated by '\t'
lines terminated by '\r\n';

load data infile '/var/lib/mysql-files/data/Book.txt'
into table book
fields terminated by '\t'
lines terminated by '\r\n';

load data infile '/var/lib/mysql-files/data/Written.txt'
into table written
fields terminated by '\t'
lines terminated by '\r\n';
```

The result is that you now have a simple database with 5 tables.

SQL DATA TYPES

As shown in the above script that creates the database *books*, then each column has a data type. There are the following options:

- *Int*, that can contain an integer between -2147483648 and 2147483647.
- *Smallint*, that can contain an integer between -32768 and 32767.
- *Tinyint*, that can contain an integer between 0 and 255.
- *Bit*, there is 0 or 1 (in MySQL you can also write *Bool*, that means the same).
- *Bigint*, that can contain an integer between the two integers -9223372036854775808 and 9223372036854775807.
- *Numeric* (or *Decimal*) to decimal numbers between -10^38 +1 and 10^38 -1
- *Money* to currency values and can contain a value between -922337203685477.5808 and 922337203685477.5807

- *Smallmoney* to currencies values from -214748.3648 to 214748.3647
- *Float* to floating points from -3.40E + 38 to 3.40E + 38 (the type is also called *Real*).
- *Double* to floating points from -1.79E+308 to 1.79F+308.
- *Datetime* to timekeeping from 1. January 1753 to 31 December 9999.
- *Smalldatetime* to timekeeping from 1. January 1900 to 6. June 2079.
- *Date* to dates on the form YYYYMMDD
- *Time* to time on the form HHMMSS
- *Char* to text of a fixed length with max 255 characters, where the field is filled with blanks.
- *Varchar* to text of variable length, which can be up to 65535 characters.

In addition, there are two types *Text* and *Blob*, which fills the same (the same capacity), but are used respectively to text and binary data. They come in several varieties:

- *Tinytext* and *Tinyblob* with space for max 255 bytes.
- *Text* and *Blob* with space for max 65535 bytes.
- *Mediumtext* and *Mediumblob* with space for max 16777215 bytes.
- *Longtext* and *Longblob* with space for max 4294967295 bytes.

The above applies to MySQL, but other databases can have several data types.

In addition to data types, there are similar to other programming languages operators, and in terms to arithmetical operators and operators for comparison the syntax is almost the same as in Java. However, there are some other operators, as I will explain in connection with examples, but the list is as follows:

- ALL
- AND
- ANY
- BETWEEN
- EXISTS
- IN
- LIKE
- NOT
- OR
- IS NULL
- UNIQUE

SQL COMMANDS

SQL is a command language, and there are the following commands:

- CREATE, that is used to create database objects such as tables
- ALTER, that is used to modify existing objects, such as tables
- DROP, that is used to delete objects such as tables

These commands are called DDL commands for *Data Definition Language*.

- GRANT, that is used to assign the rights for users
- REVOKE, that is used to remove rights from users

These commands are called DCL commands for *Data Control Language*.

- INSERT, which is used to insert a row in a table
- UPDATE, which is used to modify the content of one or more rows in a table
- DELETE, which is used to delete one or more rows in a table

These commands are called DML commands for *Data Manipulation Language*.

- SELECT, which is used for extracting rows from one or more tables

This command is called a DQL command for *Data Query Language*.

There are as such 9 commands so SQL should be easy to learn, and it also is, and it is only the last, which is complex with a very comprehensive syntax.

Note first that SQL does not differentiate between small and capital letters, but in the following, I generally write SQL names with capital letters and it is only because it for the reader should be clear what is SQL key words.

As an example is shown a command that creates a database:

```
CREATE DATABASE books;
```

If you want to delete the database again, you can do it with the command

```
DROP DATABASE books;
```

One must of course be wary of a command as above, that you do not delete a database with data – unless this is exactly what you want.

If the database management system is used to manage multiple databases (and it will always be the case), and you opens MySQL Workbench, you must tell which database the SQL commands should works on. You do this with a command like

```
USE books;
```

which makes the database *books* to the current database.

Is the database *books* the current database, you can create a table as follows:

```
CREATE TABLE publisher (
pubnr       INT      NOT NULL,
name        VARCHAR(40) NOT NULL,
PRIMARY KEY (pubnr));
```

Here you creates a table named *publisher*. The table has two columns, where the first is called *pubnr* and has the type *INT* and thus must contain integers, while the other is called *name* and has type *VARCHAR* to hold max 40 characters and then can contain text. Both columns are defined NOT NULL, and a row must have a value for both *pubnr* and *name*. Finally is defined that the column *pubnr* must be primary key. The command is written on several lines. It is not necessary and is only done for the sake of readability, but you should note where there is a comma, as they must be there. The command in question may in addition also be written as follows:

```
CREATE TABLE publisher (
pubnr       INT      PRIMARY KEY,
name        VARCHAR(40) NOT NULL );
```

When I have not specified NOT NULL for the column *pubnr*, it is because a column defined as a primary key will automatically be NOT NULL.

Below is another example of a command that creates a table:

```
CREATE TABLE book (
isbn        CHAR(13)    NOT NULL,
title       VARCHAR(100) NOT NULL,
edition     INT,
year        CHAR(4),
pages       INT,
pubnr       INT,       NOT NULL,
catnr       INT,
PRIMARY KEY (isbn),
FOREIGN KEY (pubnr) REFERENCES publisher (pubnr),
FOREIGN KEY (catnr) REFERENCES category (catnr));
```

Here you creates a table *book* with 7 columns, which should contain information about a book. Here is the first column is primary key. The next column is defined NOT NULL corresponding to that a book must have a title. The next three columns allow NULL values because you are welcome to create a book without knowing when the book is released, the edition and the page number. The column *pubnr* is foreign key to the *publisher* table and is defined NOT NULL. This means that a book must have a publisher. The column *catnr* is also a foreign key (to the table *category*), but it may well be null. This means that a book need not be assigned to a category, but if there is a value in the column, it must be the number of an existing category.

If you want to delete a table, the syntax is

```
DROP TABLE book;
```

which deletes the table *book* with all its data. You should note that if you instead tries to execute the command

```
DROP TABLE publisher;
```

would get an error when the table *book* via the foreign key refers to the table *publisher*.

It is also possible to modify database objects. As an example the following command adds a new column to the table book:

```
ALTER TABLE book ADD info VARCHAR(1024);
```

Similarly, the following command is used to change the data type of the new column:

```
ALTER TABLE book MODIFY info Text;
```

If you want to delete the new column again, you can do it with the following command:

```
ALTER TABLE book DROP COLUMN info;
```

If, for example it is such that the book's *edition* must be less than 20, you can execute the command:

```
ALTER TABLE book ADD CONSTRAINT editionCheck CHECK (edition < 20);
```

where the constraint has been given a name, so you can refer to it in SQL commands. Subsequently, one could modficere this constraint as follows:

```
ALTER TABLE book ADD CONSTRAINT editionCheck
CHECK (edition IS NULL OR (edition BETWEEN 1 AND 19));
```

there would be a more accurate control.

As a final example the below shows a command that changes the name of a column:

```
ALTER TABLE book CHANGE edition edit INT;
```

I mention the command because it can be useful, but also because it is an example of a command whose syntax depends on the current database management system.

There are many other options with the command ALTER but the above gives an impression of the syntax and what is possible.

DML COMMANDS

The above commands are all examples of DDL commands and thus commands that modify the structure of the database. The main use of these commands are in scripts that create databases. In this section I will show the syntax of the three DML commands, and unlike DDL commands, these are commands that change the content of the tables.

In this section and also in the rest of this appendix, it is assumed that the database *books* are as described in the introduction and is loaded with data as described there.

I will start with the command INSERT, that insert a row in a table. As an example, the following command inserts a row in the *author* table:

```
INSERT INTO author VALUES (186, 'Mogens', 'Trolle');
```

In order that the command can be executed, there are two requirements:

1. After *VALUES* must be in parentheses a value for each of the three columns in the table and the types must match.
2. The value of *autnr* (the value 186) must not already exist – the column are primary key

In many ways it is the simplest INSERT command, as one can imagine.

Often, the insertion of a row, however, require the insertion of rows in multiple tables. If for example, I want to create the book

78-7900-910-7, Afrikas dyreliv publiched on *Globe* as 1. edition and written by *Mogens Trolle*

then the publisher does not exists and must first be created. When the author is already created (with the command above), there must also be added a row to the table *written* and the book can be created as follows:

```
INSERT INTO publisher VALUES (21, 'Globe');
INSERT INTO book (isbn, title, edition, pubnr)
 VALUES ('87-7900-910-7', 'Afrikas dyreliv', 1, 21);
INSERT INTO written VALUES ('87-7900-910-7', 186);
```

The first command requires no additional comments beyond again to be aware that publisher number 21 must not already exist.

The second command does not insert values in all columns, and therefore you must specify in which columns to be inserted values. This is done by listing the column names in parentheses after the table name. Then VALUES must specify values for these columns. You should note that *isbn* is necessary because it is the primary key. Also the *title* is necessary, when this column is defined NOT NULL and it is ok that there is no value for *year* and *pages*, as the columns may contain NULL values. *pubnr* is defined NOT NULL, and you must indicate a publishing number, and when the column is a foreign key to the table *publisher* it must be a number on an existing publishers. It is therefore necessary, that the command that creates the publisher is executed before the command that creates the book. Note, finally, that there is not defined a category, although this column (column *catnr*) is a foreign key to the table *category*. It is not necessary, because the column allows NULL values, which in turn corresponds to, that a book does not need to have a category.

Then there is finally the last command, as there is not much to say, but you should note that the two values is a composite key, and each part of the key is a foreign key, respectively to the table *book* and table *author*.

The command UPDATE is used to change the values in the columns, and by way of the example, the following command updates a row in table book:

```
UPDATE book SET year = '2010', pages = 255 WHERE isbn = '87-7900-910-7';
```

The command is easy enough to understand. After SET follows a comma-separated list of column names and values, where the values are the new values. Then there is the WHERE clause, which is followed by a condition that determines which rows to be modified. A WHERE clause can be extremely complex, which also will appear from the following about SELECT, but in this case it defines exactly one row. A WHERE may well define multiple rows, and where appropriate, all the rows that satisfy the condition are updated. You must specifically note that it is not a requirement that an UPDATE command has a WHERE part, and if not, all the table's rows are updated. Also note that you can not change the value of the primary key.

As another example the following command shows how to set value in a column to NULL:

```
UPDATE book SET edition = NULL WHERE isbn = '87-7900-910-7';
```

The command assumes of course that the column allows NULL values.

Then there is the DELETE command that is used to delete rows. Suppose you have performed the following command:

```
INSERT INTO category VALUES (10, 'Test category');
```

that inserts a row in the table *category*. If you want to delete the row again, you can do it in the following way:

```
DELETE FROM category WHERE catnr = 10;
```

So it is simple to delete rows in a table, but there are a few things you should be aware of. First, the WHERE part can specify multiple rows, and if so all rows that satisfy the condition after the WHERE clause are deleted. Moreover, it is allowed to completely omit the WHERE clause, and if so, the command deletes all rows in the table. One should therefore be careful with DELETE, since it's easy to delete more than the thought is.

In this case, the table *book* has a foreign key to the table *category*, and if there is a book that refers to the *category* with the key 10, the row is not deleted. It is at least the default, but there are other options. In the table *book* could have defined foreign key as follows:

```
FOREIGN KEY (catnr) REFERENCES category (catnr) ON DELETE SET NULL
```

Where appropriate, the references with value 10 for *catnr* in the table *book* automatically be set to NULL. Another option is to write:

```
FOREIGN KEY (catnr) REFERENCES category (catnr) ON DELETE CASCACE
```

If so, the rows of the table *book* that refers to the number in the *category* table will also be deleted. Cascade can be important for ensuring the integrity of the database, but there is still reason to warn a little against this clause. If you have defined foreign key *catnr* as ON DELETE CASCADE, and you performs the command

```
DELETE FROM category WHERE catnr = 1;
```

then you deletes not only a row in the table *category*, but you also deletes all rows in the table *book* that refers to this row, and that is the vast majority (if not som of them are prevented to be deleted by a foreign key in the table *written*).

THE SELECT COMMAND

Then there is the SELECT command, which absolutely is the command where there is most to say. The simplest command you can think of is a command of the form:

```
SELECT * FROM publisher;
```

which extracts all rows from the table *publisher*. A * means all columns and hence the result consists of rows that should contain values from all columns. You can replace * with the names of the columns whose values you wants to show:

```
SELECT title, pages FROM book;
```

A SELECT can have a WHERE part, where you specify the rows to extract:

```
SELECT title, pages FROM book WHERE edition = 4;
```

title	pages
Java, How to Program	1546
Introduction to Java Programming	952
Teach Yourself Visual Basic 5	798
Data Kommunikation	872

The WHERE part can by using boolean operators to define more complex conditions:

```
SELECT title, pages FROM book
WHERE (edition = 1 OR edition = 3) AND pages > 650 AND pages < 700;
```

title	pages
Cryptography and Network Security	681
Programming Languages, Design and Implementation	654
Understanding Java	680
The Object of Java	670
Mastering C# Database Programmeing	665

It is also possible to specify substrings. As an example the following command extracts all rows where title starts with the word Java

```
SELECT title, pages FROM book WHERE title LIKE 'Java%';
```

while the following command extracts all rows where the title contains the word Java

```
SELECT title, pages FROM book WHERE title LIKE '%Java%';
```

The rule is that % matches 0 or more arbitrary characters. As another example extracts the command

```
SELECT title, year, pages FROM book WHERE year LIKE '19__';
```

all rows where the year (publisher year) starts with 19 and is followed by two characters, since the rule is that _ exactly matches one character.

The following command extracts the first three titles in the table book:

```
SELECT title FROM book LIMIT 3;
```

LIMIT can also be combined with a WHERE:

```
SELECT title FROM book WHERE edition = 2 LIMIT 3;
```

You should be aware that other database systems instead of LIMIT uses SELECT TOP.

A SELECT can have an ORDER BY, where the rows are sorted according to values in a particular column. Thus, the following command sorts the rows by title in ascending order:

```
SELECT isbn, title FROM book ORDER BY title;
```

If you wish instead that the rows are sorted in descending order, one can write:

```
SELECT isbn, title FROM book ORDER BY title DESC;
```

You can also sort by multiple criteria. Consider the following command, which extracts *isbn*, *title* and *edition* for all rows where the category number is 1, but so that the rows first are sorted by *edition*, and within each edition by the title:

```
SELECT isbn, title, edition FROM book WHERE catnr = 1
 ORDER BY edition, title;
```

GROUP BY

For a SELECT you can also define GROUP BY, which indicates that the result should contain one row for each value in the column that will be grouped. For example

```
SELECT edition FROM book GROUP BY edition;
```

that will show all editions. There are six rows with the numbers 1, 2, 3, …, 6. It is not so interesting for the same result could be achieved in other ways, and GROUP BY has also only of interest if you want to do something by the rows that fall within the individual groups. If, for example you were interested in determining the sum of all pages distributed on edition you could use the command:

```
SELECT edition, SUM(pages) FROM book GROUP BY edition;
```

edition	SUM(pages)
NULL	255
1	35187
2	10333
3	3527
4	4168
5	1163
6	654

You should note that the result shows 7 rows. This is because there is a book in which the value of *edition* is NULL.

GROUP BY can be combined with both WHERE and ORDER BY, and the order must be as shown below:

```
SELECT edition, SUM(pages), COUNT(*) FROM book
WHERE pages < 1000 GROUP BY edition ORDER BY edition DESC;
```

Here is selected the group (the edition), the sum of all pages within the group, the number of rows within the group, but only for the books where the page number is less than 1000:

edition	SUM(pages)	COUNT(*)
6	654	1
4	2622	3
3	3527	7
2	6901	9
1	23374	52
NULL	255	1

A WHERE clause defines which rows to be extracted, and you can therefore think of the WHERE clause as a filter that filters the rows. Similarly with GROUP BY there is a HAVING clause, that is a filter that specifies which groups to include. The syntax is as follows:

```
SELECT edition, SUM(pages), COUNT(*) FROM book
WHERE pages < 1000 GROUP BY edition
HAVING edition = 2 OR edition = 4 OR edition = 6 ORDER BY edition DESC;
```

where the HAVING clause specifies that only groups with a value of 2, 4 or 6 should be extracted:

edition	SUM(pages)	COUNT(*)
6	654	1
4	2622	3
2	6901	9

If you execute the command

```
SELECT edition FROM book;
```

you get shown all editions, and there are 87 rows. This means that the same value appears several times. If you do not want that, you can write

```
SELECT DISTINCT edition FROM book;
```

which simply means that all rows must be different.

JOIN

It is also possible to perform a SELECT command that extracts rows from multiple tables, and we often talk about a JOIN. As an example is shown a JOIN command, that extracts the title from the *book* table and the *name* from the publishers table, but such that there only are extracted a row where the value in column *pubnr* is similar in the two tables:

```
SELECT title, name FROM book, publisher WHERE book.pubnr = publisher.pubnr;
```

Put slightly differently, so are each row of the *book* table combined with each row in the *publisher* table, but only rows with same value in the columns for *pubnr* are included in the result. The result has then 87 rows with two columns. You should note that the two columns names *title* and *name* are unique determined in the combination of the two tables but both tables has a column named *pubnr*, and therefore it is necessary to qualify the name with the table name in the WHERE part.

If you do not specify concrete columns in a join such as

```
SELECT * FROM book, publisher WHERE book.pubnr = publisher.pubnr;
```

the result is still the 87 rows, but all columns from both tables is there

isbn	title	editi	year	pages	pubi	catr	pu	name
0-201-10194-7	Compilers, Principles, Techniques a…	1	1986	796	1	1	1	Addison Wesley
0-201-61272-0	Introduction to Programming Using …	1	2000	805	1	1	1	Addison Wesley
0-201-63361-2	Design Patterns	1	1995	395	1	1	1	Addison Wesley
0-201-70267-3	Applying Enterprise JavaBeans	1	2001	436	1	1	1	Addison Wesley
0-201-71107-9	Understanding Java	1	2001	680	1	1	1	Addison Wesley
0-201-71585-6	The Object of Java	1	2002	670	1	1	1	Addison Wesley

Note especially that the column *pubnr* is there twice.

There are several types of JOIN on tables, and an example as the above is called an INNER JOIN and can also be written as follows:

```
SELECT title, name FROM book INNER JOIN publisher
ON book.pubnr = publisher.pubnr;
```

An INNER JOIN between two tables returns rows in which there are matches in both tables. An INNER JOIN (and all other JOIN operations) can also be combined with a WHERE clause as such

```
SELECT title, name FROM book INNER JOIN category ON
book.catnr = category.catnr WHERE isbn LIKE '87-%';
```

that returns the title and category name for the books where isbn starts with 87- (all Danish titles):

title	name
Data Kommunikation	Computer teknologi
Active Server Pages	Computer teknologi
Javascript i praksis	Computer teknologi
Objekt orienteret analyse og design	Computer teknologi
ASP, Active Server Pages	Computer teknologi
Symbolsk Maskinsprog, Univac 100…	Computer teknologi

In addition to INNER JOIN, there are the following JOIN operations:

- *LEFT JOIN*, which returns all rows from the left table, even if there is no match in the right table.
- *RIGHT JOIN*, which returns all rows of the right table, even if there is no match in the left table.
- *FULL JOIN*, which returns all rows even if there is no match in one of the two tables.
- *CARTESIAN JOIN*, that returns the cartesian product of the two tables.

You should be aware that a FULL JOIN is not supported by MySQL. As an example I will show a LEFT JOIN:

```
SELECT title, name FROM book LEFT JOIN category ON
book.catnr = category.catnr WHERE isbn LIKE '87-%';
```

Note that in principle it is the same JOIN as above, but this time the result shows seven rows. This is because the *book* table has a row, which is NULL in the column *catnr* and therefore does not match a row in the *category* table, but it is included in the result as opposed to an INNER JOIN where there must be a match in both tables.

Below is a RIGHT JOIN:

```
SELECT title, name FROM book RIGHT JOIN category
ON book.catnr = category.catnr
WHERE isbn LIKE '87-%';
```

It also shows 6 rows (that is the same result as the corresponding INNER JOIN) because the table *category* does not have rows with a value in column *catnr*, which does not match a row in the *book* table.

The next command is again an example of a INNER JOIN:

```
SELECT title, name FROM book, publisher
WHERE book.pubnr = publisher.pubnr AND isbn LIKE '87-%';
```

and the result is 7 rows. If you delete the JOIN condition, you get a CARTESIAN JOIN:

```
SELECT title, name FROM book, publisher WHERE isbn LIKE '87-%';
```

and the result is 147 rows. The reason is that each row of the table *book* (there are 7 meeting the WHERE clause) is combined with all rows in the *publisher* table (which is 21), and the result is a total of 7 × 21 = 147 rows.

It is also possible to join more than two tables. The following command is a JOIN between three tables showing for each *title*, where the *isbn* starts with 87 a row with the *title* and author's *name* (one row for each author):

```
SELECT title, firstname, lastname FROM book
INNER JOIN written ON book.isbn = written.isbn
INNER JOIN author ON written.autnr = author.autnr
WHERE book.isbn LIKE '87-%';
```

By combining (join) *book* and *written* you get for each book the author numbers for the book's authors, and by combining this result with the table *author* you can obtain the authors' names. The command in question can also be written as follows:

```
SELECT title, firstname, lastname FROM book, written, author
WHERE book.isbn = written.isbn AND written.autnr = author.autnr
AND book.isbn LIKE '87-%';
```

As another example, shows the following command isbn and title of all mathematics books published by Prentice Hall:

```
SELECT isbn, title FROM book, category, publisher
WHERE book.catnr = category.catnr AND book.pubnr = publisher.pubnr
AND category.name LIKE '%Matematik%'
AND publisher.name LIKE '%Prentice Hall%';
```

As appears from the JOIN operations NULL values can sometimes lead to unexpected results. Although it has nothing to do with joins, please note the following syntax

```
SELECT * FROM book WHERE edition IS NULL;
```

which returns all books whose value for *edition* is NULL. Similarly, one can write

```
SELECT * FROM book WHERE edition IS NOT NULL;
```

When you joins several tables that can occur name matches where two columns has the same name. As shown above, one can solve this problem by qualifying the name with the table name. It can lead to a somewhat clumsy syntax, and additionally, it may mean that there are two columns in the result, with the same name. To solve these problems, you can use the concept of an alias, where you can give a table or a column a custom name. The following command determines the title, publisher name and category name for all Danish books:

```
SELECT title AS Text, P.name AS 'Publisher
name', C.Name AS 'Category name'
FROM book AS B, publisher AS P, category AS C
WHERE B.pubnr = P.pubnr AND B.catnr = C.catnr AND isbn LIKE '87-%';
```

There is extracted data from three tables: *book, publisher* and *category*, but these tables are assigned names *B*, *P* and *C*. In addition are each of the three columns assigned a name, and you should note that this names are used in the result:

Text	Publisher name	Category name
Data Kommunikation	Teknisk Forlag	Computer teknologi
Active Server Pages	Teknisk Forlag	Computer teknologi
Javascript i praksis	Teknisk Forlag	Computer teknologi
Objekt orienteret analyse og design	Marko	Computer teknologi
ASP, Active Server Pages	IDG	Computer teknologi
Symbolsk Maskinsprog, Univac 10…	M. S. Databøger	Computer teknologi

SET OPERATIONS

If two SELECT commands are union compatible, that is they results in the same number of columns, columns of the same types, and the columns in the same order, one can perform the UNION:

```
SELECT isbn, title, edition FROM book WHERE isbn LIKE '87-%'
UNION
SELECT isbn, title, edition FROM book WHERE edition = 3 OR edition = 4;
```

The first command returns 7 rows and the second 11 rows. Because UNION removes duplicate rows, and there are two rows that are identical, the command will result in 16 rows. If you instead writes

```
SELECT isbn, title, edition FROM book WHERE isbn LIKE '87-%'
UNION ALL
SELECT isbn, title, edition FROM book WHERE edition = 3 OR edition = 4;
```

are duplicate rows retained, and the command will result in 18 rows.

Instead of UNION you can write INTERSECT, and the result is the intersection. This clause is not supported by MySQL. It is also possible to write EXCEPT that means set difference, but this clause is not supported by MySQL.

SQL FUNCTIONS

As part of SQL are a number of functions that can be used in SQL commands. Above I have already used the SUM() and COUNT(). The following is a listing of the most important of those functions, and there are many, but they can be divided into groups:

1. general functions
2. numeric functions
3. functions to strings
4. functions to date and time

and in addition there is a (sometimes large) number of functions, depending on the database product.

GENERAL FUNCTIONS

This group of functions include:

- COUNT
- MAX
- MIN
- SUM
- AVG

and are the classic functions, where I above has used COUNT and SUM. As an example determines the following command the number of books, the total number of pages (total number of pages of all the books), the average number of pages, the smallest number of pages and the largest number of pages:

```
SELECT COUNT(*), SUM(pages), AVG(pages), MIN(pages), MAX(pages) FROM book;
```

COUNT(*)	SUM(pages)	AVG(pages)	MIN(pages)	MAX(pages)
87	55287	635.4828	71	1811

NUMERIC FUNCTIONS

ABS	ACOS	ASIN	ATAN	ATAN2	BIT_AND
BIT_COUNT	BIT_OR	CEIL	CEILING	CONV	COS
COT	DEGREES	EXP	FLOOR	FORMAT	GREATEST
INTERVAL	LEAST	LOG	LOG10	MOD	OCT
PI	POW	POWER	RADIANS	RAND	ROUND
SIN	SQRT	STD	STDDEV	TAN	TRUNCATE

The meaning of most of the functions is indicated by the name, and you can not be sure that all database systems implements all this functions. Consider as an example the following statements:

```
SET @my = 0;
SELECT @my := AVG(pages) FROM book;
SELECT RAND(), PI(), SQRT(2), STDDEV(pages),
SQRT(SUM((pages - @my) * (pages - @my)) / COUNT(*)) AS Sigma FROM book;
```

RAND()	PI()	SQRT(2)	STDDEV(pages)	Sigma
0.4956075712066214	3.141593	1.4142135623730951	361.43249038379736	361.43249038379724

Here are the results of the first three functions simple enough, while the fourth function determines the standard-deviation of the books pages. The first statement defines a variable and the next statement set this variable equal to the average (mean value) of the number of pages. The value of this variable is used in the last formula, which calculates the standard deviation of the number pages. It is of course no reason for that calculation, because there is a function STDDEV that do the same, but the example should partly show how you can use a variable and partly an example of a complex expression.

FUNCTIONS TO STRINGS

ASCII	BIN	BIN_LENGTH
CHAR_LENGTH	CHARACTER_LENGTH	CONCAT_WS
CONCAT	CONV	ELT
EXPORT_SET	FIELD	FIND_IN_SET
FORMAT	HEX	INSERT
INSTR	LCASE	LEFT
LENGTH	LOAD_FILE	LOCATE
LOWER	LPAD	LTRIM
MAKE_SET	MID	OCT
OCTET_LENGTH	ORD	POSITION
QUOTE	REGEXP	REPEAT
REPLACE	REVERT	RIGHT
RPAD	RTRIM	SOUNDEX
SOUNDEX_LIKE	SPACE	STRCMP
SUBSTR	SUBSTRING	SUBSTRING_INDEX
TRIM	UCASE	UNHEX
UPPER		

Most of the functions are easy enough to understand, but not all, and there it is necessary to look up the meaning for the function. As an example determines the following statement the title's length and the first 10 characters in the title of all books whose title contains the word Java:

```
SELECT LENGTH(title), SUBSTR(title, 1, 10) FROM book
WHERE title LIKE '%Java%';
```

FUNCTIONS TO DATE AND TIME

As discussed above, SQL types for dates clould be difficult to use correct, and there are a number of functions that specifically are used for dates:

ADD_DATE	ADD_TIME	CONVERT_TZ
CURDATE	CURRENT_DATE	CURRENT_TIME
CURRENT_TIMESTAMP	CURTIME	DATE_ADD
DATE_FORMAT	DATE_SUB	DATE
DATE_DIFF	DAY	DAYNAME
DAYOFMONTH	DAYOFWEEK	DAYOFYEAR
EXTRACT	FROM_DAYS	FROM_UNIXTIME
HOUR	LAST_DAY	LOCALTIME
LOCALTIMESTAMP	MAKEDATE	MAKETIME
MICROSECOND	MINUTE	MONTH
MONTHNAME	NOW	PERIOD_ADD
PERIOD_DIFF	QUATER	SEC_TO_TIME
SECOND	STR_TO_DATE	SUBDATE
SUBTIME	SYSDATE	TIME_FORMAT
TIME_TO_SEC	TIME	TIMEDIFF
TIMESTAMP	TIMESTAMPADD	TIMESTAMPDIFF
TO_DAYS	UNIX_TIMESTAMP	UTC_DATE
UTC_TIME	UTC_TIMESTAMP	WEEK
WEEK_DAY	WEEKOFYEAR	YEAR
YEARWEEK		

As the table shows, there are many functions and it is not so easy to figure out the meaning of all of them, but the result is that there are many opportunities to manipulate date and time in SQL. As a small example, the following statement use two of this functions:

```
SET @d = CURDATE();
SET @t = CURTIME();
SELECT @d, @T;
```

@d	@T
2016-10-13	00:06:47

VIEW'S

A view is basically nothing more than a SQL statement that is stored in the database, but it can be seen as a form of virtual table, as you in principle can use it in the same way, as you uses other tables. A view can be created on basis of one or more tables, but what you can do with a view is determined by how it is created. The purpose of a view is

to structure the content of a database corresponding to the users' use of the database

limiting access to the data that users can work with

allowing data from multiple tables to appear as a single table

In *MySQL Workbench* you can create a view by right-click on *Views* (the tab SCHEMA) and choose *Create View*. You then gets the following skeleton:

```
CREATE VIEW 'new_view' AS
```

As an example, you can write the following view:

```
CREATE VIEW Prentice_hall AS SELECT isbn, title, edition, year, pages
FROM book WHERE pubnr = 2;
```

which defines a view called *Prentice_hall* that for all books where *pubnr* are 2 extracts the *isbn*, *title*, *edition*, *year* and *pages*. That is the view of columns in the table book of all books published by Prentice Hall. When you then click *Apply,* MySQL Workbench creates the following view:

```
CREATE
  ALGORITHM = UNDEFINED
  DEFINER = 'pa'@'%'
  SQL SECURITY DEFINER
VIEW 'books'.'Prentice_hall' AS
  SELECT
    'books'.'book'.'isbn' AS 'isbn',
    'books'.'book'.'title' AS 'title',
    'books'.'book'.'edition' AS 'edition',
    'books'.'book'.'year' AS 'year',
    'books'.'book'.'pages' AS 'pages'
  FROM
    'books'.'book'
  WHERE
    ('books'.'book'.'pubnr' = 2)
```

This view may then be used in the same way as the other tables, and for example you can write:

```
SELECT * FROM Prentice_hall;
```

Indeed, one can also to some extent perform SQL INSERT, UPDATE and DELETE statements in a view. Consider as an example the following view, which extracts first and last names of all authors where the last name starts with a K:

```
CREATE VIEW knames AS SELECT DISTINCT firstname, lastname
FROM author WHERE lastname LIKE 'K%';
```

If you then performs the statement

```
SELECT * FROM knames;
```

you will see that the view contains 8 rows. If you then performs the following statements:

```
INSERT INTO author VALUES (187, 'Poul', 'Klausen');
select * from knames;
```

you will see that the view *knames* now has 9 rows. This means that after the table *author* is updated is also the view updated.

Next, if you try:

```
INSERT INTO knames VALUES ('Jens', 'Kristensen');
```

you get an error where you are told that the parent table can not be updated. Of course it is not so strange, because there is not enough information (there is not an author number) to create a row in the *author* table. Consider the following view:

```
CREATE VIEW lnames AS SELECT autnr, firatname, lastname
FROM author WHERE lastname LIKE 'L%';
```

that this time defines a column to all columns in the *author* table. Perform now the statement:

```
INSERT INTO lnames VALUES (1000, 'Knud', 'Larsen');
```

and you will see that both the view and the original table is updated. This means that an update of the view also updates the parent table, but there are the following conditions:

- the SELECT statement may not use DISTINCT
- the SELECT statement may not contain functions
- the SELECT statement may not contain operators
- the SELECT statement may not use ORDER BY
- the SELECT statement can only use one table
- the SELECT statement's WHERE part may not use a SELECT statement (se below)
- the SELECT statement may not use GROUP BY or HAVING
- the SELECT statement must include all columns from the parent table that is defined NOT NULL

INNER SELECT STATEMENTS

It is possible to use an inner SELECT statement in a WHERE clause where an inner SELECT returns data that is used in the condition. An inner SELECT can be used both in a SELECT, INSERT, UPDATE and DELETE – typically associated with operators. A typical example is, using SELECT IN:

```
SELECT isbn, title FROM book WHERE pubnr IN
(SELECT pubnr FROM publisher WHERE
 name = 'Prentice Hall' OR name = 'Addison Wesley');
```

The result is *isbn* and *title* of all books published by either Prentice Hall or Addison Wesley. Of course one can achieve the same otherwise, but SELECT IN is easy to understand.

As another example, creates the following script a new table with all the math books when the table must contain the same columns as table book, but except column *catnr*:

```
use books;

create table math (
isbn    char(13)    not null,
title   varchar(100) not null,
edition  int,
year char(4),
pages  int,
pubnr int       not null,
primary key (isbn),
foreign key (pubnr) references publisher (pubnr));

INSERT INTO math (isbn, title, edition, year, pages, pubnr)
SELECT isbn, title, edition, year, pages, pubnr FROM book WHERE catnr = 2
```

What is interesting is the last *INSERT INTO* statement that inserts data from a *SELECT*.

STORED PROCEDURES

I will conclude this appendix with a very brief introduction to stored procedures, which is actually a large area. A stored procedure is a routine consisting of SQL statements that are stored on the database server together with the database tables, and the idea is of course to save the SQL procedures which are often needed, but also that a stored procedure is translated into an internal format and thus is effective.

Below I will show a few examples of simple procedures, and how they are created using MySQL Workbench. In the database right-click on *Stored Procdures* and select *Create Stored Procedure* and MySQL Workbench creates a skeleton:

```
CREATE PROCEDURE 'new_procedure' ()
BEGIN

END
```

You can then write a stored procedure named *Hello* as follows:

```
CREATE PROCEDURE Hello ()
BEGIN
 SELECT 'Hello World';
END
```

and if you then click *Apply* the procedure is translated, and if there is no errors, it is stored in the database and MySQL Workbench will show the finished code:

```
USE 'books';
DROP procedure IF EXISTS 'Hello';

DELIMITER $$
USE 'books'$$
CREATE PROCEDURE Hello ()
BEGIN
 SELECT 'Hello World';
END$$

DELIMITER ;
```

The first thing that happens is that the procedure is deleted if there is already a procedure with the same name. Next, define a punctuation used to mark to MySQL that the procedure ends. By default it is a dollar sign. After the procedure is translated and saved, it can be performed as follows:

```
CALL Hello;
```

and the result is that the procedure prints *Hello World* on the screen.

It is obviously not a particularly interesting procedure, but it shows the principle and that is simply the case that one or more SQL statements can be executed under a common name. In addition to seeing some more examples, there are basically two things that must be addressed, namely parameters and program logic.

As an example is below shown a procedure, that calculate the number og books, where the number of pages is greater than or equal to *a* and less than or equal to *b*, and where *a* and *b* are input parameters. The result is saved in an output parameter *c*:

```
CREATE PROCEDURE 'counter' (IN a int, IN b int, OUT c int)
BEGIN
 SELECT count(isbn) FROM book WHERE a <= pages AND pages <= b INTO c;
END
```

You should notice how the procedure stores the result in c with the operatator INTO. The folowing shows how the procedure can be used to calculates the number of books, where the number of pages is greater than or equal to 200 and less than or equal to 500:

```
use books;
set @num = 0;
call counter(200, 500, @num);
select @num;
```

It is also possible to define a parameter as INOUT. The following procedure determines the average of number of pages in books where the page number is less than or equal to the value of the parameter t:

```
CREATE DEFINER='pa'@'%' PROCEDURE 'average'(INOUT t INT)
BEGIN
 DECLARE s int;
 DECLARE n int;
 SELECT SUM(pages), COUNT(*) FROM book WHERE pages <= t INTO s, n;
 SET t = s / n;
END
```

and the procedure can be used as follows:

```
use books;
set @num = 500;
call average(@num);
select @num;
```

The following script creates a database with one table, that has only one column:

```
use sys;
create database numbers;
use numbers;
create table primes (prime int primary key);
```

To this database I have added a stored procedure that adds the value of a parameter *n* to the table, if *n* is a prime and not already is in the table. The result of the procedure is stored in the parameter *r*.

```
CREATE DEFINER='pa'@'%' PROCEDURE 'isprime'(IN n int, OUT r boolean)
BEGIN
 DECLARE c int;
 SELECT count(*) FROM primes WHERE prime = n INTO c;
 IF c > 0 THEN SET r = false;
 ELSEIF n = 2 OR n = 3 OR n = 5 OR n = 7 THEN SET r = true;
 ELSEIF n < 11 OR mod(n, 2) = 0 THEN SET r = false;
 ELSE
  BEGIN
   DECLARE t int DEFAULT 3;
   DECLARE m double DEFAULT sqrt(n) + 1;
   SET r = true;
   WHILE t <= m AND r = true DO
    IF mod(n, t) = 0 THEN SET r = false;
    ELSE SET t = t + 2;
    END IF;
   END WHILE;
  END;
 END IF;
 IF r = true THEN INSERT INTO primes VALUES (n);
 END IF;
END
```

and below an example of an application of the procedure:

```
use numbers;
set @res = false;
call isprime(31, @res);
select @res;
```

The above procedure does not have great practical interest, and the goal is alone show that in a stored procedure you can use program logic in the same way as in for example Java.

As a final comment should be added that there is much more to say about stored procedures, and you may also have stored functions, but the above should suffice to illustrate what a stored procedure is.