Java Database Report

Ben Crabbe

University of Bristol, UK

March 16, 2015

1 Records

This covers the first step of the assignment, designing a Record class.

I decided that, to be true to the relational model, the order of the fields in each record should be unimportant, therefore I stored them in a Set but I choose the LinkedHashSet so that when a certain order seemed logical the fields could be defined in that order and would be printed etc in that order automatically.

I also thought that it seemed the record should know what its fields were and not just their attributes, so I created the Attribute class, and since all instances of this class must belong to a Record I made it an inner class.

I also created a custom exception to throw when any of the Record methods was used incorrectly, such as adding a field which already exists, or setting the value of a field which doesn't exist etc.

```
1
import java.util.*;
3
class Record
5 {
    public class RecordException extends Exception
7   {
        static final long serialVersionUID = 42L;
9       public RecordException()
        {
11          super();
        }
13
        public RecordException(String message)
15      {
            super(message);
17      }

19      public RecordException(String message, Throwable cause)
        {
21        super(message, cause);
        }
23
        public RecordException(Throwable cause)
25      {
            super(cause);
27      }
    }
29  class Attribute
    {
31      String name;
        String value;
33
        Attribute(String nameInput, String valueInput)
35      {
            name = nameInput;
37          value = valueInput;
        }
39  }
     Set<Attribute> data;
41
     Record() {
```

```
43          data = new LinkedHashSet<Attribute>();
44       }
45
46       void addField(String fieldName, String fieldValue) throws RecordException
47       {
48          for(Attribute field: data) {
49             if(field.name==fieldName) {
50                throw new RecordException("There already exists a field named " + fieldName + "in record.
         ");
51             }
52          }
53          Attribute newField = new Attribute(fieldName,fieldValue);
54          this.data.add(newField);
55       }
56
57       String getFieldValue(String fieldName) throws RecordException
58       {
59          for(Attribute field: data) {
60             if(field.name==fieldName) {
61                return field.value;
62             }
63          }
64          throw new RecordException("No field named " + fieldName + " in record.");
65       }
66
67       void setFieldValue(String fieldName, String fieldValue) throws RecordException
68       {
69          for(Attribute field: data) {
70             if(field.name==fieldName) {
71                field.value = fieldValue;
72                return;
73             }
74          }
75          throw new RecordException("No field named " + fieldName+ " in record.");
76       }
77
78       int countFields()
79       {
80          return data.size();
81       }
82
83       public static void main(String[] args)
84       {
85          Record row = new Record();
86          row.testRecord();
87       }
88
89       void testRecord()
90       {
91          try {
92             this.addField("Field1","value1");
93             this.addField("Field2","value2");
94             this.setFieldValue("Field2","value2.2");
95             System.out.println(this.getFieldValue("Field1"));
96             System.out.println(this.getFieldValue("Field2"));
97             System.out.println( this.countFields() );
98          } catch (Exception e) {
99             System.out.println(e.getMessage());
100         }
101      }
    }
```

2 Tables

I decided that the main data on the attribute names, types etc should be held by the table since every row is the same, so I split the Attribute class into two: Attribute and AttributeValue. Attributes hold the name (and other things in the future), AttributeValues, which are an inner class to Tuple (I renamed Record to fit better with the relational model) hold the values and a reference to the Attribute class it belongs to. This way the values are still directly searchable by attribute name, but the name is stored in only 1 place.

I also got rid of my custom exception. It wouldn't let me pass multiple string types to store as a message, and it didn't seem to gain much to be worth writing a bunch more constructors for.

```
import java.util.*;

class Table
{
    String name;
    List<Attribute> tableHeading;// attribute names, types, constraints etc
    Set<Tuple> tableBody;// a set of tuples

    Table(String name)
    {
        this.name = name;
        tableHeading = new ArrayList<Attribute>();//columns
        tableBody = new LinkedHashSet<Tuple>();//rows
    }

    Table(String name, List<String> fieldNames)
    {
        this.name = name;
        tableHeading = new ArrayList<Attribute>();
        tableBody = new LinkedHashSet<Tuple>();
        for(String newField: fieldNames) {
            try {
                addAttribute(newField);
            } catch(Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }

    boolean attributeExists(String attributeName)
    {
        for(Attribute a: tableHeading) {
            if(a.name==attributeName) {
                return true;
            }
        }
        return false;
    }

    void addAttribute(String attributeName) throws Exception
    {
        if(attributeExists(attributeName)){
            throw new Exception("attribute " + attributeName + " already exists");
        }
        Attribute newAtt = new Attribute(attributeName);
        tableHeading.add(newAtt);
    }

    int countFields()
    {
       return tableHeading.size();
    }

    int countRows()
```

```
55      {
            return tableBody.size();
57      }

59      List<String> getFieldNames()
        {
61          List<String> fieldNames = new ArrayList<String>();
            for(Attribute field: tableHeading) {
63              fieldNames.add(field.name);
            }
65          return fieldNames;
        }
67
        void editAttributeName(String oldName, String newName) throws Exception
69      {
            if(attributeExists(oldName)) {
71              Attribute editMyName = getAttributeFromName(oldName);
                editMyName.name = newName;
73          }
            else {
75              throw new Exception("Could not edit attribute name. No attribute named " +
                                    oldName + " in table.");
77          }
        }
79
        private Attribute getAttributeFromName(String name) throws Exception
81      {
            for(Attribute field: tableHeading) {
83              if(field.name==name) {
                    return field;
85              }
            }
87          throw new Exception("Could not get attribute. None named " + name + " in table.");
        }
89
        void addTuple(String... values) throws Exception
91      {
            if(values.length != tableHeading.size()) {
93              throw new Exception("addTuple() for Table " + name + " expects " +
                                    tableHeading.size() + " data values. It received " +
95                                  values.length);
            }
97          else {
                Tuple newTuple = new Tuple(tableHeading, Arrays.asList(values));
99              tableBody.add(newTuple);
            }
101     }

103     void deleteTuple(String attributeName, String value) throws Exception
        {
105         if(attributeExists(attributeName)) {
                Attribute attributeToDelete = getAttributeFromName(attributeName);
107             try {
                    for(Tuple row: tableBody) {
109                     if(row.getAttributeValue(attributeName)==value) {
                            tableBody.remove(row);
111                         return;
                        }
113                 }
                } catch(Exception e) {
115                 throw new Exception("deleteTuple() there are no tuples where " +
                                        attributeName + " has a value of " + value );
117             }
            } else {
119             throw new Exception("deleteTuple()  No attribute named " + attributeName +
                                    " in table " + name);
```

```
        }
    }

    static void is(Object x, Object y)
    {
        if (x == y) return;
        if (x != null && x.equals(y)) return;
        throw new Error("Error:\n" + x + "\n" + y);
    }

    void testTable()
    {

    }

    public static void main(String[] args)
    {
        Table t = new Table("testTable");
        t.testTable();
    }


}
```

```
import java.util.*;


class Tuple
{
   List<AttributeValue> attributes;

   class AttributeValue
   {
        String value;
        Attribute type;

        AttributeValue(Attribute attributeType, String value)
        {
           this.value = value;
           this.type = attributeType;
        }
    }


    Tuple()
    {
        attributes = new ArrayList<AttributeValue>();
    }

    Tuple( List<Attribute> tableHeading, List<String> values)
    {
        attributes = new ArrayList<AttributeValue>();
        for(int i=0; i<values.size(); ++i) {
            AttributeValue newAttribute = new AttributeValue(tableHeading.get(i), values.get(i));
        }
    }

    private AttributeValue getAttributeValueObjFromName(String attributeName) throws Exception
    {
        for(AttributeValue attribute: attributes) {
            if(attribute.type.name==attributeName) {
                return attribute;
            }
        }
```

```
        throw new Exception("No attribute named " + attributeName + " in tuple." );
    }

    boolean attributeExists(String name)
    {
        for(AttributeValue attribute: attributes) {
            if(attribute.type.name==name) {
                return true;
            }
        }
        return false;
    }

    String getAttributeValue(String attributeName) throws Exception
    {
        AttributeValue attribute =  getAttributeValueObjFromName(attributeName);
        return attribute.value;
    }


    void setAttributeValue(String attributeName, String attributeValue) throws Exception
    {
        AttributeValue attribute =  getAttributeValueObjFromName(attributeName);
        attribute.value=attributeValue;
    }

    int countAttributes()
    {
        return attributes.size();
    }

    void addAttribute(Attribute attributeType, String value) throws Exception
    {
        if(attributeExists(attributeType.name)) {
            throw new Exception("addAttribute() attribute named " + attributeType.name +
                                " already exists in tuple.");
        }
        AttributeValue newAttribute = new AttributeValue(attributeType,value);
        attributes.add(newAttribute);
    }
    static void is(Object x, Object y)
    {
        if (x==y || (x != null && x.equals(y)) ) {
            System.out.println("pass");
            return;
        }
        System.out.println("fail");
        throw new Error("Error:\n" + x + "\n" + y);
    }

    public static void main(String[] args)
    {
        Tuple row = new Tuple();
        row.testTuple();
    }

    void testTuple()
    {
        try {
            Attribute newAttribute = new Attribute("test");
            addAttribute(newAttribute,"value1");
            is(1,countAttributes());
            is("value1",getAttributeValue("test"));
            setAttributeValue("test", "value2");
            is("value2",getAttributeValue("test"));
        } catch (Exception e) {
```

```
            System.out.println(e.getMessage());
        }
    }
}
```

```
import java.util.*;

class Attribute
{
    String name;
    //      String type;

    Attribute(String name)
    {
        this.name = name;
        //    this.value = type;
    }
}
```

3 FIles

To handle table saving, loading and exporting I have made a FileHandler class.

3.1 Saving

To save files I added a toSaveString() method to Table class,

```
    String toSaveString()
    {
        String tableString = name + "||";
        try {
            List<String> attNames = getAttributeNames();
            for(String column: attNames) {
                tableString = tableString.concat(column);
                tableString = tableString.concat("|");
            }
            tableString = tableString.concat("|");
            for(Tuple row: tableBody) {
                for(String attName: attNames) {
                    tableString = tableString.concat(row.getAttributeValue(attName));
                    tableString = tableString.concat("|");
                }
                tableString =  tableString.concat("|");
            }
        } catch(Exception e) {
            System.out.println(e.getMessage());
            throw new Error();
        }
        return tableString;
    }
```

I used a single pipe("|") to delimit columns, and double for rows these were defined in the FileHandler class:

```

    final static private String encoding = "UTF-8";
    final static String newLineDelim = Pattern.quote("||");
    final static String columnDelim = Pattern.quote("|");
```

To stop field names containing pipes I added a method isValidName() which all attribute values and names, and table names were passed through before being set.

```
static boolean isValidName(String name) throws Exception
{
    if( name.contains(FileHandler.columnDelim) ||
       name.contains(FileHandler.newLineDelim) ) {
        throw new Exception(name + " is invalid. names cannot contain any of: " +
                            FileHandler.columnDelim + FileHandler.newLineDelim);
    }
    else return true;
}
```

The FileHandlerClass is then used to save the table. saveTableToFile() provides a descriptive interface, whilst stringToFile() to which it delegates the actual io , is generic and reusable.

```

void saveTableToFile(Table t, String fpath)
{
    stringToFile( t.toSaveString(), fpath);
}

private void stringToFile(String s, String fpath)
{
    Writer writer = null;
    try {
        writer = new BufferedWriter(
                    new OutputStreamWriter(
                        new FileOutputStream(fpath), encoding));
        writer.write(s);
    } catch (IOException e) {
        System.out.println(e.getMessage());
    } finally {
        try {
            writer.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

3.2 Loading

Loading from the produced save strings is handled similarly. A generic method readFile() gets all the text form a file:

```
static String readFile(String fpath) throws IOException
{
    byte[] encoded = Files.readAllBytes(Paths.get(fpath));
    return new String(encoded, encoding);
}
```

and a function which splits the saved string up and builds the table which it returns. I think that theres perhaps too much detailed table handling for this to be in the FileHandler class, it could be better to just pass the file string to a constructor and let the Table class deal with it, but there is already a constructor that accepts a single String and sets just the table name. So I am going to leave it like this until I have worked out how users will create tables.

```
Table loadTableFromFile(String fpath)
{
    try {
```

```
            String tableString = new String(readFile(fpath));
            String[] tableRowStrings = tableString.split(newLineDelim);
            String[] tableHeadings = tableRowStrings[1].split(columnDelim);
            Table newTable = new Table(tableRowStrings[0],//[0] has table name
                                       Arrays.asList(tableHeadings));
            for(int i=2; i<tableRowStrings.length; ++i) {
                newTable.addTuple(tableRowStrings[i].split(columnDelim));
            }
            return newTable;
        } catch (Exception e) {
            System.out.println(e.getMessage());
            throw new Error();
        }
    }
```

3.3 Printing/Exporting

FileHandler provides the interface:

```
    void exportTableAsTxt(Table t, String fpath)
    {
        List<String> rowStrings = t.presentTableForPrinting();
        stringListToFile(rowStrings, fpath);
    }
```

which just uses the stringToFileMethod I already wrote:

```
    private void stringListToFile(List<String> strList, String fpath)
    {
        String combined = "";
        for(String str: strList) {
            combined = combined.concat(str);
        }
        stringToFile(combined,fpath);
    }
```

the table handles the formatting:

```
    private int spacesBetweenCols = 4;

    List<String> presentTableForPrinting()
    {
        List<String> presentedTable = new ArrayList<String>();
        List<String> headings = getAttributeNames();
        List<Integer> headingWidths = getColumnWidths(headings);

        presentedTable.add(name);
        presentedTable.add("\n\n\n");

        presentedTable.add(presentTableRow(headings, headingWidths));
        presentedTable.add( makeStringOfChar( getTableWidth(headingWidths), '-') + "\n" );

        for(Tuple row: tableBody) {
            List<String> rowString = row.toListOfStrings();
            presentedTable.add(presentTableRow(rowString, headingWidths));
        }
        return presentedTable;
    }

    private int getTableWidth(List<Integer> headingWidths)
```

```
23      {
            int sum = 0;
25          for(int headingWidth : headingWidths) {
                sum += headingWidth;
27          }
            sum +=spacesBetweenCols*countColumns();
29          return sum;
        }
31
        private String presentTableRow(List<String> rowStrings, List<Integer> headingWidths)
33      {
            String presentedRow = "";
35          for(int i=0; i<rowStrings.size(); ++i)
            {
37              int strWidth = getWidthOfString(rowStrings.get(i));
                int spacesNeeded = headingWidths.get(i) - strWidth;
39              presentedRow = presentedRow.concat(rowStrings.get(i));
                presentedRow = presentedRow.concat(makeStringOfChar(spacesNeeded+spacesBetweenCols, ' '))
    ;
41          }
            presentedRow += '\n';
43          return presentedRow;
        }
45
        private String makeStringOfChar(int number, char c)
47      {
            String s = "";
49          for(int i=0; i<number; ++i)
            {
51              s = s + c;
            }
53          return s;
        }
55
        private List<Integer> getColumnWidths(List<String> headings)
57      {
            List<Integer> headingWidths = new ArrayList<Integer>();
59          for(String heading: headings) {
                headingWidths.add(getWidthOfString(heading));
61          }
            return headingWidths;
63      }

65      private int getColumnWidth(String attributeName) throws Exception
        {
67          Attribute a = getAttributeFromName(attributeName);
            int maxwidth = getWidthOfString(attributeName);
69          for(Tuple row: tableBody) {
                String valueString = row.getAttributeValue(attributeName);
71              int width = getWidthOfString(valueString);
                maxwidth = width > maxwidth ? width : maxwidth;
73          }
            return maxwidth;
75      }

77      static int getWidthOfString(String str)
        {
79          int width=0, maxwidth=0;
            for( char c: str.toCharArray() ) {
81              if(c=='\n' || c=='\r') {
                    maxwidth = width > maxwidth ? width : maxwidth;
83                  width = 0;
                }
85              else {
                    ++width;
87              }
```

```
        }
89          maxwidth = width > maxwidth ? width : maxwidth;
            return maxwidth;
91      }
```

an example of the output is:

```
testTable2

Attribute1      Att2     A3
-------------------------------
value1          val1     v1
value2          val2     v2
```

Here is the full module:

```


import static java.nio.file.StandardOpenOption.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
import java.io.*;
import java.util.*;
import java.lang.Object;
import java.util.regex.Pattern;

class FileHandler
{
    final static private String encoding = "UTF-8";
    final static String newLineDelim = Pattern.quote("||");
    final static String columnDelim = Pattern.quote("|");

    FileHandler()
    {

    }

    void exportTableAsTxt(Table t, String fpath)
    {
        List<String> rowStrings = t.presentTableForPrinting();
        stringListToFile(rowStrings, fpath);
    }

    private void stringListToFile(List<String> strList, String fpath)
    {
        String combined = "";
        for(String str: strList) {
            combined = combined.concat(str);
        }
        stringToFile(combined,fpath);
    }

    void saveTableToFile(Table t, String fpath)
    {
        stringToFile( t.toSaveString(), fpath);
    }

    private void stringToFile(String s, String fpath)
    {
        Writer writer = null;
```

```
        try {
48          writer = new BufferedWriter(
                        new OutputStreamWriter(
50                          new FileOutputStream(fpath), encoding));
            writer.write(s);
52      } catch (IOException e) {
            System.out.println(e.getMessage());
54      } finally {
            try {
56              writer.close();
            } catch (Exception e) {
58              System.out.println(e.getMessage());
            }
60      }
    }
62
    static private List<String> trimWhiteSpace(String[] oldStrings)
64  {
        List<String> newStrings = new ArrayList<String>();
66      for(String s: oldStrings) {
            newStrings.add(s.trim());
68      }
        return newStrings;
70  }

72  Table loadTableFromFile(String fpath)
    {
74      try {
            String tableString = new String(readFile(fpath));
76          String[] tableRowStrings = tableString.split(newLineDelim);
            String[] tableHeadings = tableRowStrings[1].split(columnDelim);
78          Table newTable = new Table(tableRowStrings[0],//[0] has table name
                                       Arrays.asList(tableHeadings));
80          for(int i=2; i<tableRowStrings.length; ++i) {
                newTable.addTuple(tableRowStrings[i].split(columnDelim));
82          }
            return newTable;
84      } catch (Exception e) {
            System.out.println(e.getMessage());
86          throw new Error();
        }
88  }

90  static String readFile(String fpath) throws IOException
    {
92      byte[] encoded = Files.readAllBytes(Paths.get(fpath));
        return new String(encoded, encoding);
94  }

96  void testReadFile()
    {
98      testSaveTable();
        try {
100         String file = readFile("./saves/test.txt");
            System.out.println(file);
102     } catch (Exception e) {
            System.out.println(e.getMessage());
104         throw new Error();
        }
106 }

108 void testSaveTable()
    {
110     List<String> attributeNames2 = new ArrayList<String>(3);
        attributeNames2.add("Attribute1");
112     attributeNames2.add("Attribute2");
```

```
        attributeNames2.add("Attribute3");
        try {
            Table t2 = new Table("testTable2",attributeNames2);
            t2.addTuple("value1","value1","value1");
            t2.addTuple("value2","value2","value2");
            String tableString = t2.toSaveString();
            saveTableToFile(t2,"./saves/test.txt");
        } catch(Exception e) {
            System.out.println(e.getMessage());
            throw new Error();
        }
    }

    void testLoadTableFromFile()
    {
        try {
            testSaveTable();//make sure we have a file there
            Table testT = loadTableFromFile("./saves/test.txt");
            String stringInFile = readFile("./saves/test.txt");
            is(testT.toSaveString(),stringInFile);
        } catch(Exception e) {
            System.out.println(e.getMessage());
            throw new Error();
        }
    }

    void testExportTableAsTxt()
    {
        List<String> attributeNames2 = new ArrayList<String>(3);
        attributeNames2.add("Attribute1");
        attributeNames2.add("Att2");
        attributeNames2.add("A3");
        try {
            Table t2 = new Table("testTable2",attributeNames2);
            t2.addTuple("value1","val1","v1");
            t2.addTuple("value2","val2","v2");
            exportTableAsTxt(t2, "./exports/test.txt");
        } catch(Exception e) {
            System.out.println(e.getMessage());
            throw new Error();
        }
    }

    static void is(Object x, Object y)
    {
        System.out.print("testing " + x.toString() + " = " + y.toString() );

        if (x==y || (x != null && x.equals(y)) ) {
            System.out.println("...pass");
            return;
        }
        System.out.print("...fail");
    }

    public static void main(String[] args)
    {
        FileHandler fh = new FileHandler();
        fh.testSaveTable();
        fh.testReadFile();
        fh.testLoadTableFromFile();
        fh.testExportTableAsTxt();

    }
}
```

4 Keys

Since I had tried to follow the relational model as closely as possible from the start attributes were already addressed by their values rather than row/col numbers.

To add a defined PrimaryKey I added a member to the Table class : List≺ Attribute≻primaryKey;

As default the table constructor sets primaryKey to all the attributes, which is ensured to be unique by a method doesTupleExist i have added, and called within each addTuple call.

```
    private boolean doesTupleExist(String... values)
    {
        List<String> newTupleValues = Arrays.asList(values);
        for(Tuple row: tableBody) {
            List<String> rowValues = row.getAllValuesOfTuple();
            if(rowValues.size() == newTupleValues.size() && rowValues.containsAll(newTupleValues)) {
                return true;
            }
        }
        return false;
    }

    void addTuple(String... values) throws Exception
    {
        if(doesTupleExist(values)) {
            throw new Exception("Duplicate tuple discarded");
        }
        if(values.length != tableHeading.size()) {
            throw new Exception("addTuple() for Table " + name + " expects " +
                                tableHeading.size() + " data values. It received " +
                                values.length);
        }
        else {
            Tuple newTuple = new Tuple(tableHeading, Arrays.asList(values));
            tableBody.add(newTuple);
        }
    }
```

which uses this in the Tuple class:

```
  List<String> getAllValuesOfTuple()
    {
        List<String> allValuesOfTuple = new ArrayList<String>();
        for(AttributeValue av: attributes) {
            allValuesOfTuple.add(av.value);
        }
        return allValuesOfTuple;
    }
```

The primary key can then be set to a specific attribute at any point with.

```

    void setPrimaryKey(String... attributeNames) throws Exception
    {
        if( countColumns()==0 ) {
            throw new Exception("There are no attributes.");
        }
        if( countRows()>0 )  {
            if( !isAttributeSetUnique(attributeNames) ) {
                throw new Exception("Suggested PK is not unique. Cannot set.");
            }
        }
        primaryKey.clear();
        for(String attributeName: attributeNames) {
```

```
            primaryKey.add(getAttributeFromName(attributeName));
        }
    }


    private boolean isAttributeSetUnique(String... attributeNames) throws Exception
    {
        List<String> allValues = getAllValuesOfAttribute(attributeNames);
        Set<String> allValuesSet = new HashSet<String>(allValues);
        if( allValuesSet.size() < allValues.size() ) return false;
        else return true;
    }
```

I have added an extra bit to the save/load methods to store/extract the tables primary key.

I have replaced the Table constructor that accepted just a table name to one that works better with the loadTableFromFile method. That method now deals with all the file decoding, and sends all the info to the new constructor:

```
    static Table loadTableFromFile(String fpath)
    {
        try {
            String tableString = new String(readFile(fpath));
            String[] tableRowStrings = tableString.split(newLineDelim);
            String[] tableHeadings = tableRowStrings[2].split(columnDelim);
            String[] primaryKey = tableRowStrings[1].split(columnDelim);
            String tableName = tableRowStrings[0];
            Table newTable = new Table(tableName, primaryKey, tableHeadings,
                                       tableRowStrings);
            return newTable;
        } catch (Exception e) {
            System.out.println(e.getMessage());
            throw new Error();
        }
    }


  Table(String tableName, String[] primaryKey, String[] tableHeadings, String[] tableRows) throws
    Exception
    {
        this(tableName, tableHeadings);
        try {
            for(int i=3; i<tableRows.length; ++i) {
                addTuple(tableRows[i].split(FileHandler.columnDelim));
            }
            setPrimaryKey(primaryKey);
        } catch (Exception e) {
            System.out.println(e.getMessage());
            throw new Exception("Could not load table");
        }
    }
```

5 Database

I decided to store tables in a set, since order should not be important. Names are required to be unique.

```
import java.util.*;


class Database
{
    Set<Table> tables;
```

```
7
8       Database()
9       {
10          tables = new LinkedHashSet<Table>();
11      }
12
13      Database(List<Table> tables) throws Exception
14      {
15          this();
16          for(Table t: tables) {
17              addTable(t);
18          }
19      }
20
21      List<String> showTables()
22      {
23          List<String> tableNames = new ArrayList<String>();
24          for(Table t: tables) {
25              tableNames.add(t.name);
26          }
27          return tableNames;
28      }
29
30      boolean isNameClash(String name)
31      {
32          for(Table t: tables) {
33              if(t.name.equals(name)) {
34                  return true;
35              }
36          }
37          return false;
38      }
39
40      void addTable(Table t) throws Exception
41      {
42          if(isNameClash(t.name)) {
43              throw new Exception("Table called " + t.name + " already exists in Database");
44          } else {
45              tables.add(t);
46          }
47      }
48
49      String toSaveString()
50      {
51          String saveString = new String();
52          for(Table t: tables) {
53              saveString += t.toSaveString();
54              saveString += "|";
55          }
56          return saveString;
57      }
58
59      void testLoadDB()
60      {
61          Driver.is(showTables().contains("testTable2"),true);
62          Driver.is(showTables().contains("testTable3"),true);
63          Driver.is(showTables().contains("testTable"),true);
64      }
65
66      void testLoadDbWithNameClash()
67      {
68          Driver.is(tables.size(),2);
69          Driver.is(showTables().contains("testTable2"),true);
70          Driver.is(showTables().contains("testTable"),true);
71      }
```

```
    public static void main(String[] args)
    {
        Database db = FileHandler.loadDataBaseFromFile("./saves/testDB.txt");
        db.testLoadDB();
        Database db2 = FileHandler.loadDataBaseFromFile("./saves/testDBaddTable.txt");
        db2.testLoadDbWithNameClash();

    }
}
```

I added loading and saving methods to FileHandler to deal with databases. These use a 3rd pipe to delimit tables.

```
    static Database loadDataBaseFromFile(String fpath)
    {
        try {
            String dbString = new String(readFile(fpath));
            String[] tableStrings = dbString.split(newTableDelim);
            List<Table> tables =  new ArrayList<Table>();
            for(String tableString: tableStrings) {
                tables.add(loadTableFromString(tableString));
            }
            Database db = new Database(tables);
            return db;
        } catch (Exception e) {
            System.out.println(e.getMessage());
            throw new Error();
        }
    }
```