Summary
We discuss a programming problem. We look at some attempted student solutions, and discuss their problems.
We present several approaches for solving the problem.

Some Code
In a course I taught I gave the following assignment (this is just part of the assignment):
One can learn much about DBMS data by examining its values without the benefit of the schema. In fact, by analyzing the data one can often infer some or all of the schema. …
For this assignment design and implement algorithms to analyze data. …
Data will be provided in csv files, one file per table. The tables may be quite large.
Implement a set of analysis functions, …. …
1. Analyze a set of columns in one table 
   a. Given a set of columns, could the set of columns be a primary key? 
2. Analyze 2 tables 
   a. Given a column from each table, is a column in one table a possible foreign key for a column in the other table?
Let’s look at two attempts to solve problem 1 written by former students of mine.
Could the Set of Columns Be a Primary Key?

Consider this code. (Ignore the misspellings in the comments, but, please, spell-check your comments. Well written comments increase the quality of your code. Comments should pass the spell checker.)

What do you notice about the method isPrimaryKey?

Does it do the right thing? No. The assignment asks for code that ‘could the set of columns be a primary key’. This code only examines one column, column h.

How should we define a ‘set of columns that could be a primary key’? First, the values in the set of columns must have no duplicate rows in the table. Second, the values in the set of columns must have no NULLs in the table.

Does the method isPrimaryKey correctly determine whether column h is a primary key? Partly. It does return false when a pair of values in the column are equal. But it doesn’t satisfy the second part of the definition, since it does not check for NULLs.

Is it described by clear comments? No. It should have a javadoc comment. The comment should state the meaning of the return value.

Does it make good use of data structures and algorithms available in the language or its libraries? No, none are used.

Does it work efficiently? No. Its complexity is \( n^2 \), where \( n \) is the size of the table. There are several faster algorithms, including one that uses sort and costs \( n \log(n) \) and one that uses a hash table and costs \( n \).

Is there evidence that it has been tested? No.

Here’s the code:

---

1 Just scan the column, storing it in the hash, and if a duplicate shows then return false, i.e., not a primary key. Here’s an implementation in an o-o style

```java
/**
 * check whether a column could be a primary key.
 * technique: use a hash; Just scan the column, storing it in the hash,
 * and if a duplicate shows then return false, i.e., not a primary key.
 * complexity: O( number of elements in the column)
 * @param the_column  the column to check
 * @return  true if column could be a primary key
 */

boolean is_primary_key( column the_column)
    hash the_hash = new hash()
    for_each entry in the_column
        if ( the_hash.contains(entry) )
            return false
        the_hash.put(entry)
    end_for
    return true
```

To check multiple columns simply concatenate the columns together and treat the set of multiple columns as a single column in the algorithm above. One precaution: insert a separator—that is not in the data—between the columns. Otherwise these 2 rows which are actually different would be the same:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>
import java.util.*;

/*
 This class emulates a table in a Database. It contains a set of columns
 that collectively make up the dataset and provides handles on those columns.
 The methods set, as well as provide access to values in the data. Most of the
 public methods are practically self explanatory by name.
 */

public class Table {

// Some methods left out o o o

/*To check if column h is a primary key*/
public boolean isPrimaryKey(int h) {
    boolean doesMatch = true;
    Coloumn c = (Coloumn) this.coloumns.get(new Integer(h));
    for (int i = 0; i < c.entries.size(); i++) {
        String s1 = (String) c.entries.get(i);
        for (int j = i + 1; j < c.entries.size(); j++) {
            String s2 = (String) c.entries.get(j);
            /*If an entry in Coloumn C repeats then check if the rows
            corresponding to both entries are equal*/
            if (s1.compareTo(s2) == 0) {
                String row1 = this.getRowAsString(i);
                String row2 = this.getRowAsString(j);
                if (row1.compareTo(row2) != 0) {
                    doesMatch = false;
                    return doesMatch;
                }
            }
        }
    }
    return doesMatch;
}

public Table() {
    this.coloumns = new Hashtable();
}

public Coloumn getColoumn(int i) {
    return (Coloumn)this.coloumns.get(new Integer(i));
}

/**Testing*/
public static void main(String args[]) {
    String s1 = "abcd";
    String s2 = "getfs";
    s2 = s2.concat(s1);
    System.out.println(s2);
}

/* Class designed to emulate a column in a database. Has utility methods to perform...*/
public class Column {

    public void addEntry(String s, int i) {
        this.entries.add(s);
    }

    public String getEntry(int k) {
        return (String)this.entries.get(k);
    }

    public Column () {
        this.entries = new ArrayList();
    }

    public Columnn () {
        this.entries = new ArrayList();
    }

    //Params

    public ArrayList entries;

}
Here’s another implementation. Let’s answer the same questions about it. TODO

```java
/**
 * @param c column
 * @return true if column is a primary key
 */
public boolean isPrimaryKey(Column c)
{
    java.util.HashSet checkedAttrs = new java.util.HashSet();
    // String[] values = c.getValues();
    String[] otherAtt = new String[columns.size() - 1];
    int size = getNumberOfRows();
    String primeKey = null;
    String primeKeyVal = null;
    String nextKey = null;
    String nextKeyVal = null;

    initOtherAttr(otherAtt, c.getName());

    // j is going down on the rows
    for (int j = 0; j < size - 1; j++)
    {
        primeKey = (String) c.getValue(j);
        // collecting values accross
        primeKeyVal = catValuesAt(otherAtt, j);
        if (!checkedAttrs.contains(primeKey))
        {
            checkedAttrs.add(primeKey);
            // check attribute "primeKey" against all attributes in this column
            // where attribute.name = "primeKey" and values of other columns at
            // forms a distinct result
            for (int i = j + 1; i < size; i++)
            {
                nextKey = (String) c.getValue(i);
                // if next Key is the same as prime
                // check if attributes of other columns at nextKey's index
                // are equal to attributes at primeKey index
                if (primeKey.equals(nextKey))
                {
                    // collecting values accross
                    nextKeyVal = catValuesAt(otherAtt, i);
                    if (!primeKeyVal.equals(nextKeyVal))
                    {
                        return false;
                    }
                }
            }
        } // end if attributes was not checked yet.
    }

    return true;
}
/**
 * @return private util function. takes values in columns specified in pKeys
 * column name list at index and returns concat version of these values.
 * this function is used in order to determine primeKey
 */
private String catValuesAt(String[] pKeys, int index)
{
    if (pKeys == null)
    {
        return "";
    }
```
sb.setLength(0);
Column c = null;

for (int i = 0; i < pKeys.length; i++)
{
    c = getColumn(pKeys[i]);
    sb.append((String) c.getValue(index));
}

return sb.toString();
A Top-down Design, Pseudocode Approach

One approach (there are others) to solving this problem is the following:

1. Understand the specification
2. Write interfaces for the primary functions (methods)
3. Identify the library functions that would be useful for the implementation
4. Write high-level pseudocode describing the implementation
5. Go back to 3 if the design seems inadequate
6. Write test cases and their outputs
7. Expand the pseudocode into code
8. Perform the unit test, using existing and new test cases

Let’s try this.

1. Understand the specification

The key phrase in the specification is “Given a set of columns (in one table), could the set of columns be a primary key?” The relational database definition of primary key says: one or more columns (attributes) in a table may be declared the table’s ‘primary key’. First, the set of columns cannot have duplicate rows in the table. That is, there cannot be two rows with the same values for all the columns in the primary key. Second, the values in the set of columns must have no NULLs in the table.

2. Write interfaces for the primary functions (methods)

We might as well write the javadoc at the same time.

```java
/**
 * check whether a set of columns could be a primary key in a table,
 * based on the values in the columns.
 *
 * @param theColumns  a list of columns
 * @return  true if columns could be a primary key
 * @exception  illegalColumnList  if theColumns contains illegal values
 * or duplicate entries (todo check this javadoc field)
 */
boolean couldBePrimaryKey( whatsThisDatatype theColumns )
```

We assume this is a method in a table class which stores tables and collects operations on them. We’ll need to make some decisions about how tables (and columns) are stored. Then we’ll be able to decide on the datatype of theColumns, which needs to be a set of column identifiers.

3. Identify the library functions that would be useful for the implementation

Determining whether the values have NULLs depends simply on a linear search of the data. There’s no shortcut for this, so no library functions would be helpful.

Finding duplicate entries in the columns involves checking whether any row appears more than once in the table. One way to do this would be to load all the rows of the columns being examined into some data structure and sort them and then scan them and look for matching adjacent entries. This costs at least $n \log(n)$ where $n$ is the number of rows because the sort must finish. It costs at most $n \log(n)+n$.

But note that we don’t really need the rows in order, we just need to look for matches. A faster way to find duplicate entries would be to scan all the rows, looking for them in a
hash table and entering them into the hash table. If a row appears more than once, it will
be found in the hash table. This costs at most $n$, assuming the hash table get and put cost
$O(1)$.

So let’s look for a hash table in the JDK. The API documentation for HashMap says:

```java
public class HashMap
extends AbstractMap
implements Map, Cloneable, Serializable
```

Hash table based implementation of the Map interface. This implementation
provides all of the optional map operations, and permits null values and the
null key. (The HashMap class is roughly equivalent to Hashtable, except that it
is unsynchronized and permits nulls.) This class makes no guarantees as
to the order of the map; in particular, it does not guarantee that the order
will remain constant over time.

This implementation provides constant-time performance for the basic
operations (get and put), assuming the hash function disperses the
elements properly among the buckets. Iteration over collection views
requires time proportional to the "capacity" of the HashMap instance (the
number of buckets) plus its size (the number of key-value mappings).
Thus, it’s very important not to set the initial capacity too high (or the load
factor too low) if iteration performance is important.

The HashMap looks suitable; it uses a hash table, it provides constant time performance
for the basic operations, and it isn’t synchronized, which we don’t need.
The put method says:

```java
public Object put(Object key,
                   Object value)
```

Associates the specified value with the specified key in this map. If the
map previously contained a mapping for this key, the old value is
replaced.

**Specified by:**
put in interface Map

**Overrides:**
put in class AbstractMap

**Parameters:**
key - key with which the specified value is to be associated.
value - value to be associated with the specified key.

**Returns:**
previous value associated with specified key, or null if there was no
mapping for key. A null return can also indicate that the HashMap
previously associated null with the specified key.

This is OK, but more functionality than we need. To solve the duplicate row problem we
just need to put in a row and then determine whether later rows match it. That is we can
use the key parameter, but don’t need the value. So let’s look at another class, HashSet:
public class HashSet
extends AbstractSet
implements Set, Cloneable, Serializable
This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.
This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

**Note that this implementation is not synchronized.** If multiple threads access a set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the HashSet instance:

```java
Set s = Collections.synchronizedSet(new HashSet(...));
```

The iterators returned by this class's iterator method are fail-fast: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the Iterator throws a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future. Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

**Method Summary**

boolean add(Object o)
- Adds the specified element to this set if it is not already present.

void clear()
- Removes all of the elements from this set.

Object clone()
- Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.

boolean contains(Object o)
- Returns true if this set contains the specified element.
boolean **isEmpty**()

Returns true if this set contains no elements.

**Iterator iterator()**

Returns an iterator over the elements in this set.

boolean **remove**(Object o)

Removes the specified element from this set if it is present.

int **size**()

Returns the number of elements in this set (its cardinality).

HashSet looks perfect. We can use contains to check for rows, and add to add them. It offers “offers constant time performance for the basic operations” which we want. We don’t need synchronization and we’ll set the capacity low, at least smaller than the number of rows in the table.

4. **Write high-level pseudocode describing the implementation**

First, we need to think about data structures. How should the table be represented? Let’s pick something simple that will make it easy to access columns. (Note that this may not be good for some database operations, but considering all database operations is not part of the assignment.) One of the simplest thing I can think of is an array for each column and then an array of columns for each table. (I did consider the dual, an array for each row and an array of rows for each table, but then it would be more difficult to manipulate columns.) Perhaps the column will need some other parameters, like name and number. We can simplify by assuming that all the data is of type String.

To summarize:

A table contains an array of columns.

A column stores its data in an array of entries which are Strings; it may also contain some identification information such as a name or index.

How will we identify columns? The assignment does not mention column names, so I think we should simply identify them by index number. Let’s just count the columns from 1 to n as they appear in the CSV file.

Now the pseudocode should practically write itself:

```java
boolean couldBePrimaryKey( whatsThisDatatype theColumns )
    If there’s a problem with theColumns then
        throw the illegalColumnList exception
    Create a HashSet
    For each row in the table
        Assemble the columns into a single object
        If the object is in the HashSet then
            return false
        Insert the object into the HashSet
    End for
    Return true
```

It seems obvious that theColumns should be represented as a list of integer column indices. An array of integers should work fine.

The first “if” statement will be straightforward to implement. A few other issues need to be resolved.

The table object needs methods for the following: a constructor, return a column, read the CSV file into the table. The first two are easy, but I don’t know how to read a CSV file because I don’t know how the CSV standard escapes a comma in a field. Also, I don’t know whether the first line is data or description. With a little research
I find that there is no CSV standard, but the format used by MSFT has the following features:

1. **Fields are separated by commas.**
2. **Leading and trailing space-characters adjacent to comma field separators are ignored.**
   
   So *John , Doe ,...* resolves to "John" and "Doe", etc. Space characters can be spaces, or tabs.
3. **Fields with embedded commas must be delimited with double-quote characters.**
4. **Fields that contain double quote characters must be surrounded by double-quotes, and the embedded double-quotes must each be represented by a pair of consecutive double quotes.**
   
   So, *John "Da Man" Doe* would convert to "John ""Da Man""Doe ", and a single quote that is not a delimiter is an error
5. **A field that contains embedded line-breaks must be surrounded by double-quotes**
6. **The first record in a CSV file may be a header record containing column (field) names**

First I check whether there is a CSV class in the JDK, but I cannot find one. The CSV standard above involves more complexity than we want to handle in this assignment, so I decide to implement just features 1, and 2 and ignore the others. (If this were a real programming task or assignment I’d check with the customer or professor that this decision would be OK.) I’ll document this in the code. We’ll solve the CSV parsing problem completely later. Here’s pseudocode for reading a CSV file. I insert the lines from the specification into the pseudocode to keep them together.

```java
Read CSV file
   Open the file
   If the first line in the file cannot be read then
      Throw exception
   Read the first line
   // **Fields are separated by commas.**
   Separate the line into tokens separated by commas
   Create a column for each token
   Process the line
   For each line in the file
      // **Fields are separated by commas.**
      Separate the line into tokens separated by commas
      Process the line
   End for

Process the line
   For each token (up to the last column)
      // **Leading and trailing space-characters adjacent to comma field separators are ignored.**
      Strip whitespace at the edges
      Insert the token in the column
```
End for

We still need to decide how to represent the columns as a single object.

5. Go back to 3 if the design seems inadequate
The design seems adequate.

6. Write test cases and their outputs
We want to test the common cases and the extreme cases. We want to test all lines of the code, which is called coverage testing.

Woops, I just realized that the pseudocode for couldBePrimaryKey doesn’t test for NULL values. Here’s a fixed version:

```java
boolean couldBePrimaryKey( whatsThisDatatype theColumns )
    If there’s a problem with theColumns then
        throw the illegalColumnList exception
    Create a HashSet
    For each row in the table
        If any of the columns contains a NULL value then
            Return false
        Assemble the columns into a single object
        If the object is in the HashSet then
            return false
        Insert the object into the HashSet
    End for
    Return true
```

Here are some good test cases:

- Empty table, which I think should return false since there’s no data to base the decision on. (Woops, I just realized that the pseudocode doesn’t check for this. I’ll fix it later.)
- Bad column lists in theColumns: No columns indicated, negative column value, huge column value, non-positive column value, duplicate column values (I’m unsure whether this should be illegal; certainly a column only needs to be examined once, but duplicating it causes no real problems).
- Big table.
- Tables with duplicate rows: first and last row, all rows the same, last 2 rows, first 2 rows.
- Table with NULL values.
- A table like this, in which a lack of a separator could cause mistaken duplicates:

```
1 23
12 3
```

We organize the test cases in this table. They clearly execute all lines in the pseudocode for couldBePrimaryKey.

<table>
<thead>
<tr>
<th>Output</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Empty table</td>
</tr>
<tr>
<td>F</td>
<td>Bad column lists in theColumns: No columns indicated, negative column value, huge column value, non-positive column value</td>
</tr>
<tr>
<td>T</td>
<td>Big table with no duplicate rows</td>
</tr>
<tr>
<td>F</td>
<td>Big table with duplicate rows</td>
</tr>
</tbody>
</table>
F  Tables with duplicate rows: first and last row, all rows the same, last 2 rows, first 2 rows
T  Table with 1 row, no NULLs
F  Table with NULL values
T  A table like this, in which a lack of a separator could cause mistaken duplicates:

<table>
<thead>
<tr>
<th>1</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>3</td>
</tr>
</tbody>
</table>

Test cases for CSV parsing are left as an exercise.

7. *Expand the pseudocode into code*

We need two classes, column and table, plus a test class. Let’s sketch them out.

<table>
<thead>
<tr>
<th>Class</th>
<th>Instance variable</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>Int: Index number</td>
<td>Identifier</td>
</tr>
<tr>
<td></td>
<td>Arraylist of string: data</td>
<td></td>
</tr>
<tr>
<td>Table</td>
<td>String: filename</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Array of column : the data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int: number columns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Int: number of rows</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column</td>
<td>New</td>
<td>Make new column</td>
</tr>
<tr>
<td></td>
<td>Put</td>
<td>Put a field</td>
</tr>
<tr>
<td></td>
<td>Get</td>
<td>Get field value</td>
</tr>
<tr>
<td>Table</td>
<td>New</td>
<td>Make new table</td>
</tr>
<tr>
<td></td>
<td>insertColumn</td>
<td>Add a column</td>
</tr>
<tr>
<td></td>
<td>Public couldBePrimaryKey</td>
<td>Determine whether a set of columns could be a primary key</td>
</tr>
<tr>
<td></td>
<td>Public readCSVFile</td>
<td>Read a CSV file into a table (^2)</td>
</tr>
<tr>
<td></td>
<td>Private processCSVLine</td>
<td>Process a line in a CSV file being read</td>
</tr>
<tr>
<td>Test</td>
<td>Main</td>
<td>Run a bunch of tests</td>
</tr>
<tr>
<td></td>
<td>testOneFile</td>
<td>Test one CSV file</td>
</tr>
</tbody>
</table>

Are there any remaining issues?

<table>
<thead>
<tr>
<th>Issue</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>In couldBePrimaryKey how do we Assemble the columns into a single object?</td>
<td>Since the fields do not contain commas we can simply use commas; we must document this decision, since it will fail when the code is changed to use full featured CSV files in which fields can contain commas</td>
</tr>
<tr>
<td>In Test how do we “Run a”</td>
<td>Ignore until later</td>
</tr>
</tbody>
</table>

\(^2\) I just realized that the pseudocode for readCSVFile needs the line “Create new table” before the line “Read the first line”.

13
bunch of tests”?

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>How do we write the big test cases?</td>
<td>Ignore until later</td>
</tr>
<tr>
<td>How do we test performance?</td>
<td>Ignore until later</td>
</tr>
</tbody>
</table>

With no major issues remaining, I think, we’re ready to implement column, table and test. This is my initial implementation of column and table, after compilation and personal code review, but before any testing. *I expect to find bugs in this!* 

```java
import java.util.*;

/**
 * A Column represents a column of data in a table.
 * It can store any kind of Object.
 * @author Arthur Goldberg, NYU
 * 9/9/3
 */
public class Column {
    int indexNumber;
    // the index number this Column, in the array of columns in the table
    ArrayList theData;
    // an ArrayList of Strings, containing the data
    Column(int theIndexNumber) {
        indexNumber = theIndexNumber;
        theData = new ArrayList();
    }
    /**
     * append a field to the column.
     * @param field the object being stored in the field
     */
    void append(Object field) {
        theData.add(field);
    }
    /**
     * Store a field in the column.
     * @param index the index of the field in the Column.
     * @param field the object being stored in the field
     * @throws IndexOutOfBoundsException if the index is not in the Column's range
     */
    void put(int index, Object field) throws IndexOutOfBoundsException {
        theData.set(index, field);
    }
    /**
     * Get field value
     * @param index the index of the field in the Column.
     * @throws IndexOutOfBoundsException if the index is not in the Column's range
     * @return the Object at index.
     */
    Object get(int index) throws IndexOutOfBoundsException {
        return theData.get(index);
    }
}
```

Note that I decided to store Objects in columns, not Strings as I’d considered before, because Objects are more general.

In the Table class, the columns are stores in an `ArrayList` instead of an array, because then an arbitrary number of columns can be stored and we avoid declaring a maximum number of columns which would be a constant that would be cumbersome to determine. I found some helpful classes in the JDK, including `BufferedReader` and `StringTokenizer`, which make it easy to implement the CSV file reading. It takes some
time and searching to find good (i.e., the best) classes in the JDK, but the time is well spent.

I call `System.exit(1)` when an unrecoverable error occurs; in a real application I’d want to generate a more useful error.

I’m unsure whether I should keep the pseudocode permanently, but its not too much trouble right now.

```java
import java.util.*;
import java.io.*;
/**
 * A Table represents a database table.
 * It can store any kind of Object. The data are stored in Columns.
 *
 * @author Arthur Goldberg, NYU
 * 9/9/3
 */
public class Table
{
    private String filename;
    // the name of the file storing this table's data
    // an ArrayList of Columns, containing this table's data.
    private ArrayList theData;
    // number of rows in this table, set when the data is loaded
    private int numberRows = 0;
    // number of Columns in this table, set when the data is loaded
    private int numberColumns = 0;
    public Table(String theFilename)
    {
        theData = new ArrayList();
        filename = theFilename;
    }
    /**
     * Read a Comma Separated Values (CSV) file into a table
     * See description at (http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm)
     * Implement just a little of the "spec":
     * 1. Fields are separated by commas.
     * 2. Leading and trailing space-characters adjacent to comma field separators are ignored.
     * So John , Doe ,... resolves to "John" and "Doe", etc. Space characters can be spaces, or tabs.
     * 3. Fields with embedded commas must be delimited with double-quote characters.
     * 4. Fields that contain double quote characters must be surrounded by double-quotes, and the embedded double-quotes must each be represented by a pair of consecutive double quotes.
     * So, John "Da Man" Doe would convert to "John ""Da Man""""Doe", and a single quote that is not a delimiter is an error
     * 5. A field that contains embedded line-breaks must be surrounded by double-quotes.
     * The first record in a CSV file may be a header record containing column names
     */
    public void readCSVFile()
    {
        String line = new String();
        BufferedReader in;
        // Open the file
        if (true) // If the first line in the file cannot be read then
            throw exception;
        // Read the first line
        // Fields are separated by commas.
        // Separate the line into tokens separated by commas
        // Create a column for each token
        // Process the line
        // For each line in the file
        // Process the line
        // End for
    }
}
```
try
{
    in = new BufferedReader(new FileReader(filename));
    line = in.readLine();
    if (line == null)
    {
        // error
        System.out.println("Cannot read first line in: " + filename);
        System.exit(1);
    }
    StringTokenizer st =
        new StringTokenizer(line, ",");
    while (st.hasMoreTokens())
    {
        numberColumns++;
        theData.add(new Column(numberColumns));
        processCSVLine(line);
        numberRows++;
    }
}

catch (FileNotFoundException e)
{
    // error
    System.out.println("File not found: " + filename);
    System.exit(1);
}

catch (IOException e1)
{
    System.out.println("IOException on: " + filename);
    System.exit(1);
}

/**
* Process a line in a CSV file being read
* @param Line the line of text from the CSV file.
*/

private void processCSVLine(String Line)
{
    StringTokenizer st = new StringTokenizer(Line, ",");
    int columnNumber = 1;
    while (st.hasMoreTokens())
    {
        String field = st.nextToken().trim();
        Column theColumn = (Column) theData.get(columnNumber);
        theColumn.append(field);
        columnNumber++;
        if (numberColumns < columnNumber)
        {
            // if too many fields in the line, give up
            // TODO: maybe produce an error
            return;
        }
    }
}

/**
* Determine whether a set of columns could be a primary key
* @param columnNumbers a list of the column numbers in the set.
* @return true, if the data in the table would allow the set
public boolean couldBePrimaryKey(int[] columnNumbers) throws IllegalColumnList /*
  * pseudoCode:
  * boolean couldBePrimaryKey( whatsThisDatatype theColumns )
  * If there's a problem with theColumns then
  * throw the illegalColumnList exception
  * Create a HashSet
  * For each row in the table
  *   If any of the columns contains a NULL value then
  *     Return false
  * Assemble the columns into a single object
  * If the object is in the HashSet then
  *   return false
  *   Insert the object into the HashSet
  * End for
  * Return true
  */
{
  /*If there's a problem with theColumns then
   throw the illegalColumnList exception
  */
  if (columnNumbers.length <= 0)
  {
    throw new IllegalColumnList(" empty ColumnList");
  }
  for (int i = 1; i <= columnNumbers.length; i++)
  {
    if (columnNumbers[i] < 1
        || numberColumns < columnNumbers[i])
    {
      throw new IllegalColumnList(" column number too small or large");
    }
  }
  HashMap possiblePrimaryKeyRows = new HashMap();
  /*
  *    For each row in the table
  *      If any of the columns contains a NULL value then
  *        Return false
  * Assemble the columns into a single object
  *    If the object is in the HashSet then
  *      Insert the object into the HashSet
  */
  for (int row = 1; row <= numberRows; row++)
  {
    StringBuffer theRowValue = new StringBuffer();
    for (int i = 1; i <= columnNumbers.length; i++)
    {
      int columnNumber = columnNumbers[i];
      Column theColumn = (Column) theData.get(columnNumber);
      String fieldValue = (String) theColumn.get(row);
      // TODO: check that a String is returned
      if (fieldValue == null) // TODO: think ... do i need to check for empty
      {
        return false;
      }
      /* BUG
       * Since the fields do not contain commas we can simply use commas;
       * THIS WILL FAIL when the code is changed to use full featured CSV files
       * in which fields can contain commas
       */
      theRowValue.append(fieldValue);
      theRowValue.append(",");
    }
    if (possiblePrimaryKeyRows.get(theRowValue) != null)
    {
      return false;
    }
    possiblePrimaryKeyRows.put( theRowValue, theRowValue);
Here’s the exception.

```java
public class IllegalColumnList extends Exception {
    public String error;
    public IllegalColumnList(String description) {
        super( "Illegal columnList" + description );
        error = description;
    }
}
```

8. Perform the unit test, using existing and new test cases

I found and fixed these bugs.

Infinite loop in

```java
while (st.hasMoreTokens()) {
    numberColumns++;  
    theData.add(new Column(numberColumns));
}
```

fixed by

```java
while (st.hasMoreTokens()) {
    numberColumns++;  
    theData.add(new Column(numberColumns));
    st.nextToken();
}
```

Array indexing errors at

```java
for (int i = 1; i <= columnNumbers.length; i++) {
    if (columnNumbers[i] < 1 || numberColumns < columnNumbers[i]) {
        throw new IllegalColumnList(" column number too small or large");
    }
}
```

I now realize that I ignored my own advice and used a HashMap instead of a HashSet. Although it doesn’t matter much, since a HashSet is implemented with a HashMap, I’ll change it so I follow my advice.

Finally, it took me a while to realize and remember that the methods in Object for hashCode and equals, which are used by HashSet, only find identical objects to be equal, not objects with, loosely speaking, the same value. So to overload these operators from Object, I added an object row that stores a row and can be put in the HashSet.

At first I implemented row.equals with the signature

```java
boolean equals(row otherRow )
```

but this did not work. I set breakpoints and traced the code in the debugger and found that it was never called, of course, because equals expects the signature

```java
boolean equals( Object anOtherRow )
```

I think I’ve made this mistake before!

For my last bug, I implemented the StringBuffer comparison in row.equals with StringBuffer.equals before I remembered that I needed to use StringBuffer.compareTo().

Here’s the code.

Column.java:
import java.util.*;
/**
   * A Column represents a column of data in a table.
   * It can store any kind of Object.
   * @author Arthur Goldberg, NYU
   * 9/9/3
   */
public class Column
{
    int indexNumber;
    // the index number this Column, in the array of columns in the table
    // counts from 0 to numberOfColumns-1
    ArrayList theData;
    // an ArrayList of Strings, containing the data
    Column(int theIndexNumber)
    {
        indexNumber = theIndexNumber;
        theData = new ArrayList();
    }
    /**
     * append a field to the column.
     * @param field the object being stored in the field
     */
    void append(Object field)
    {
        theData.add(field);
    }
    /**
     * Store a field in the column.
     * @param index the index of the field in the Column.
     * @param field the object being stored in the field
     * @throws IndexOutOfBoundsException if the index is not in the Column's range
     */
    void put(int index, Object field)
    throws IndexOutOfBoundsException
    {
        theData.set(index, field);
    }
    /**
     * Get field value
     * @param index the index of the field in the Column.
     * @param index the index of the field in the Column.
     * @throws IndexOutOfBoundsException if the index is not in the Column's range
     * @return the Object at index.
     */
    Object get(int index)
    throws IndexOutOfBoundsException
    {
        return theData.get(index);
    }
}

Table.java:
import java.util.*;
import java.io.*;
/**
   * A Table represents a database table.
   * It can store any kind of Object. The data are stored in Columns.
   * @author Arthur Goldberg, NYU
   * 9/15/3
   */
public class Table
{
    private String filename;
    // the name of the file storing this table's data
    // an ArrayList of Columns, containing this table's data.
    private ArrayList theData;
    // number of rows in this table, set when the data is loaded
    private int numRows = 0;
    // number of Columns in this table, set when the data is loaded
    private int numberOfColumns = 0;
    static private String csvDelimiter = *, *
    public Table(String theFilename)
    {
        theData = new ArrayList();
        filename = theFilename;
    }
/**
*  Read a Comma Separated Values (CSV) file into a table
*  See description at (http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm)
*  Implement just a little of the "spec":
*  1. Fields are separated by commas.
*  2. Leading and trailing space-characters adjacent to comma field separators
*      are ignored.
*  So John , Doe ,... resolves to "John" and "Doe", etc. Space characters can
*      be spaces, or tabs.
*  3. Fields with embedded commas must be delimited with double-quote characters.
*  4. Fields that contain double quote characters must be surrounded by
*      double-quotes, and the embedded double-quotes must each be represented by a
*      pair of consecutive double quotes.
*  5. A field that contains embedded line-breaks must be surrounded by double-quotes
*  6. The first record in a CSV file may be a header record containing column (field)
*      names
*  */

public void readCSVFile()
{
    /*
    *  Read CSV file
    *  Open the file
    *  If the first line in the file cannot be read then
    *      Throw exception
    *  Read the first line
    *  // Fields are separated by commas.
    *  Separate the line into tokens separated by commas
    *  Create a column for each token
    *  Process the line
    *  For each line in the file
    *  Process the line
    *  End for
    */

    String line = new String();
    BufferedReader in;
    try {
        in = new BufferedReader(new FileReader(filename));
        line = in.readLine();
        if (line == null)
        {
            // error
            System.out.println("Cannot read first line in: " + filename);
            System.exit(1);
        }
        StringTokenizer st =
            new StringTokenizer(line, csvDelimiter);
        while (st.hasMoreTokens())
        {
            theData.add(new Column(numberColumns));
            numberColumns++;
            st.nextToken();
        }
        processCSVLine(line);
        numberRows++;
        while (null != (line = in.readLine()))
        {
            processCSVLine(line);
            numberRows++;
        }
    }
    catch (FileNotFoundException e)
    {
        // error
        // TODO: clearly need better error handling
        System.out.println("File not found: " + filename);
        System.exit(1);
    }
    catch (IOException e1)
    {
        System.out.println("IOException on: " + filename);
        System.exit(1);
    }
}

private void processCSVLine(String Line)
{
// Fields are separated by commas.
Separate the line into tokens separated by commas
For each token (up to the last column)
   Leading and trailing space-characters adjacent to comma field separators are ignored.
Strip whitespace at the edges
append the token to the column
End for

/*
 *  pseudo code:
 *  field is separated by comma.
 *  Separate the line into tokens separated by commas.
 *  For each token (up to the last column)
 *  Leading and trailing space-characters adjacent to comma field separators are ignored.
 *  Strip whitespace at the edges.
 *  append the token to the column
 */

StringTokenizer st = new StringTokenizer(Line, csvDelimiter);
int columnNumber = 0;
while (st.hasMoreTokens())
{
   // TODO: BUG, StringTokenizer creates neither of the two null fields in this line: ,,3,4,5
   // actually trims more than 'space and tab'; trims all control characters
   String field = st.nextToken().trim();
   Column theColumn = (Column) theData.get(columnNumber);
   theColumn.append(field);
   columnNumber++;
   if (numberColumns < columnNumber)
   {
      // if too many fields in the line, give up
      // TODO: maybe produce an error
      return;
   }
}

/**
 * Determine whether a set of columns could be a primary key
 *
 * @param columnNumbers a list of the column numbers in the set.
 *
 * @return true, if the data in the table would allow the set of columns to be a primary key, otherwise false.
 *
 * @throws IllegalColumnList if columnNumbers is empty or any of the columns in columnNumbers are illegal.
 */
public boolean couldBePrimaryKey(int[] columnNumbers)
 throws IllegalColumnList
{
   /*
   *  pseudoCode:
   *  boolean couldBePrimaryKey( whatsThisDatatype theColumns )
   *  If there's a problem with theColumns then
   *     throw the IllegalColumnList exception
   *  Create a HashSet
   *  For each row in the table
   *     If any of the columns contains a NULL value then
   *        Return false
   *     Assemble the columns into a single object
   *     If the object is in the HashSet then
   *        return false
   *     Insert the object into the HashSet
   *  End for
   *  Return true
   */
   if (columnNumbers.length <= 0)
   {
      throw new IllegalColumnList(" empty ColumnList");
   }
   for (int i = 0; i < columnNumbers.length; i++)
   {
      if (columnNumbers[i] < 0
          || numberColumns <= columnNumbers[i])
      {
         throw new IllegalColumnList(" column number too small or large");
      }
   }
   /*
   *  For each row in the table
   *     If any of the columns contains a NULL value then
   *        Return false
   *     Assemble the columns into a single object
   *     If the object is in the HashSet then
   *        return false
   *     Insert the object into the HashSet
   *  End for
   */
   HashSet possiblePrimaryKeyRows = new HashSet();
   for (int row = 0; row < numberRows; row++)
   {
      StringBuffer theRowValue = new StringBuffer();
      for (int i = 0; i < columnNumbers.length; i++)
      {
         int columnNumber = columnNumbers[i];
   */
Column theColumn = (Column) theData.get(columnNumber);
String fieldValue = (String) theColumn.get(row);

// TODO: check that a String is returned, handle classCast exception
if (fieldValue == null) {
    return false;
}
// check for empty string too?
if (fieldValue == null) {
    return false;
}

/* BUG
   Since the fields do not contain commas we can simply use commas;
   THIS WILL FAIL when the code is changed to use full featured CSV files
   in which fields can contain commas. For example, it would say
   that these two different rows were the same:
   A, B,C
   A,B, C
*/
theRowValue.append(fieldValue);
theRowValue.append(csvDelimiter);

row theRow = new row(theRowValue);
if (possiblePrimaryKeyRows.contains(theRow)) {
    return false;
}
possiblePrimaryKeyRows.add(theRow);

return true;

class row
{
    StringBuffer theRowValue;

    row(StringBuffer theRowValue)
    {
        this.theRowValue = theRowValue;
    }

    /**
     * Generate a hashCode for this row that depends only on the value of
     * its 'theRowValue'. Therefore, two 'rows's with equal values for
     * theRowValue have equal hashCodes.
     *
     * @return the hashCode
     */
    public int hashCode()
    {
        int hash = 0;
        for (int j = 0; j < theRowValue.length(); j++)
        {
            hash += theRowValue.charAt(j);
        }
        return hash;
    }

    /**
     * Determine whether two 'row's are equal, that is, whether their
     * theRowValues are equal.
     *
     * @param otherRow the other row.
     * @return true, if the 'row's have equal rowValues, otherwise false.
     */
    public boolean equals(Object anOtherRow)
    {
        row otherRow = (row) anOtherRow;
        if (this.theRowValue.length() != otherRow.theRowValue.length())
        {
            return false;
        }
        for (int j = 0; j < this.theRowValue.length(); j++)
        {
            if (otherRow.theRowValue.charAt(j) != this.theRowValue.charAt(j))
        }
        return false;
    }

/*
3 This is a lingering bug, present because we entered the wrong code, and cvs didn’t input NULL fields, so
it was not tested.
Test.java:
/**
 * Ugly code for testing Table and Column.
 * @author Arthur Goldberg, NYU
 * @ *
 * 9/15/3
 */
public class Test {
  // test cases
  // TODO: would be nicer to make a single array, rather than 2 parallel arrays
  static int output = 0;
  static int filename = 1;
  static int description = 2;
  static String[][] testResultsAndFiles = {  // output, filename, description
      { "Throw illegalColumnList", "Empty table", "Empty table" },
      { "Throw illegalColumnList", "file not read", "No columns indicated" },
      { "Throw illegalColumnList", "file not read", "negative column value" },
      { "Throw illegalColumnList", "file not read", "huge column value" },
      { "true", "Table with 1 row", "Table with 1 row, no NULLs" },
      { "false", "Table with NULL values", "Table with NULL values" },
      { "false", "Table with duplicate rows", "Big table with duplicate rows" },
      { "false", "Table with duplicate first and last row", "" },
      // { "false", "big Table with all rows the same", "" },
      { "false", "Table with last 2 rows the same", "" },
      { "false", "Table with first 2 rows the same", "" },
      { "true", "lack of a separator", "A table like this, in which a lack of a separator could cause mistaken duplicates:1 23 12 3" }};
  static int[][] testColumnLists = {  // column list
      {0},
      {},
      {1, 0, 3},
      {0, 1, 0, 3},
      {0, 1, 3, 2},
      {0, 1, 3, 2},
      // {0, 1, 3, 2},
      {0, 1, 3, 2},
      {0, 1, 3, 2},
      {0, 1, 3, 2},
      {1, 2},
  };

  public static void main(String[] args) {
    String directory = "F:\\TEMP\\csv tests\";
    String suffix = ".csv";
    // loop through tests
    for (int i = 0; i < testColumnLists.length; i++) {
      testOneFile(
        testColumnLists[i],
        testResultsAndFiles[i][output],
        directory + testResultsAndFiles[i][filename] + suffix,
        testResultsAndFiles[i][description]);
    }
  }

  static void testOneFile(
    int[] columnList,
    String output,
    String file,
    String description)
  {
    Table t = new Table(file);
    t.readCSVFile();
    boolean result;
    try {
      return false;
    }
    return true;
  }
}
result = t.couldBePrimaryKey(columnList);
System.out.println("Result is: 
+ result + "\n" + "Result should be: " + output + "\n" + "File: "+ file + "\n" + description+ "\n" );
}

} catch (IllegalColumnList e)
{
    System.out.println(e + " thrown: 
" + "Result should be: " + output + "\n" + "File: "+ file + "\n" + description+ "\n" );
}

} IllegalColumnList.java:
public class IllegalColumnList extends Exception
{
    public String error;
    public IllegalColumnList(String description)
    {
        super("Illegal columnList" + description);
        error = description;
    }
} Todo: show test case input and output.

Is a Column in One Table a Possible Foreign Key for a Column in the Other Table?

This method could have the signature

boolean possibleForeignKey(Table table1, int Column1, Table table2, int Column2);

Right?

What are the semantics of possibleForeignKey? How about:

/**
 * Determine whether a given column in one table (Table1) could be a possible foreign key for a given column in another table (Table2).
 * First, the column in Table2 must be a primary key, that is, the column in Table2 must have no NULL entries, and each entry must be distinct.
 * Second, each entry in the column in Table1 must be non-NULL and contain a value that is in an entry of the column in Table2.
 * Note that Table1 and Table2 could be the same table.
 *
 * @param table1 Table1.
 * @param Column1 The number of the column in Table1.
 * @param table2 Table2.
 * @param Column2 The number of the column in Table2.
 * @return True if Column1 in Table1 could be a foreign key for Column2 in Table2.
 */

What pseudocode would describe this program?
How would you do this assignment?