CPS352 - DATABASE SYSTEMS

Programming Project

Part I due Monday, March 27, at the start of class
Part II Milestone due Monday, April 17, at the start of class
Final Submission due Monday, May 1, at the start of class

For this project, you will be implementing a GUI application for managing a database meeting some of the information needs of a hypothetical library. The project must be done in teams of two.

Each team may use a single database installed on one of their computers, or each team member may choose to run the project on their own computer. In either case, use the database(s) that were created when the Db2 container was installed (`project`). Even if both computers are used, though, t**he team should share one version of the files - not two separate versions.**

You will be given a sql command file (`createdb.sql`) for creating the database objects. As given to you, it will create all the tables and attributes you will need, plus two variables and two triggers. You will need to add appropriate constraints and two additional triggers. You will also be given a file to delete the objects you have created (`dropdb.sql`) which drops the tables in reverse order from the way they were created. You will need to use this after making changes to the creation code before re-creating the tables,

You will also be given compiled code for the GUI. Your task is simply to complete the implementation of one class: `Database`, which connects the OO GUI to the relational database. This will involve adding code to the file `Database.sqlj`, You will also create a test file in `test.dat`.

Attached as Appendices A-E are an E-R diagram that has been produced by a systems analyst based on conversations with the staff of the library at East Podunk State University, plus a database design based on this E-R diagram, plus lists of functional and non-functional requirements for the system developed by this analyst, plus formats for the various printed reports. (Please note: you are to implement the database design exactly. You may have good ideas about additional information that should be represented, or other ways to represent the information, but for the purposes of the project you are to follow the analysis as given, because your executable code will be tested automatically by the professor.)

Your project will be turned in in two parts, with a milestone for the second part. Specific requirements for each are below.

Starter Files

The file `CPS352SemesterProgrammingProject.zip` on Canvas contains the four files you will need to edit for the project and four other files (that you will not need to modify) that you will use in building your project. Both team members may keep a copy of this folder on their computers, but you should turn in just one copy for the team for assignments.

Download this zip file into the directory you created for sharing files with your container, and then expand it. This will produce a folder named `CPS352SemesterProgrammingProject` which will also appear in `/database` in your container. On your host, rename this folder to teamX - where X is the number assigned to your team (this will also change the name in the `database` folder). In order for the build process to work properly, it is necessary to avoid changing the location of any file within this hierarchy. You can edit the files on your host computer using whatever text editor you desire. Changes made to files on the host will also appear in `/database` in the container.

What Will be in the Starter Files Directory `teamX`:

Files you will modify:

`createdb.sql` - a command file for creating the database objects - you need to add constraints and additional triggers. `create trigger` statements should be added at the end, after all the tables are created.
dropdb.sql - a command file for dropping the database objects so you can re-create them. You will need to add commands to drop any triggers you create. You will need to add your triggers to this, since any triggers should be dropped before dropping the table they reference. (Triggers should be added at the start of this file.)
`semesterproject` containing `semesterproject/Database.sqlj` - the source to which you will add code. (To satisfy Java requirements, `Database.sqlj` will be in the subfolder named `semesterproject`)
`test.dat` - a starter file for test data for testing your project - see Appendix J.

Files you will not change:

`Bind.class` - executable support file used when building your code
`Fine.html` - Javadoc documentation for the `Fine` class your code will use.
`makefile` - commands for building your code.
`project.jar` - a compiled file containing the GUI and some other code you will not need to write.

Project Requirements

Part I Requirements (30% of project grade) - Turn in as specified below.

1. A canonical cover for the complete set of functional and multivalued dependencies for the database scheme as given in Appendix B. **Note:** Most of the FDs can be inferred from the schema diagram - but not all - so also look carefully at the non-functional requirements in Appendix D. The MVD's can be inferred from the ER diagram in Appendix A. Turn in a single **Word or pdf document** to the Semester Project Part I assignment on Canvas.

2. A modified SQL command file for creating the database (`createdb.sql`) that incorporates constraints and triggers. **Turn in your complete project folder with name changed as noted above** to DropBox at the link specified on in the Part I assignment on Canvas.

3. **• Be sure to give your foreign key and check constraints names.** Your code will need to recognize these names in order to produce "friendly" error messages. **You will define these as symbolic constants in** `Database.sqlj`**- note how provision has been made for this in some cases in the file you will be given - but there are many others needed as well, and be sure to fill in the actual constraint names in the symbolic constant definitions in** `Database.sqlj`. Remember that SQL is not case-sensitive, so regardless of what case you use for these names, Db2 will convert them to all upper-case in messages. Therefore, the values of the symbolic constants you define should be the all upper-case equivalents of what is used in `createdb.sql`.

. **• Be sure that triggers that disallow an operation by throwing an error report a recognizable SQLState.** Again, your code will need to recognize these values in order to produce "friendly" error messages, **so be sure to define appropriate symbolic constants in** `Database.sqlj`.

. **• Of course, the file you turn in needs to work correctly.** I will verify that this works correctly.

4.

Notes:

a. The foreign key constraints can be inferred from the schema diagram. Some - but not all - should have cascading delete. Include `on delete cascade` only in cases where there is something in the requirements that explicitly requires it, but do not include otherwise.
b. Be sure to include the `check` constraints necessitated by non-functional requirements.
c. You will find the Birchall chapter discussing triggers to be very helpful. The rule requiring that all information about a book should be deleted when the last copy is deleted and the "can't renew an overdue book" rule will require triggers. These have been written for you in the `createdb.sql` file given to you.
d. You will need to create triggers for assessing a Fine when a book is overdue and for forbidding checking out too many books Use the two triggers written for you as examples of how to write a trigger that performs something additional after a legitimate operation or forbids an illegal operation before it happens by signaling an exception,.
e. Note that you are actually submitting two things on Canvas: a file of dependencies and a SQL command file. **The former must be turned in as a Word or pdf file and the latter must be turned in to DropBox.**

Part II Requirements (70% of project grade)

1. Stylistically correct and completely operational code in `Database.sqlj` that fulfills all functional and non-functional requirements. **Be sure you have followed the "single statement" requirement for reports discussed in Appendix E!**

2. A file of test data used to exercise your code in `test.dat`, and a report showing the results of testing (results of executing your test data) (see Appendix J). Turn in a screeenshot of your testing, being sure to **use a monospaced font so it is readable!** This will be evaluated in two ways:

   • Thoroughness of testing requirements - both legitimate data and error detection.

   • Analysis of results. If your testing uncovers an error that you cannot fix, make it clear on your test report that you noticed the error, and indicate - as best you can - what it is. It's better for your project grade for you to report that you found the error than for me to find it!

Milestone: Implement just the borrower operation portion of the final requirements (add, get, update, delete, report) in `.Database.sqlj` **Of course, test your code to be sure that the file you turn in needs to work correctly.** I will read it to see if it looks correct, and may test it to verify apparent errors - and note that I can often spot errors by reading, so be sure there are none there for me to find! The milestone will not be graded separately, but credit for turning in apparently correct code on time will be included in the Part II grade. **Turn in your complete project folder with name changed as noted above** to the DropBox link in the Milestone assignment on Canvas.

Final Submission: Turn in your complete project folder containing all the files that were there initially (possibly edited by you) to the DropBox folder listed on Canvas: Your screenshot and analysis should be turned in on Canvas as a **single Word or pdf document** - with the screenshot using a **monospaced font if created via cut and paste**.

Each team should make one submission for each of the above reflecting their joint work.

Editing, Building, and Running your Program

You will probably edit you files on your host system, build your program inside the container, but you run it on your host computer.

To edit your files

1. You may edit your files in `teamX` in the shared directory on your host computer or in `/database` in your container. However, you will probably prefer to do your editing on your host computer using a familiar text editor rather than using vi, which is the only text editor in the container.

2. Because there may be some delay between modifying a file in one place and the modification showing up in the other place, you should be consistent about where you edit your files and should be sure that modifications made on the host have shown up in the container.

To build your database

1. Edit the file `createdb.sql` to add appropriate constraints and triggers. Also add your names and team number to the initial comment. (Note: distinct types have not been used to minimize complications in the code.) Also edit `dropdb.sql` to drop any triggers that you created before the table(s) that they reference are dropped. **Do not forget to add dropping triggers to `dropdb.sql`.**
2. Enter the container as the database administrator, not root.
3. Change directory to `/database/teamX`.
4. Issue the following as a shell command:

   ```
   db2 -t +p < createdb.sql
   ```

5. Scan the output from Db2 carefully to identify and correct any errors in the file it detected.
6. Always use `dropdb.sql` before retrying `createdb.sql` to drop any tables that were successfully created to avoid an error caused by recreating an object that already exists. You can execute this similarly to the way you executed `createdb.sql`
7. You will also need to repeat the previous steps after modifying the file itself to add something you discover later that was omitted.

To build your program

1. Edit `Database.sqlj` to add SQL code to carry out the various operations. Also add your name and team number to the initial comment. You would be wise to work incrementally - i.e. add a bit of sql code, build and run your program, and then test that the added code works correctly before moving on to the next.
2. Enter the container as the database administrator, not root.
3. Change directory to `/database/teamX`.
4. Issue the shell command `make`, which will run the commands in the makefile. See detailed directions in Appendix F.

To run your program

1. Be sure both docker and the Db2 container are running. (This should normally be the case without any special effort on your part, but you can verify by trying to access the database as you have done at other times)

2. On your host computer, change directory to the directory in the folder you shared with the container (where all you project work is).

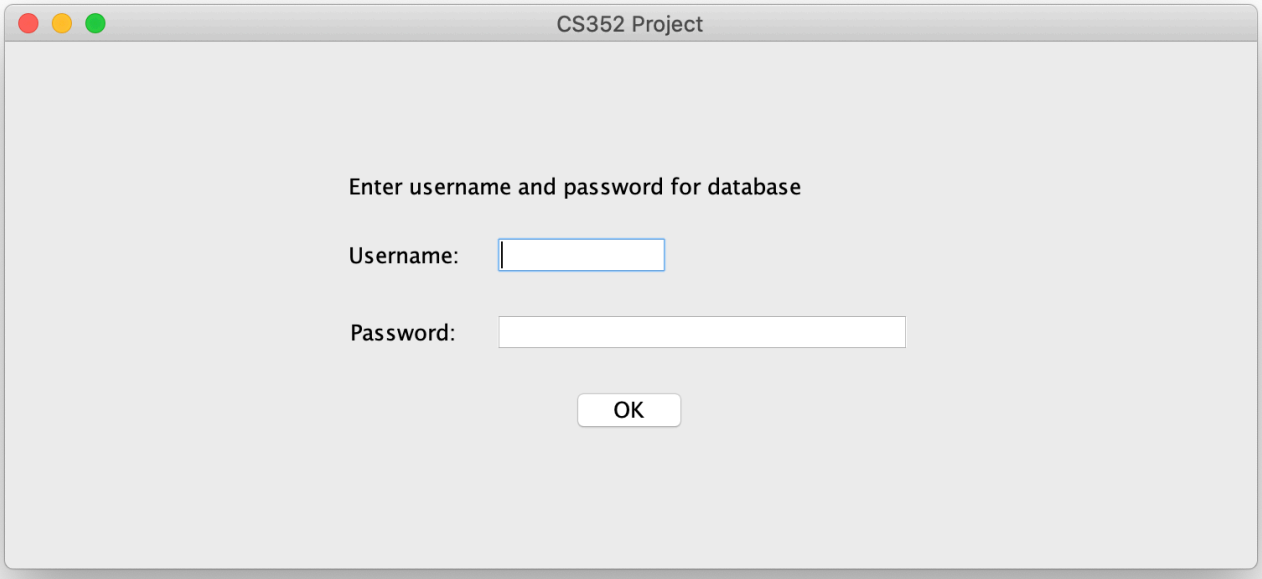
3. Issue the following command-line command

```
java -jar project.jar
```

4. The GUI for the project will pop up looking like below.

5. Enter the username `db2inst1` and t**he password you specified for this user when you created the container (not** your Gordon password).

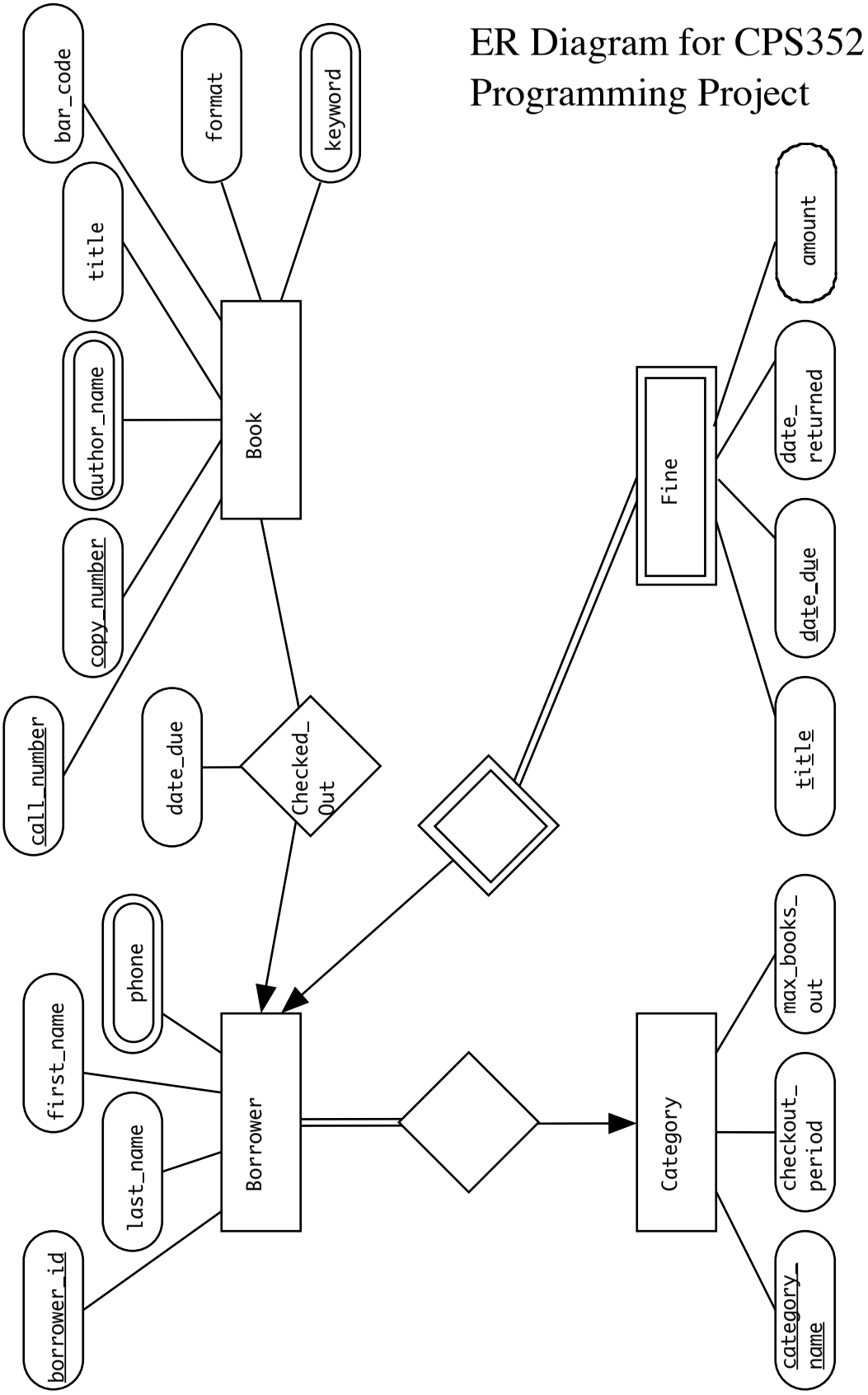
6. If you forgot the password you chose for `db2inst1` in the container - or it is too much to have to type repeatedly - you can enter the container as root and then change the password by using the shell `passwd` command. As root, you can get away with any password you want, even you are warned that it is insecure - and in this case security isn't really an issue anyway.




```
passwd db2inst1
```

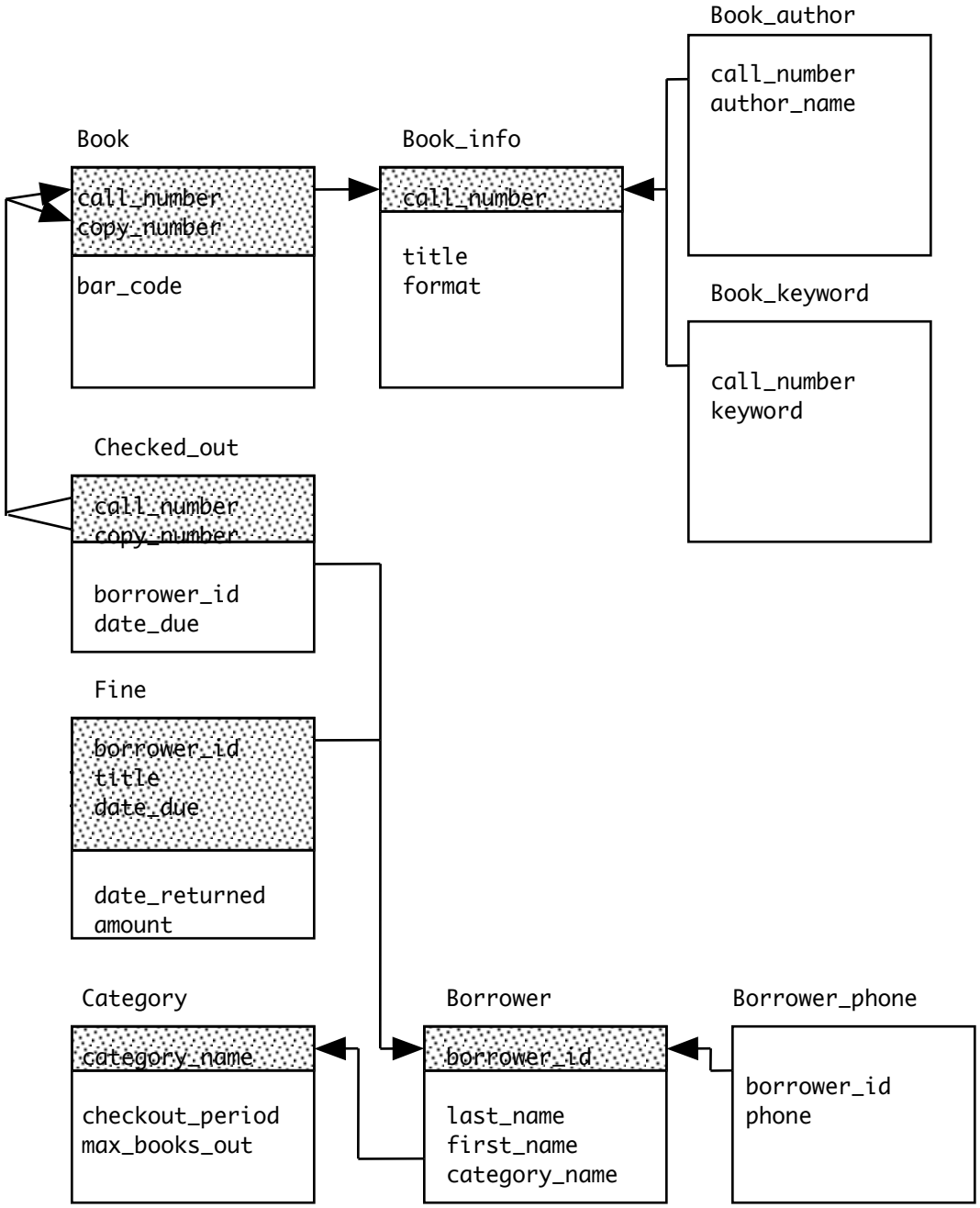
APPENDIX A

Note: Some relationships that will not actually become distinct tables in the database schema have not been given names below.



APPENDIX B - NORMALIZED RELATIONAL DATABASE DESIGN FOR CPS352 PROGRAMMING PROJECT

This design corresponds to the ER diagram above. It has more tables, due to the requirements of normalization. Many relationships have been represented by using appropriate foreign keys directly in the participating entity tables, rather than by creating a separate table for the relationship. (However, Checked_out is made a separate table.). It basically corresponds to the set of tables one would get if one started with a universal relation, except in the case of Checked_out.



APPENDIX C - FUNCTIONAL REQUIREMENTS FOR CPS352 PROGRAMMING PROJECT

The database system for the library of East Podunk State University must support the following functions:

1. Allow librarians to directly maintain the various strong entities that appear in the ER Diagram (Books, Categories, Borrowers). It must be possible to add a new entity, to edit an entity (changing any non-key information) or to delete an entity.

   • In the case of books, it must be necessary to enter title, format, author and keyword information for a given call number only once. Subsequent copies added with the same call number should use the information already entered. In the case of a change of title, format, author(s) or keyword(s) for a given call number, the changes will apply to all copies of that call number.

   It should be possible to delete an individual copy of a book. This will not affect the other copies of the same book. However, if, as a result, the library no longer has any copies of that call number, then the remaining information about the book (title, author(s), format, keyword(s)) must be automatically deleted as well.

   It is possible to delete a book that is checked out; in this case any row for it in the Checked_out table must be deleted as well. (This might happen, for example, if a borrower reports a book lost.)

   It is possible to delete a book that has a fine outstanding on it, but the fine must <u>not</u> be deleted.

   • A Borrower who has books recorded as checked out cannot be deleted from the database. The database must enforce this.

   • A Borrower who has outstanding fines <u>can</u> be deleted from the database. If this happens, the database must ensure that the Fines are deleted as well.

2. Check out book(s) to a borrower. The length of time the book is checked out for (and hence the due date) is determined by the Borrower's category.

   It is not permissible to check a book out to a borrower if the result would be that the borrower has more books out than allowed for his/her category. The database must enforce this.

3. Return book(s). If a book is returned late, an appropriate fine will be assessed.

4. Renew book(s). The renewal period is the same as the original checkout period, calculated based on the <u>original</u> date due (not the date on which the book is renewed) if the book is on time. There is no limit on the number of times a book can be renewed. However, a book that is overdue cannot be renewed. If a borrower attempts to do so, an error should be reported and the database should not be changed.

5. Maintain a record of fines assessed when items are returned late.

6. Record the payment of fine(s) by a borrower.

7. Look up books by keyword. (This requirement will not actually be implemented for this project, though you will produce a report that contains this information)

8. Produce reports. The specific reports required, and the format of each, is shown in Appendix E.

APPENDIX D - NON-FUNCTIONAL REQUIREMENTS FOR CPS352 PROGRAMMING PROJECT

1. Entity integrity will be enforced for all entities. [That is, the primary key specified in the database design (Appendix B) will be enforced.]

   a. Although call_number/copy_number is considered the primary key for the Book entity, it is also the case that each book must have a unique bar code. The database must enforce this.

   b. It is possible to have two borrowers with the same last name and first name.

2. Referential integrity will be enforced for all relationships. [That is, appropriate foreign key constraints will be enforced for each relationship.]

3. Domain integrity will be enforced for all attributes.

   a. No attributes can be null.

   b. The value of the format attribute of a Book must be one of the following: HC, C SC, D, MF, or PE (denoting hardcover book, soft cover book, CD (or DVD), microfiche, or periodical, respectively.) The database must enforce this.

   c. Each keyword for a book will be a single word, with no spaces, though embedded underscores are allowed - e.g. '`DATABASE`' is an acceptable keyword for a book, and '`MANAGEMENT`' is an acceptable keyword, but '`DATABASE MANAGEMENT`' is not. However, '`DATABASE_MANAGEMENT`' would be acceptable, as would each word individually. The database must enforce this.

APPENDIX E: REPORT FORMAT REQUIREMENTS FOR CPS352 PROGRAMMING PROJECT

The various reports that are generated by buttons on the Reports pane should be printed to System.out. Each report should be formatted in accordance with the samples below, and should be printed in the order specified. Each report should be generated by outputting the values returned by a **single** select statement, and ordering should be handled by an order by clause. In the case of the Fines report, doing the report with a single statement is tricky, but it can be done. Look at the code given you for `booksReport()` to see how a report requiring multiple lines for one entity can avoid repeating common information - e.g. the BookInformation Report for a book with multiple authors shows the Call Number, Title, Format and # copies just once though it produces a separate line for each author. The starter code for each report include symbolic constant definitions that you should use in generating your report to ensure uniformity of format with the master your reports will be compared to.

Book Information Report (in order of call number)

Call Number	Title	Author	Format	# copies
QA76.9.D3 D3659 1995	An Introduction to Database Systems	Date	HC	1
QA76.9.D3 R237 2000	Database Management Systems	Ramakrishnan Gehrke	HC	1
QA76.9.D3 S5637 2002	Database System Concepts	Korth Silberschatz Sudarshan	HC	2

Book Copies Report (in order of bar code)

Bar Code	Call Number	Copy #	Title
807	QA76.9.D3 S5637 2002	1	Database System Concepts
808	QA76.9.D3 D3659 1995	1	An Introduction to Database Systems
809	QA76.9.D3 R237 2000	1	Database Management Systems
810	QA76.9.D3 S5637 2002	2	Database System Concepts

Book Keywords Report (in order of keyword, then call number within keyword)

Keyword	Call Number	Title
DATABASE	QA76.9.D3 D3659 1995	An Introduction to Database Systems
DATABASE	QA76.9.D3 R237 2000	Database Management Systems
DATABASE	QA76.9.D3 S5637 2002	Database System Concepts
PJNF	QA76.9.D3 D3659 1995	An Introduction to Database Systems
RELATIONAL	QA76.9.D3 D3659 1995	An Introduction to Database Systems
RELATIONAL	QA76.9.D3 R237 2000	Database Management Systems
RELATIONAL	QA76.9.D3 S5637 2002	Database System Concepts

Categories Report (in order of category name)

Category Name	Maximum Books Out	Checkout Period
BigCheese	100	1000
Peon	5	14

Borrowers Report (in order of borrower id) - Note that a single borrower can have multiple phones that should be listed on separate lines without repeating the rest of the borrower information - this must be done by a single sql query handled as in the Book Information Report done as an example. Study the code given to you carefully, and note how it uses a method called produceReport() (given to you) that takes care of suppressing duplicate lines - study it carefully to be sure you understand how to use it.]

Borrower ID	Last Name	First Name	Phone #	Category
12345	Aardvark	Anthony	123-4567 x9999	BigCheese
20123	Buffalo	Bill	555-1212	Peon

Fines Report (in order of total fine amount owed - largest first) - Note that a single borrower can have multiple phones that should be listed on separate lines without repeating the rest of the borrower information - this must be done by a single sql query handled as in the Book Information Report done as an example. Yes, this can be done by using a single query, but it's tricky - so you're likely to need to see me for help. (The key is to embed a `select` expression in the `from` clause of the main query to generate the totals. See Birchall pages 175 and 176 for the requirement that you must use the `table` function in this case,)

Last Name	First Name	Phone #	Total Fines
Buffalo	Bill	555-1212	12.00
Aardvark	Anthony	123-4567 x9999	0.05

Overdue Books Report (in order of borrower last name, then first name, then date due)- Note that a single borrower can have multiple overdue books that should be listed on separate lines without repeating the rest of the borrower information - this must be done by a single sql query handled as in the Book Information Report done as an example - see the comment on BorrowersReport above.

Last Name	First Name	Title	Date Due
Aardvark	Anthony	Database System Concepts	10/10/2018
		An Introduction To Database Systems	10/15/2018
Buffalo	Bill	Database System Concepts	10/01/2018

APPENDIX F: EMBEDDED SQL MECHANICS

For this project, you will use embedded SQL: the program contains a mixture of ordinary Java statements and SQL. Both are translated as part of the program compilation process (which is now a bit more complex). In general, static SQL offers faster program execution, because the SQL statement is compiled once, rather than being interpreted each time it is to be executed. This also allows the compiler to expend a fair amount of effort on planning an efficient strategy for each query. (When we talk about this topic in lecture, you'll see why this can be a significant advantage of compiled code.) For this reason, most significant DBMS applications make use of embedded SQL. See Appendix G for a discussion of embedded SQL syntax.

When you build your program using the makefile as described above, the following occurs:

1. The original `Database.sqlj` file - containing embedded SQL - is first converted into two files - one "pure Java" file (`Database.java`) and one containing a translated form of the SQL code. The SQL code will eventually be bound to the database. The "pure Java" file contains code which causes the SQL code to be run when needed. (You should feel free to look at the file Database.java produced by the translator to see what it does, but don't expect to fully understand the generated code!) This is done by the `sqlj` command in the `makefile`.

2. The "pure java" output from this operation is now translated into standard class files. This is done by the `javac` command in the `makefile`. Any errors reported by the java compiler will be reported in terms of line numbers in the pure java file, but will need to be fixed in `Database.sqlj`, so you will need to look for the line corresponding to the reported error in that file.

3. The SQL code is now bound to the database. This is done by the `bind` command in the `makefile`.

   You will need to supply the password you specified when you created the container when prompted for it. (Note: the program invoked in the `makefile` actually runs another program supplied by IBM which needs your password to access the database. However, the IBM-supplied program requires the password to appear on the command line, which is *horrible* security. This program gets your password from you in a more secure way and then runs the IBM-supplied program.) Be sure you get the following message indicating successful completion (at the end of a series of messages):

   ```
   [ibm][db2][jcc][sqlj] Bind complete for Database_SJProfile0
   ```

   If the binding process results in one or more errors, the error message(s) will report a SQL code, You can find the explanation of these using `? SQLxxxx` in interactive Db2.

APPENDIX G: USING EMBEDDED SQL IN A JAVA PROGRAM

To assist you in writing your embedded SQL, I have already written the code in `class Database` necessary to perform the various operations on the entity "Category", including the printing of the Categories report. Study all this code carefully - not just the portions discussed below.

Example 1: The following code is called when the user enters information about a new category into the appropriate screen and then clicks "OK" to add it to the database:

```
/** Add a new category to the database.
 *
 *  @param categoryName the name of the category to add
 *  @param checkoutPeriod the period borrowers in this category can check
 *         books out for
 *  @param maxBooksOut the maximum number of books borrowers in this category
 *         can have out
 *
 *  @exception an ErrorMessage is thrown with an appropriate message if
 *          the operation fails.  Any failure results in no changes being
 *          made to the database
 */
public void addCategory(String categoryName,
                            int checkoutPeriod,
                            int maxBooksOut) throws ErrorMessage
{
    try
    {
        #sql { insert into category
                        values (:categoryName, :checkoutPeriod, :maxBooksOut)
             };
        #sql { commit };
    }
    catch(SQLException e)
    {
        rollback();
        if (e.getErrorCode() == DUPLICATE_KEY_SQL_ERROR)
            throw new ErrorMessage(
                  "Category name is the same as an existing category");
        else
            throw new ErrorMessage("Unexpected SQL error " + e.getMessage());
    }
 }
```

This method contains two embedded SQL statements - an `insert` and a `commit`. Observe :

• Embedded SQL statements are preceded by `#sql`, enclosed in braces, and followed by a semicolon after the closing brace. Each embedded SQL statement is "wrapped" separately.

• An embedded SQL statement can reference variables occurring in the surrounding code (called host variables.) Host variables are preceded by a colon, and use the name used by the Java code. (Any name not preceded by a colon is taken as being a name defined in the database.)

• Embedded SQL must be enclosed in a `try ... catch` block. Any problem detected by the DBMS causes a `SQLException` to be thrown. The Java `catch` block must check the exception code to determine exactly what went wrong, so that it can display an appropriate message - see discussion of this in the implementation notes.

• Every operation on the database must be followed either by a `commit` or by a `rollback`. An operation that simply reads the database without altering it must still be followed by `rollback` - there are no changes to commit - else the DBMS will consider it part of the next transaction.

• If a SQL operation fails due to the violation of some constraint, it must be rolled back - otherwise its failure will pollute the next transaction. Because a rollback can potentially throw an exception, the procedure `rollback()` has been written for you to handle this case. It writes an error message to System.out and aborts the program if the rollback also fails (a grave problem!).

Example 2: The following code is called when the user clicks the "Edit" or "Delete" button in the category pane, after specifying the name of a category to be edited or deleted. Its task is to look up the existing information in the database and send it to the screen which displays it and allows the user to change it.

```
/** Get information on an existing category about to be edited or deleted
 *
 *  @param categoryName the name of the category
 *  @return values recorded in the database for this category - an array
 *          of strings.
 *  @exception an ErrorMessage is thrown with an appropriate message if
 *          the category does not exist
 */
public String[] getCategoryInformation(String categoryName) throws ErrorMessage
{
    String [] values = new String[3];
    values[0] = categoryName;
    int checkoutPeriod, maxBooksOut;
    try
    {
        #sql {   select checkout_period, max_books_out
                        into :checkoutPeriod, :maxBooksOut
                        from category
                        where category_name = :categoryName
             };
        values[1] = "" + checkoutPeriod;
        values[2] = "" + maxBooksOut;
        return values;
    }
    catch(SQLException e)
    {
        if (e.getSQLState().equals(NO_ROW_SQL_STATE))
            throw new ErrorMessage("No such category");
        else
            throw new ErrorMessage("Unexpected SQL error " + e.getMessage());
    }
    finally
    {
        rollback();
    }
}
```

• Since this code only serves to access information, it is appropriate to always follow it by a `rollback`, whether it succeeds or fails. This is handled by using a `finally` block after the `try ... catch`. Code inside a `finally` is absolutely guaranteed to be done, regardless of how the other code completes.

• There are two variants of the `select` statement that can be embedded in a program. This example used a singleton select (`select ... into`.) This may only be used when it is expected that a `select` will return exactly one row (no more and no less.).

• Ordinarily, a select statement that found no rows would not be an error; but an exception is thrown in this case due to the singleton select. An error arises because the code requires us to assign values to the various host variables, but there are no values to assign if no row was found. (An exception would also be thrown if the select statement produced a result of more than one row. A singleton select should only be used in cases where it is known that this cannot occur.)

Example 3: The next example uses the other variant, which must be used if a query could potentially return more than one row. This code is used to return a list of fines owed by a specific borrower.

```
/** Get a list of fines owed by a particular borrower
 *
 *  @param borrowerID the borrower whose fines are wanted
 *  @return a list of Fine objects - one for each fine the borrower owes
 *
 *  @exception an ErrorMessage is thrown with an appropriate message if
 *          there is no such borrower, or the borrower has no fines.
 */
#sql iterator FineCursor(String, short, String, Date, Date, int);
public List getFines(String borrowerID) throws ErrorMessage
{
    List result = new ArrayList();
    try
    {
        FineCursor cursor;
        // The compiler doesn't catch that the fetch below initializes
        // these variables, so give them default values now to keep the
        // compiler happy
        String callNumber = null;
        short copyNumber = 0;
        String title = null;
        Date dateDue = null;
        Date dateReturned = null;
        BigDecimal amount = 0;
        #sql cursor = { select fine.call_number, copy_number, title,
                                    date_due, date_returned,
                                    amount
                                from fine join book_info
                                  on fine.call_number = book_info.call_number
                                where borrower_id = :borrowerID
                                order by date_returned
                      };
        while(true)
        {
            #sql { fetch :cursor into :callNumber,
                                    :copyNumber,
                                    :title,
                                    :dateDue,
```

```
                                        :dateReturned,
                                        :amount
                    };
            if (cursor.endFetch()) break;
            Fine fine = new Fine(callNumber, copyNumber, title,
                                            dateDue, dateReturned, amount);
            result.add(fine);
        }
        cursor.close();
    }
    catch(SQLException e)
    {
        throw new ErrorMessage("Unexpected SQL error " + e.getMessage());
    }
    finally
    {
        rollback();
    }
    if (result.size() == 0)
    {
        // No fines found - either the borrower doesn't exist, or has no fines
        // The following determines which is the case
        boolean borrowerExists;
        String dummy = null;
        try
        {
            #sql { select last_name into :dummy
                            from borrower
                            where borrower_id = : borrowerID };
            borrowerExists = true;
        }
        catch(SQLException e)
        {
            // If the above statement failed because no row was found,
            // this means the borrower doesn't exist

            if (e.getSQLState().equals(NO_ROW_SQL_STATE))
                  borrowerExists = false;
            else
                  throw new ErrorMessage("Unexpected SQL error " + e.getMessage());
        }
        finally
        {
            rollback();
        }
        if (borrowerExists)
            throw new ErrorMessage("This borrower has no fines");
        else
            throw new ErrorMessage("No such borrower");
    }
    else
        return result;
}
```

• This code uses the other form of `select` statement possible in embedded SQL - using a cursor. A cursor must be used when a `select` can meaningfully return either no or multiple rows. (It could also be used if you knew the query would return a unique row, but in this case the singleton select is much simpler.) Note that the distinction is made on the basis of what result the query might produce.

• When using as cursor, the cursor type must be declared outside of the method in which it is used. The cursor declaration specifies the (Java) types of the various columns that will be specified by the `select` statement. The cursor declaration is preceeded by `#sql`, but is not "wrapped" in braces - though it is terminated by a semicolon. The sqlj processor turns the cursor type declaration into a new class whose name is the one you give (e.g. FineCursor), with instance variables of the types specified.

• A cursor variable is declared inside the method. It is a Java variable like any other variable.

• The cursor is associated with a particular `select` statement by a special kind of assignment statement. Note carefully the syntax - especially the placement of the braces. (Note: in other languages hosting embedded SQL, this is commonly called "opening" the cursor.)

• Each row from the result of the `select` is fetched separately, using a `fetch` statement. The `endFetch()` function becomes true when fetch fails to find a next row. (Hence, you must perform this test before actually trying to use the values fetched)

• After being used, the cursor is closed and the transaction is rolled back.

• Note that this example actually illustrates the use of both types of `select` statement. If the original statement fails to find any fines for a given borrower, there are two possibilities: either the borrower doesn't exist at all, or the borrower exists but has no fines. A singleton select is used at the end of the method in the case where there are no fines to determine which is the case.

APPENDIX H: USING THE GUI THAT HAS BEEN FURNISHED FOR YOU

When you run the program, you will first be presented with a login screen, requiring you to establish a connection to the database using the appropriate username and password.

• You can run the GUI with dummy data - without actually accessing any database - by entering a username of `none` (actually type this word into the GUI) leaving the password blank. Note that you can even do this on a computer that does not have db2 installed. (You should ignore any error message about being unable to find the db2 driver in this case.)

- To interact with the database, enter the username `db2inst1` and the password for this user in the container. (You can also interact with the `project` database manually in the container to verify operation of GUI commands.)

• Once you have logged in, you will be presented with a tabbed pane, with tabs for the following kinds of operations:

- Checking out book(s) to a borrower
- Returning or renewing book(s)
- Maintaining the various entities that appeared in the ER diagram (Books, Categories, Borrowers, and Fines) - note that the structure of the GUI is based on the E-R diagram, so the Book entity will actually involve several tables in the database.
- Producing reports

Most of these panes will require the user to enter one or two values - e.g. the borrower id of the borrower to check books out to, or the key of one of the entities to be modified. In some cases, clicking a button in one of the panes will bring the user to a separate screen for entering/ presenting additional information. For example, clicking the Add button in the Categories pane will bring the user to a screen with slots for the user to fill in the attributes of the new Category to be added; clicking the Edit button in the Categories pane will bring the user to a screen with the current values displayed in editable fields, and clicking on the Delete button in the Categories pane will bring the user to a screen with current values displayed in non-editable fields, asking the user to confirm that he/she really intends to delete this Category.

You will probably find it worthwhile to experiment with the GUI without being connected to a database to get a feel for how it works. You should also experiment with the pre-written code for Categories while connected to the database.

Where multiple values are possible for an item (e.g. phone numbers for a borrower), the GUI will contain a combo box with any existing values included plus a slot labelled `(new value)`. To insert a new item, you can replace `(new value)` with a new value, and then press Return. When you do this, another `(new value)` slot will appear at the end of the list. You can edit any existing item. You can delete an existing item by clearing out its slot.

It is <u>extremely important</u> that you study and <u>understand</u> the Database code already written for you before proceeding!

APPENDIX I: IMPLEMENTATION NOTES

1. I have already written some initial database creation code, the GUI and quite a bit of additional code for you to use in this project. Your only task will be to add constraints/ triggers to the database creation code and finish the embedded SQL in `Database.java` to access/modify the database.

2. All of your embedded SQL / Java coding for this project will be done in the file defining the `Database` class (`Database.sqlj`). Much of this code has been written for you, but there is enough left for you to write so you won't be bored :-) [Be sure to keep it in the `semesterproject` folder].

3. The file `project.jar` contains all the remaining code (for the main program and the GUI) in compiled form in a java `jar` file. (You may ask to see the source code if you wish.)

4. The file `Fine.html` contains the javadoc documentation for the class `Fine`, which your code will need to use.

5. You will find some code for testing/demonstrating the GUI incorporated in the `Database.sqlj` file. (Typically, this code supplies "hardwired" values instead of accessing the database, or writes a message to `System.out` instead of modifying the database.) This code is recognizable because it is labeled "STUB", and is not indented (i.e. each line starts at the left margin.) You must remove all this STUB code from `Database.sqlj` and replace it with real code of your own.

   Be very careful to delete all the STUB code, but not to delete any real (i.e. indented) code that has been written for you.

6. You will probably find it easiest to start with the operations that maintain database entities (add, edit, delete.) Of course, you can use interactive SQL to manually check the correctness of these operations and/or to insert data into the database for testing these operations.

7. When a SQL statement fails due to the violation of some constraint or a signal from a trigger, a `SQLException` is thrown. Thus, all embedded SQL statements must be "wrapped" in a `try..catch` block. Where the code anticipates that an exception might be thrown (e.g. accessing nonexistent data or performing some operation that violates a constraint), the exception handler should test the exception object to determine what went wrong and produce a suitable descriptive error message. The "unexpected SQL error" case should be reserved for handling exceptions that could not be anticipated (typically due to some sort of severe system error), since the resultant message will not be "user friendly".

   Unfortunately, determining the exact cause of an exception is a bit complex, because the cause is found in different ways for different exceptions. All `SQLException` objects have three methods that can be used to provide information about them:

   `getErrorCode()` returns a vendor-specific numeric code.
   `getSQLState()` returns a string which also represents a code for the error.
   `getMessage()` returns a descriptive message (which is usually not particularly "friendly").

   When an exception is thrown, the cause can be determined by examining one or more of these values. There are examples of each in the code you were given, but of course you will have to also write your own code to test for similar cases elsewhere.

• If the exception is thrown as the result of violating a primary key or unique constraint, the error code returned by `getErrorCode()` will be -803, and the name of the table will occur in the message returned by `getMessage()`. For an example of testing for this and creating a more friendly message for the user, see the code you were given for `addBook()`. Note how this code makes use of the symbolic constant `DUPLICATE_KEY_SQL_ERROR`, which is defined at the end of the file.

• If the exception is thrown as the result of violating a foreign key or check constraint, and the constraint has a name, the table name and constraint name will appear in the message returned by `getMessage()`. For an example of testing for this and creating a more friendly message for the user, see the code you were given for `addBook()`. **To make this code work, you need to fill in the appropriate symbolic constant definitions using the names you gave the constraints in your database creation code!**

• If the exception is thrown as a result of executing a singleton select and no matching rows were found, the SQL state returned by `getSQLState()` will be `"02000"`. (Note that this is a String). For an example of testing for this and creating a more friendly message for the user, see the code you were given for `getBookInformation()`. Note how this code makes use of the the symbolic constant `NO_ROW_SQL_STATE`, which is defined at the end of the file.

• If the exception is thrown as the result of executing a SQL `signal` statement , the `signal` statement will specify a SQL state that can be tested for as discussed above.

8. The add and edit screens for the various entities handle multivalued attributes by using a GUI drop-down list, receiving/returning a (possibly empty) `Java.util.List` of Strings representing the contents of this drop-down. As noted above, in the GUI, you can add, edit or delete items in this list.

   In the case of the edit operation, the method you write is called with two parameters for each such attribute - one the original list of values and one the updated list - but you never actually update a row in a table. If you use the method suggested below, you will not actually make use of the original items list. Note how this is handled in the `updateBook()` method written for you.

9. Because your program has to deal with issues like overdue books, realistic testing could require months. To get around this difficulty, the database creation code supplied for you defines a database variable called `today` which represents the program's notion of the date to allow pretending to run the program on some other day. The `Diddle with Date` option in the `Test` menu can be used to modify this value, Using this facility, you can roll the program's notion of "today" forward or backward.

   The following excerpts illustrate how you might incorporate a reference to this variable into a SQL statement appearing in your code:

   • Calculate due date for a book - given `checkout_period` from `Category` table
   ```
   today + checkout_period days
   ```

   • Calculate days a book is overdue - given `date_due` from Checkout table (<= 0 means not overdue)
   ```
   days(today) - days(date_due)
   ```

   (In a production version, all references to `today` would be changed to `current date`, of course)

   All computations involving dates must be done by SQL code in the database. Do not, under any circumstances, attempt to do any date computations in Java!

10. The GUI given to you includes a `Test` menu to facilitate testing. (This would not be present in a production version of the program, of course.). It includes the following options:

    a. A `Diddle with Date` option as discussed above
    b. An "`Automatic Test`" option . This reads a series of commands from a text file (chosen from a FileChooser dialog) and executes them as if you had chosen them from the GUI. See Appendix J for information about the format of this file.
    c. A option that allows you to delete all the data currently in the database to make a clean start in testing. (The tables themselves are left unchanged - just their contents are deleted)

To the maximum extent possible, all computations must be done by the DBMS - not the program. In cases where a requirement can be met either by an appropriate constraint trigger or by writing Java code, the constraint / trigger must be used. That means you should be very suspicious of "`if`" occurring in your code. There will be places you will need to use "`if`". Most (but not all) of these will be in code deciding what message to print, but be sure you're not using it to test something the database can test using a constraint or trigger.
Most operations should be done by a single SQL query, possibly followed by a commit.

(The **only** exceptions are checking out/returning/renewing books where a select is needed to get the due date, and adding or editing books or borrowers, due to multiple tables being required by 4NF)

Each report must be produced by a single SQL query (it can be done). The `produceReport()` method furnished should be used to actually print the report. Before writing your own code, you will want to carefully study the examples done for you in the code given you!

DB2 will throw a "`truncation error`" exception if a `String` parameter to some SQL statement is longer than the width of the corresponding column. There is a way to prevent this error from occurring, but it involves some complexity. You do not need to check for such errors; simply assume that all data the user will never type a data value that is too long for the corresponding column (and be sure you comply with this while testing).

Please plan your work on this project on the assumption that there will be issues requiring my attention that arise as you are doing it. Each time I have assigned this project I have made significant changes, so expect that questions will arise! Be sure to allow enough time for me to figure out what's happening!

APPENDIX J: AUTOMATIC TESTING OF CPS352 PROGRAMMING PROJECT

As noted above, the GUI you have been furnished includes a `Test` menu containing an `Automatic Test` option which will execute commands read from a text file chosen via a JFileChooser dialog. You can also run a test file from the command line - without using the GUI - if you start the program with the command `java -jar project.jar` *testfilename*. The format of this file is as follows:

Each test is represented on a single line of the file, with the first character specifying the test to be performed, and subsequent content being additional data, as described below

= Fresh start; delete all items currently in the database and reset date to today.

C = Check out a book
R = Return a book
W = Renew a book

Remainder of line contains borrower ID (for checkout), then call number and copy number

A = Add an entity
E = Edit an entity
D = Delete an entity

Entity is one of the following

K = book
C = category
R = borrower

Remainder of line contains appropriate information - all fields for add, primary key plus all modifiable fields for edit, primary key for delete (note: for book, primary key is just call number for edit; call number + copy number for delete)

If an attribute contains spaces (e.g. a title or to test a keyword containing spaces), the value should be enclosed in quotes (""). In the case of an attribute that potentially takes multiple values, the list of values must be enclosed in braces (`{ }`), with a space after the opening brace and before the closing brace. (An empty list (`{ }`) is permissible).

G = Get fine information.

Remainder of the line contains the borrower ID. The report will show all of the fines recorded for this borrower, one per line - or an exception will be reported if there are none.

F = Pay a fine.

Remainder of the line contains the borrower ID, title, and an integer representing the difference between the due date and the date the test is being run..

P = Print a report

Remainder of the line is a report specifier - one of the following:

B = book
b = book copies (note lower case)
K = keywords
C = categories
R = borrowers
F = fines
O = overdue books

+ = increment "today"

Remainder of line contains number of days to add/subtract to database notion of today

! = start comment line - line is logged to standard output but ignored

The following is a sample of a simple test data file illustrating each kind of command. This has been included as the file `test.dat` in the starter file you have been given. Note that if adds it own data to the database - so it is meant to start out with an empty database. **This is just to illustrate the format of the file - it is far from a complete test of the program! You will need to include a comprehensive tester in your final submission!**

```
! Example of a testing file.   THESE ARE FAR FROM SUFFICIENT TESTS!
! STUDENTS: BE SURE THE FILE YOU TURN IN TESTS EVERYTHING; PUT YOUR NAME IN IT
!
! Delete pre-existing data
#
! Examples of add tests
! Two copies of the same book, one category, one borrower
AK AA.00 "LEGITIMATE BOOK" { AARDVARK ZEBRA } HC { ORIGINAL WORDS }
AK AA.00
AC PEON 14 100
AR 1234 AARDVARK ANTHONY { 123-4567 X1234 } PEON
! Example of an illegal add - a book with spaces in a keyword
AK AB.00 "SPACES IN KEYWORD" { } HC { "SPACE IN KW" }
! Examples of checkout, renewal requests
C 1234 AA.00 1
C 1234 AA.00 2
W AA.00 1
! Example of date modification and returning a book late
+ 30
R AA.00 2
! Examples of fine tests.
G 1234
! 14 used for F because due date is 14 days after start
F 1234 "LEGITIMATE BOOK" 14
G 1234
! Examples of edit tests
EK AA.00 "EDITED LEGITIMATE BOOK" { NEWAUTHOR AARDVARK } SC { WORDS EDITED }
EC PEON 5 2
ER 1234 ZEBRA ZELDA { X1234 987-6543 } PEON
! Examples of reports
PB
Pb
PK
PC
PR
PF
PO
! Examples of delete tests - copy 1 of book, category and borrower deleted
DK AA.00 1
DR 1234
DC PEON
```

APPENDIX K: RUBRIC THAT WILL BE USED FOR THE EVALUATION OF PART II

CPS352 - DATABASE SYSTEMS

Evaluation of Programming Project Part II - _____

_____

Key: M = Missing; GD = Grave Deficiencies; SD = Serious Deficiencies;
MD = Moderate Deficiencies; MC = Mostly Correct; ATC = (Almost) Totally Correct

Item	M	GD	SD	MD	MC	ATC
Milestone submitted on time and essentially correct	0	1	2	3	4	5
Code in Database.sqlj is appropriately commented, indented, reasonably efficient, etc.	0	1	2	3	4	5
Test plan covers all project requirements	0	1	2	3	4	5
Test cases thoroughly cover possible errors	0	1	2	3	4	5
Test report documents thorough testing of project	0	1	2	3	4	5
Correct operation (from next page)				_____		
TOTAL FOR PART II (MAX 100)				**_____**		

For Correct Operation See Next Page

The number in parentheses is max points for correct operation

- For intentional errors, 1/2 credit if an exception is thrown but message is a generic SQL error message or indicates the wrong error.
- For given items 1 point off for "breaking" it

OVERALL PROJECT EVALUATION

Part I: _____ x 30% = _____

Part II: _____ x 70% = _____

TOTAL FOR PROJECT; _____

Legitimate Adds	
Book - first for its call number	given
Book - another copy of existing	given
Category	given
Borrower (2)	2
Illegal Adds	
Book - wrong format	1
Book - duplicate author	1
Book - keyword with spaces	1
Book - duplicate keyword	1
Category - duplicate name	given
Borrower - duplicate ID	1
Borrower - duplicate phone	1
Borrower - undefined category	1
Legitimate Edits	
Book	given
Category	given
Borrower	2
Illegal Edits	
Book - no such	given
Book - wrong format	1
Book - duplicate author	1
Book - keyword with spaces	1
Book - duplicate keyword	1
Category - no such	given
Borrower - no such	1
Borrower - duplicate phone	1
Borrower - undefined category	1
Legitimate Deletes	
Book (2) [more copies same book]	2
Book (4) [last copy of book]	1
Book (1) [w/checked out cascade]	1
Book (1) [Fine on it stays]	1
Category	given
Borrower (2) [w/Fine cascade]	2

Points for illegal operations require appropriate message

Illegal Deletes	
Book - no such	given
Category - no such	given
Category - borrowers still in it	1
Borrower - no such	1
Borrower - books checked out	1
Checkouts	
Legitimate	2
No such borrower	1
No such book	1
Book already out to someone else	1
Too many books for borrower	4
Returns	
Legitimate	2
Book is not out	1
Overdue is fined	4
Renewals	
Legitimate	2
Book is not out	1
Overdue not allowed	given
Fines	
Details recorded correctly	2
Pay Legitimate	2
Get - No such borrower	2
Get - Borrower has none	2
Reports	
Book information	given
Book copies	4
Book keywords	4
Categories	given
Borrowers	4
Fines	4
Overdue Books	4

Points for Operation (Max 75)_____