CSCE 747 - Inspection Activity
Name(s):

You are inspecting the source code for the Graduate Record and Data System (GRADS). A system that graduate students can log into and use to view their transcript or a summary of their progress towards graduation. Your current task is to inspect the class “Session” - a class used to track information about a user of the system, as well as to store the contents of databases in memory. You have been provided with a checklist of common Java code style issues, and are to inspect the Session class against that list.

The checklist and program are attached. Working in pairs, document below which checklist items were not met, and why they were not met. Provide advice on how to address that shortcoming.

Sample Checklist for Java Programs

- File Header
  - Are the following included and consistent?
    - Author and current maintainer.
- File Footer
  - Is there a revision log to minimum of one year or most recent point release?
- Import Section
  - Is there a brief comment on each import with the exception of standard java.io.* or java.util.*?
- Class Declaration
  - Is the constructor explicit?
  - Is the class protected or private?
    - If not, is there justification for public access?
  - Does the JavaDoc header include:
    - A one sentence summary of class functionality?
    - Usage instructions?
- Class
  - Are names compliant with the following rules?
    - Class or interface: CapitalizedWithEachWord
    - Exception: ClassNameEndsWithException
    - Constants (final): ALL_CAPS_UNDERSCORES
    - Field Name: capsAfterFirstWord
      - Must be meaningful outside of context.
- Methods
  - Are names compliant with the following rules?
    - Method name: capsAfterFirstWord
    - Local variables: capsAfterFirstWord
      - Names may be short (e.g., i for integer) if scope of declaration and use is less than 30 lines.
    - Factory method for X: newX
    - Converter to X: toX
    - Getter for attribute X: getX();
    - Setter for attribute X: void setX;
  - Are method semantics consistent with similarly named methods?
    - (put(O) matches put(O) use for other classes)
  - Are usage examples provided for nontrivial methods?
- Fields
  - Is the field necessary (cannot be a local variable)?
  - Is the field protected or private?
    - If not, is there justification for public access?
  - Are there comments describing the purpose of the field?
  - Are there any constraints or invariants documented in the field or class comment header?

Class: Session

```
/**
 * Session
 * Author: Gregory Gay and Jason Biatek
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 *
 * - Redistributions of source code must retain the above copyright notice, this
 *   list of conditions and the following disclaimer.
 * - Redistributions in binary form must reproduce the above copyright notice,
 *   this list of conditions and the following disclaimer in the documentation
 *   and/or other materials provided with the distribution.
 * - Neither the name of the University of Minnesota nor the names of its
 *   contributors may be used to endorse or promote products derived from this
 *   software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

package edu.umn.csce5801;

import java.util.ArrayList;

// Object that represents a course.
import edu.umn.csci5801.model.Course;
import edu.umn.csci5801.model.CourseTaken;
// Object that represents a progress summary.
import edu.umn.csci5801.model.ProgressSummary;
// Object that represents a student record.
import edu.umn.csci5801.model.StudentRecord;
//Object that represents a system user.
import edu.umn.csci5801.model.User;

/**
* Class Session
* Session-relevant information for the graduate record and data system.
* Instantiated by the system interface when a user logs in, then
* destroyed when they log out.
* Used to load contents of course, user, and records databases into local memory.
*/
public class Session {

        // Current user
        private User currentUser;
```

```
// List of loaded student records
private ArrayList<StudentRecord> records;
// Name of the records file.
private String recordsFile;
// ID of the requested item.
Private String userId;
// List of courses offered.
private ArrayList<Course> coursesOffered;
// List of users
private ArrayList<User> users;
// Progress summary builder
private ProgressSummaryBuilder builder = new ProgressSummaryBuilder();

// Standard set of getters and setters
protected User getUser(){
        return this.currentUser;
}

// Sets the user, requires that the user data file have been read,
// and that the user exists.
protected void setUser(User user) throws GRADSDataException{
        if(users.size() > 0){
                if(users.contains(user)){
                        currentUser = user;
                }else{
                        throw new GRADSDataException("User "+user.getId()+" does not exist.");
                }
        }else{
                throw new GRADSDataException("Users file has not been read.");
        }
}

// Sets the user, requires that the user data file have been read,
// and that the user exists.
protected void setUser(String userId) throws GRADSDataException{
        this.userId = userId;
        if(this.users.size() > 0){
                ArrayList<String> ids = this.getIDsUsers();

                if(ids.contains(userId)){
                        currentUser = users.get(ids.indexOf(userId));
                }else{
                        throw new GRADSDataException("User "+userId+" does not exist.");
                }
        }else{
                throw new GRADSDataException("Users file has not been read.");
        }
}

protected ArrayList<StudentRecord> getRecords(){
        return records;
}

protected void setRecords(ArrayList<StudentRecord> records){
        this.records = records;
}
```

```
// Need to keep track, as we might write changes back to this file.
protected String getRecordsFile(){
	return recordsFile;
}

protected void setRecordsFile(String recordsFile){
	this.recordsFile = recordsFile;
}

protected ArrayList<Course> getCoursesOffered(){
	return coursesOffered;
}

protected void setCoursesOffered(ArrayList<Course> coursesOffered){
	this.coursesOffered = coursesOffered;
}

protected ArrayList<User> getUsers(){
	return users;
}

protected void setUsers(ArrayList<User> users){
	this.users = users;
}

// Private function to get the set of user IDs.
// Private to protect data security.
private ArrayList<String> getIDsUsers(){
	ArrayList<String> ids = new ArrayList<String>();

	for(User user: users){
		ids.add(user.getId());
	}

	return ids;
}

// Private function to get the set of student IDs for which records exist.
// Private to protect data security.
private ArrayList<String> getIDsRecordsInternal(){
	ArrayList<String> ids = new ArrayList<String>();

	for(StudentRecord r: records){
		ids.add(r.getStudent().getId());
	}

	return ids;
}

// Public-facing function to get the set of student IDs for which records exist.
protected ArrayList<String> getIDsRecords() throws GRADSPermissionsException{
	if(currentUser.getRole().toString().equals("GRADUATE_PROGRAM_COORDINATOR")){
		if(currentUser.getDepartment().toString().equals("COMPUTER_SCIENCE")){
			ArrayList<String> ids = new ArrayList<String>();

			for(StudentRecord record: records){
				ids.add(record.getStudent().getId());
```

```
                        }

                        return ids;
                }else{
                        throw new GRADSPermissionsException("User "+
                                currentUser.getId()+" is not part of the CS department.");
                }
        }else{
                throw new GRADSPermissionsException("User "+
                        currentUser.getId()+" does not have permission to take this action.");
        }

}

// Retrieves the requested student record if the user is either the owner of
// that record, or is a graduate program coordinator.
protected StudentRecord getRecord(String userId) throws Exception{
        this.userId = userId;
        ArrayList<String> ids = this.getIDsRecordsInternal();

        if(ids.contains(userId)){
                if(currentUser.getRole().toString().equals("GRADUATE_PROGRAM_COORDINATOR") ||
                         currentUser.getId().equals(userId)){

                        if(currentUser.getDepartment().toString().equals(records.get(id
                        s.
                                indexOf(userId)).getDepartment().toString())){
                                return new StudentRecord
                                        (records.get(ids.indexOf(userId)));
                        }else{
                                throw new GRADSPermissionsException("User "
                                        +currentUser.getId()
                                        +" is not part of the correct department.");
                        }
                }else{
                        throw new GRADSPermissionsException("User
"+currentUser.getId()+
                                " does not have permission to access the transcript for
"
                                +userId);
                }
        }else{
                throw new GRADSDataException("User ID "+userId+" does not exist.");
        }
}

protected void updateRecord(String userId, StudentRecord record) throws Exception{
        this.userId = userId;
        ArrayList<String> ids = this.getIDsRecordsInternal();

        if(ids.contains(userId)){
                if(currentUser.getRole().toString().equals("GRADUATE_PROGRAM_COORDINATOR")){

                        if(currentUser.getDepartment().toString().equals(records.get(
                                ids.indexOf(userId)).getDepartment().toString())){

                                if(record.getStudent().getId().equals(userId)){
```

```
                                        records.set(ids.indexOf(userId), record);
                                }else{
                                        throw new GRADSDataException
                                                ("Not allowed to change user ID.");
                                }
                        }else{
                                throw new GRADSPermissionsException("User
"+currentUser.getId()+
                                        " is not part of the correct department.");
                        }
                }else{
                        throw new GRADSPermissionsException("User "+currentUser.getId()+
                                " does not have permission to edit the transcript for "
                                +userId);
                }
        }else{
                throw new GRADSDataException("User ID "+userId+" does not exist.");
        }
}

// Adds a note to a student record.
protected void addNote(String userId, String note) throws Exception{
        this.userId = userId;
        int index = this.getIDsRecordsInternal().indexOf(userId);
        StudentRecord student = new StudentRecord(records.get(index));

        ArrayList<String> notes = new ArrayList<String>(student.getNotes());
        notes.add(note);
        for (String n: notes){
                System.out.println(n);
        }

        student.setNotes(notes);
        records.set(index, student);
}

// Adds course areas from the course database to the student record
// This is done to improve maintenance costs. Only have to change course
// areas in one central datastore.
protected StudentRecord addAreasToRecord(StudentRecord record){
        StudentRecord toreturn = new StudentRecord(record);

        ArrayList<String> courseIds = new ArrayList<String>();
        for(Course course: coursesOffered){
                courseIds.add(course.getId());
        }

        ArrayList<CourseTaken> coursesTaken =
                new ArrayList<CourseTaken>(toReturn.getCoursesTaken());
        ArrayList<Integer> toRemove = new ArrayList<Integer>();

        for(int entry=0; entry< coursesTaken.size();entry++){
                CourseTaken course = coursesTaken.get(entry);

                // Is this a course that is still offered?
                if(courseIds.contains(course.getCourse().getId())){
                        // Does it have a course area
```

```
                                Course taken = course.getCourse();
                                taken.setCourseArea(coursesOffered.get(courseIds.indexOf
                                        (course.getCourse().getId())).getCourseArea());
                                course.setCourse(taken);
                                coursesTaken.set(entry, course);
                        }else if(course.getCourse().getId().contains("csci")){
                                toRemove.add(entry);
                        }
                }

                for(int entry: toRemove){
                        coursesTaken.remove(entry);
                }

                toreturn.setCoursesTaken(coursesTaken);
                return toreturn;
        }

        // Starts the process of building the progress summary for the requested student ID.
        protected ProgressSummary generateProgressSummary(String userId) throws Exception{
                ArrayList<String> ids = this.getIDsRecordsInternal();

                if(ids.contains(userId)){
                        if(currentUser.getRole().toString().equals("GRADUATE_PROGRAM_COORDINATOR")){

                                if(currentUser.getDepartment().toString().equals(
                                        records.get(ids.indexOf(userId)).getDepartment().toString())){
                                        StudentRecord toBuild = this.addAreasToRecord(records.get
                                                (ids.indexOf(userId)));
                                        builder.setRecord(toBuild);

                                        return builder.buildProgressSummary();
                                }else{
                                        throw new GRADSPermissionsException("User
"+currentUser.getId()+
                                                " is not part of the correct department.");
                                }
                        }else{
                                throw new GRADSPermissionsException("User "+currentUser.getId()+
                                " does not have permission to generate a progress summary for
"+userId);
                        }
                }else{
                        throw new GRADSDataException("User ID "+userId+" does not exist.");
                }
        }
/*
* Revision History:
* 09/03/2014: Added exception handling to control access.
* 09/02/2014: Fixed a fault regarding record storage.
* 09/01/2014: Initial file creation.
*/

}
```