CS 5004 Final Exam - Sunday exam, Fall 2021

- This is a **computer-based exam**. That means that you will want to use your own computer to **individually** work on the problems during the designated exam time, and you will want to submit your solutions by pushing them to your **individual repo on our GitHub course organization**. **Finally, you will want to post the final release link to your code on Canvas.**
- The exam will be opened between 12:01am PT on Sunday, December 12th until 11:59pm PDT on Monday, December 13th.
- During your chosen exam day, you are welcome to spend up to **six hours working on the exam problems**..
- When pushing your code to your repo, you will want to use our **course Gradle project** (the same Gradle project, and the corresponding build file that we have been using throughout this course).
- Additionally, we will use submission rules similar to those we followed for the homework assignments:
  - **Repository content:** Your repositories should contain only the `build.gradle` file and `src` folder with appropriate sub-folders with your code and tests.
  - **Gradle built:** Your project should successfully build using the course Gradle configuration, and it should generate all the default reports.
- Some additional expectations and restrictions:
  - **Naming convention:** Please follow the naming convention for classes, methods, variables, constants, interfaces and abstract classes that we have been following throughout this whole course (camel case). Please avoid **magic numbers**.
  - **Immutability:** working in the immutable Java world is not the requirement on this exam.
  - **Methods `hashCode(), equals(), toString()`:** your classes should provide appropriate implementations for methods: `boolean equals(Object o), int hashCode(), String toString()`. Appropriate means that it is sufficient to autogenerate these methods, as long as autogenerated methods suffice for your specific implementation.
  - **Testing your code:** tests are required **only if/where specifically asked for in the assignment**.
  - **Javadoc:** when asked, please include a short description of your class/method, as well as tags `@params` and `@returns` in your Javadoc documentations (code comments). Additionally, if your method throws an exception, please also include a tag `@throws` to indicate that.

Good Luck!

Question	Points	Score
1	30	
2	35	
3	35	
Total:	100	

Problem 1 *(30 pts)*

Please consider classes `Name.java` , `Email.java` and `EmailAccount.java` provided below.

```java
package P1;

import java.util.Objects;

/*
Class Name contains information about the first, middle and last name. All name parts
are encoded as Strings.
 */
public class Name {

  private String firstName;
  private String middleName;
  private String lastName;

  public Name(String firstName, String middleName, String lastName) {
    this.firstName = firstName;
    this.middleName = middleName;
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getMiddleName() {
    return middleName;
  }

  public String getLastName() {
    return lastName;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Name)) return false;
    Name name = (Name) o;
    return Objects.equals(getFirstName(), name.getFirstName()) &&
        Objects.equals(getMiddleName(), name.getMiddleName()) &&
        Objects.equals(getLastName(), name.getLastName());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getFirstName(), getMiddleName(), getLastName());
  }

  @Override
  public String toString() {
    return "Name{" +
        "firstName='" + firstName + '\'' +
        ", middleName='" + middleName + '\'' +
        ", lastName='" + lastName + '\'' +
        '}';
  }
}

package P1;

import java.time.LocalDate;
import java.util.List;
import java.util.Objects;

/*
```

```java
Class email contains information about an email - email address of a sender, email addresses of all recipients,
CC-ed recipients, as well as information whether or not someone was BCC-ed. Additionally, class contains
information about a subject of an email, and the date it was sent. Note: email address is stored as a String,
in a format: name@domain (e.g., t.bonaci@northeastern.edu)
 */
public class Email {

  private String sender;
  private List<String> recipients;
  private List<String> CCedRecipients;
  private Boolean BCCedRecipentsIncluded;
  private String subject;
  private LocalDate dateSent;

  public Email(String sender, List<String> recipients, List<String> CCedRecipients,
              Boolean BCCedRecipentsIncluded, String subject, LocalDate dateSent) {
    this.sender = sender;
    this.recipients = recipients;
    this.CCedRecipients = CCedRecipients;
    this.BCCedRecipentsIncluded = BCCedRecipentsIncluded;
    this.subject = subject;
    this.dateSent = dateSent;
  }

  public String getSender() {
    return sender;
  }

  public List<String> getRecipients() {
    return recipients;
  }

  public List<String> getCCedRecipients() {
    return CCedRecipients;
  }

  public Boolean getBCCedRecipentsIncluded() {
    return BCCedRecipentsIncluded;
  }

  public String getSubject() {
    return subject;
  }

  public LocalDate getDateSent() {
    return dateSent;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Email)) return false;
    Email email = (Email) o;
    return Objects.equals(getSender(), email.getSender()) &&
        Objects.equals(getRecipients(), email.getRecipients()) &&
        Objects.equals(getCCedRecipients(), email.getCCedRecipients()) &&
        Objects.equals(getBCCedRecipentsIncluded(), email.getBCCedRecipentsIncluded()) &&
        Objects.equals(getSubject(), email.getSubject()) &&
        Objects.equals(getDateSent(), email.getDateSent());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getSender(), getRecipients(), getCCedRecipients(),
        getBCCedRecipentsIncluded(), getSubject(), getDateSent());
  }
  @Override
  public String toString() {
    return "Email{" +
        "sender='" + sender + '\'' +
        ", recipients=" + recipients +
        ", CCedRecipients=" + CCedRecipients +
        ", BCCedRecipentsIncluded=" + BCCedRecipentsIncluded +
        ", subject='" + subject + '\'' +
        ", dateSent=" + dateSent +
        '}';
  }
}

package P1;

import java.util.List;
import java.util.Objects;

/*
Class EmailAccount contains information about an owner of a specific email account (owner's name), email address
of the account, as well as information about all the emails sent and received by that email account.
 */
public class EmailAccount {
  private Name owner;
  private String emailAddress;
  private List<Email> receivedEmails;
  private List<Email> sentEmails;

  public EmailAccount(Name owner, String emailAddress, List<Email> receivedEmails, List<Email> sentEmails) {
    this.owner = owner;
    this.emailAddress = emailAddress;
    this.receivedEmails = receivedEmails;
    this.sentEmails = sentEmails;
  }

  public Name getOwner() {
    return owner;
  }

  public void setOwner(Name owner) {
    this.owner = owner;
  }

  public String getEmailAddress() {
    return emailAddress;
  }

  public void setEmailAddress(String emailAddress) {
    this.emailAddress = emailAddress;
  }

  public List<Email> getReceivedEmails() {
    return receivedEmails;
  }

  public void setReceivedEmails(List<Email> receivedEmails) {
    this.receivedEmails = receivedEmails;
  }

  public List<Email> getSentEmails() {
    return sentEmails;
  }

  public void setSentEmails(List<Email> sentEmails) {
    this.sentEmails = sentEmails;
  }
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof EmailAccount)) return false;
    EmailAccount that = (EmailAccount) o;
    return Objects.equals(getOwner(), that.getOwner()) &&
        Objects.equals(getEmailAddress(), that.getEmailAddress()) &&
        Objects.equals(getReceivedEmails(), that.getReceivedEmails()) &&
        Objects.equals(getSentEmails(), that.getSentEmails());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getOwner(), getEmailAddress(), getReceivedEmails(), getSentEmails());
  }

  @Override
  public String toString() {
    return "EmailAccount{" +
        "owner=" + owner +
        ", emailAddress='" + emailAddress + '\'' +
        ", receivedEmails=" + receivedEmails +
        ", sentEmails=" + sentEmails +
        '}';
  }
}
```

Now consider class `EmailProcessor.java`, also provided below.

```java
package P1;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

/*
Class EmailProcessor provides functionality related to processing of a specific email account. For a given email
account, it allows searching and filtering emails by a sender, by a recipients, by subject and by date sent.
 */
public class EmailProcessor {

  private EmailAccount emailAccount;
  private static final Integer RECIPIENTS_CUTOFF = 5;

  public EmailProcessor(EmailAccount emailAccount) {
    this.emailAccount = emailAccount;
  }

  public EmailAccount getEmailAccount() {
    return emailAccount;
  }

  public List<String> mysteryMethod(LocalDate date){

    return this.emailAccount.getReceivedEmails().stream().filter(x -> x.getBCCedRecipentsIncluded())
                            .filter(y -> y.getRecipients().size() >= RECIPIENTS_CUTOFF)
                            .filter(z -> z.getDateSent().equals(date))
                            .map(w -> w.getSubject()).collect(Collectors.toList());
  }

  public List<Email> filterSentEmailsBySubjectAndDate(String subject, LocalDate date){
    //YOUR WORK HERE
    return null;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof EmailProcessor)) return false;
    EmailProcessor that = (EmailProcessor) o;
    return Objects.equals(getEmailAccount(), that.getEmailAccount());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getEmailAccount());
  }

  @Override
  public String toString() {
    return "EmailProcessor{" +
        "emailAccount=" + emailAccount +
        '}';
  }
}
```

(a) Please generate the appropriate Javadoc and unit tests for method `mysteryMethod(LocalDate date)`. ***(15 pts)***

(b) Please consider classes `Email.java`, `EmailAccount` and `EmailProcessor.java` again. ***(15 pts)***
In class `EmailProcessor.java`, you will notice method `filterSentEmailsBySubjectAndDate(String subject, LocalDate date)`.

The goal of this method is to return a list of all emails with subject equal to the given input argument `String subject`, and sent on date, provided as input argument `LocalDate date`.

Please provide code, Javadoc and unit tests for method

`filterSentEmailsBySubjectAndDate(String subject, LocalDate date)`. In doing so, please feel free to develop any helper methods that you might need within classes `EmailProcessor.java`, or `EmailAccount`, but please do not modify class `Email`.

Note: This problem is intended to be solved using elements of functional Java. All other correct solutions, that do not involve elements of functional programming in Java (e.g., streams, lambdas) will get a maximum of 7 points on this part.

Problem 2 *(35 pts)*

Please consider classes `Name.java`, `Course.java` and `Student.java` provided below.

```
package P2;

import java.util.Objects;

/*
Class Name contains information about the first, middle and last name. All name parts
are encoded as Strings.
 */
public class Name {

  private String firstName;
  private String middleName;
  private String lastName;

  public Name(String firstName, String middleName, String lastName) {
    this.firstName = firstName;
    this.middleName = middleName;
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getMiddleName() {
    return middleName;
  }

  public String getLastName() {
    return lastName;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Name)) return false;
    Name name = (Name) o;
    return Objects.equals(getFirstName(), name.getFirstName()) &&
        Objects.equals(getMiddleName(), name.getMiddleName()) &&
        Objects.equals(getLastName(), name.getLastName());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getFirstName(), getMiddleName(), getLastName());
  }

  @Override
  public String toString() {
    return "Name{" +
        "firstName='" + firstName + '\'' +
        ", middleName='" + middleName + '\'' +
        ", lastName='" + lastName + '\'' +
        '}';
  }
}

package P2;
import java.time.LocalDate;
import java.util.Objects;

/*
Class Course contains information about a course (course name and course code), as well as information
about which year has a student taken the course, and which grade have they gotten.
```

```
 */
public class Course {

  private String courseName;
  private String courseCode;
  private LocalDate yearTaken;
  private Integer grade;

  public Course(String courseName, String courseCode, LocalDate yearTaken, Integer grade) {
    this.courseName = courseName;
    this.courseCode = courseCode;
    this.yearTaken = yearTaken;
    this.grade = grade;
  }

  public String getCourseName() {
    return courseName;
  }

  public String getCourseCode() {
    return courseCode;
  }

  public LocalDate getYearTaken() {
    return yearTaken;
  }

  public Integer getGrade() {
    return grade;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Course)) return false;
    Course course = (Course) o;
    return Objects.equals(getCourseName(), course.getCourseName()) &&
        Objects.equals(getCourseCode(), course.getCourseCode()) &&
        Objects.equals(getYearTaken(), course.getYearTaken()) &&
        Objects.equals(getGrade(), course.getGrade());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getCourseName(), getCourseCode(), getYearTaken(), getGrade());
  }

  @Override
  public String toString() {
    return "Course{" +
        "courseName='" + courseName + '\'' +
        ", courseCode='" + courseCode + '\'' +
        ", yearTaken=" + yearTaken +
        ", grade=" + grade +
        '}';
  }
}

package P2;

import java.util.List;
import java.util.Objects;

/*
Class Student contains information about a student - students name and their studentID, academic program a student is
enrolled into, as well as the list of courses the student has taken.
 */
```

```java
public class Student implements Comparable {

  private Name name;
  private String studentID;
  private String academicProgram;
  private List<Course> takenCourses;

  public Student(Name name, String studentID, String academicProgram, List<Course> takenCourses) {
    this.name = name;
    this.studentID = studentID;
    this.academicProgram = academicProgram;
    this.takenCourses = takenCourses;
  }

  public Name getName() {
    return name;
  }

  public String getStudentID() {
    return studentID;
  }

  public String getAcademicProgram() {
    return academicProgram;
  }

  public List<Course> getTakenCourses() {
    return takenCourses;
  }

  public Float getGPA(){
    Integer gradeSum = 0;
    for (Course course : this.takenCourses){
      gradeSum += course.getGrade();
    }
    return Float.valueOf(gradeSum/this.takenCourses.size());
  }

  /**
   * Compares this object with the specified object for order.  Returns a
   * negative integer, zero, or a positive integer as this object is less
   * than, equal to, or greater than the specified object.
   *
   * <p>The implementor must ensure
   * {@code sgn(x.compareTo(y)) == -sgn(y.compareTo(x))}
   * for all {@code x} and {@code y}.  (This
   * implies that {@code x.compareTo(y)} must throw an exception iff
   * {@code y.compareTo(x)} throws an exception.)
   *
   * <p>The implementor must also ensure that the relation is transitive:
   * {@code (x.compareTo(y) > 0 && y.compareTo(z) > 0)} implies
   * {@code x.compareTo(z) > 0}.
   *
   * <p>Finally, the implementor must ensure that {@code x.compareTo(y)==0}
   * implies that {@code sgn(x.compareTo(z)) == sgn(y.compareTo(z))}, for
   * all {@code z}.
   *
   * <p>It is strongly recommended, but <i>not</i> strictly required that
   * {@code (x.compareTo(y)==0) == (x.equals(y))}.  Generally speaking, any
   * class that implements the {@code Comparable} interface and violates
   * this condition should clearly indicate this fact.  The recommended
   * language is "Note: this class has a natural ordering that is
   * inconsistent with equals."
   *
   * <p>In the foregoing description, the notation
   * {@code sgn(}<i>expression</i>{@code )} designates the mathematical
```

```
    * <i>signum</i> function, which is defined to return one of {@code -1},
    * {@code 0}, or {@code 1} according to whether the value of
    * <i>expression</i> is negative, zero, or positive, respectively.
    *
    * @param o the object to be compared.
    * @return a negative integer, zero, or a positive integer as this object
    * is less than, equal to, or greater than the specified object.
    * @throws NullPointerException if the specified object is null
    * @throws ClassCastException   if the specified object's type prevents it
    *                              from being compared to this object.
    */
   @Override
   public int compareTo(Object o) {

     //YOUR CODE HERE
     return 0;
   }

   @Override
   public boolean equals(Object o) {
     if (this == o) return true;
     if (!(o instanceof Student)) return false;
     Student student = (Student) o;
     return Objects.equals(getName(), student.getName()) &&
         Objects.equals(getStudentID(), student.getStudentID()) &&
         Objects.equals(getAcademicProgram(), student.getAcademicProgram()) &&
         Objects.equals(getTakenCourses(), student.getTakenCourses());
   }

   @Override
   public int hashCode() {
     return Objects.hash(getName(), getStudentID(), getAcademicProgram(), getTakenCourses());
   }

   @Override
   public String toString() {
     return "Student{" +
         "name=" + name +
         ", studentID='" + studentID + '\'' +
         ", academicProgram='" + academicProgram + '\'' +
         ", takenCourses=" + takenCourses +
         '}';
   }
 }
```

(a) Please notice that class `Student.java` implements interface `Comparable`. ***(5 pts)***

Implement method `compareTo(Student otherStudent)`, that compares two `Student` objects based upon their overall GPA (*note: please notice public method `Float getGPA()`* in class `Student`).

For example, let's assume there exist three students, `Student1`, `Student2` and `Student3`. Student1's GPA is 3.89, Student2's 3.45 and Student3's 3.92. Based upon our comparison criterion, `Student2` comes before `Student1`, and `Student1` comes before `Student3`, i.e., `Student2` < `Student1` < `Student3`. (increasing sort).

(b) Consider the given class `Student.java` again. In addition to comparing student based on their overall GPA, we would also like to compare students based upon the number of courses they have taken, and where they have gotten a grade of 3.3 or higher. We want to achieve this capability through the use of the interface `Comparator`. ***(15 pts)***

Please consider class `GradesFilteringComparator`:

```
package P2;

import java.util.Comparator;

public class GradesFilteringComparator implements Comparator<Student> {
  /**
   * Compares its two arguments for order.  Returns a negative integer,
   * zero, or a positive integer as the first argument is less than, equal
   * to, or greater than the second.<p>
   * <p>
   * The implementor must ensure that {@code sgn(compare(x, y)) ==
   * -sgn(compare(y, x))} for all {@code x} and {@code y}.  (This
   * implies that {@code compare(x, y)} must throw an exception if and only
   * if {@code compare(y, x)} throws an exception.)<p>
   * <p>
   * The implementor must also ensure that the relation is transitive:
   * {@code ((compare(x, y)>0) && (compare(y, z)>0))} implies
   * {@code compare(x, z)>0}.<p>
   * <p>
   * Finally, the implementor must ensure that {@code compare(x, y)==0}
   * implies that {@code sgn(compare(x, z))==sgn(compare(y, z))} for all
   * {@code z}.<p>
   * <p>
   * It is generally the case, but <i>not</i> strictly required that
   * {@code (compare(x, y)==0) == (x.equals(y))}.  Generally speaking,
   * any comparator that violates this condition should clearly indicate
   * this fact.  The recommended language is "Note: this comparator
   * imposes orderings that are inconsistent with equals."<p>
   * <p>
   * In the foregoing description, the notation
   * {@code sgn(}<i>expression</i>{@code )} designates the mathematical
   * <i>signum</i> function, which is defined to return one of {@code -1},
   * {@code 0}, or {@code 1} according to whether the value of
   * <i>expression</i> is negative, zero, or positive, respectively.
   *
   * @param o1 the first object to be compared.
   * @param o2 the second object to be compared.
   * @return a negative integer, zero, or a positive integer as the
   * first argument is less than, equal to, or greater than the
   * second.
   * @throws NullPointerException if an argument is null and this
   *                              comparator does not permit null arguments
   * @throws ClassCastException   if the arguments' types prevent them from
   *                              being compared by this comparator.
   */
  @Override
  public int compare(Student o1, Student o2) {

    //YOUR CODE HERE
    return 0;
  }
}
```

It contains method `compare(Student o1, Student o2)`. Please provide code, and Javadoc for method `compare()` which compares two students based upon the number of courses they have taken, and where they have gotten a grade 3.3 or higher.

In doing so, please feel free to develop any helper methods that you might need within class `GradesFilteringComparator.java`, but please do not modify classes `Student.java` or `Course.java`.

(c) Given a list of `Students`, ***(15 pts)***

`List<Student> students = new ArrayList<>();,`

where interface `List<E>` and concrete class `ArrayList` represent built-in implementations from the `java.util` **package**, implement an iterator `StudentIterator` that iterates over the given list of `Students`, and returns those students who have taken all four ALIGN courses. To find students who have taken all four ALIGN courses, please search for students who have taken courses with course codes `''CS 5001'', ''CS 5002'', ''CS5004'', ''CS 5008''`.

Note: You do not have to implement method `remove()`.

Problem 3 *(35 pts)*

(a) Please consider classes `Name.java`, `Course.java` and `Student.java` provided below. ***(5 pts)***

```java
package P3;

import java.util.Objects;

/*
Class Name contains information about the first, middle and last name. All name parts
are encoded as Strings.
 */
public class Name {

  private String firstName;
  private String middleName;
  private String lastName;

  public Name(String firstName, String middleName, String lastName) {
    this.firstName = firstName;
    this.middleName = middleName;
    this.lastName = lastName;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getMiddleName() {
    return middleName;
  }

  public String getLastName() {
    return lastName;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Name)) return false;
    Name name = (Name) o;
    return Objects.equals(getFirstName(), name.getFirstName()) &&
        Objects.equals(getMiddleName(), name.getMiddleName()) &&
        Objects.equals(getLastName(), name.getLastName());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getFirstName(), getMiddleName(), getLastName());
  }

  @Override
  public String toString() {
    return "Name{" +
        "firstName='" + firstName + '\'' +
        ", middleName='" + middleName + '\'' +
        ", lastName='" + lastName + '\'' +
        '}';
  }
}

package P3;
import java.time.LocalDate;
import java.util.Objects;

/*
Class Course contains information about a course (course name and course code), as well as information
```

```
about which year has a student taken the course, and which grade have they gotten.
 */
public class Course {

  private String courseName;
  private String courseCode;
  private LocalDate yearTaken;
  private Integer grade;

  public Course(String courseName, String courseCode, LocalDate yearTaken, Integer grade) {
    this.courseName = courseName;
    this.courseCode = courseCode;
    this.yearTaken = yearTaken;
    this.grade = grade;
  }

  public String getCourseName() {
    return courseName;
  }

  public String getCourseCode() {
    return courseCode;
  }

  public LocalDate getYearTaken() {
    return yearTaken;
  }

  public Integer getGrade() {
    return grade;
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Course)) return false;
    Course course = (Course) o;
    return Objects.equals(getCourseName(), course.getCourseName()) &&
        Objects.equals(getCourseCode(), course.getCourseCode()) &&
        Objects.equals(getYearTaken(), course.getYearTaken()) &&
        Objects.equals(getGrade(), course.getGrade());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getCourseName(), getCourseCode(), getYearTaken(), getGrade());
  }

  @Override
  public String toString() {
    return "Course{" +
        "courseName='" + courseName + '\'' +
        ", courseCode='" + courseCode + '\'' +
        ", yearTaken=" + yearTaken +
        ", grade=" + grade +
        '}';
  }
}

package P3;

import java.util.List;
import java.util.Objects;

/*
Class Student contains information about a student - students name and their studentID, academic program a student is
enrolled into, as well as the list of courses the student has taken.
```

```java
 */
public class Student {

  private Name name;
  private String studentID;
  private String academicProgram;
  private List<Course> takenCourses;

  public Student(Name name, String studentID, String academicProgram, List<Course> takenCourses) {
    this.name = name;
    this.studentID = studentID;
    this.academicProgram = academicProgram;
    this.takenCourses = takenCourses;
  }

  public Name getName() {
    return name;
  }

  public String getStudentID() {
    return studentID;
  }

  public String getAcademicProgram() {
    return academicProgram;
  }

  public List<Course> getTakenCourses() {
    return takenCourses;
  }

  public Float getGPA(){
    Integer gradeSum = 0;
    for (Course course : this.takenCourses){
      gradeSum += course.getGrade();
    }
    return Float.valueOf(gradeSum/this.takenCourses.size());
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Student)) return false;
    Student student = (Student) o;
    return Objects.equals(getName(), student.getName()) &&
        Objects.equals(getStudentID(), student.getStudentID()) &&
        Objects.equals(getAcademicProgram(), student.getAcademicProgram()) &&
        Objects.equals(getTakenCourses(), student.getTakenCourses());
  }

  @Override
  public int hashCode() {
    return Objects.hash(getName(), getStudentID(), getAcademicProgram(), getTakenCourses());
  }

  @Override
  public String toString() {
    return "Student{" +
        "name=" + name +
        ", studentID='" + studentID + '\'' +
        ", academicProgram='" + academicProgram + '\'' +
        ", takenCourses=" + takenCourses +
        '}';
  }
}
```

Now, design and implement a new class `CourseRecommender` that contains the following private fields:

- List of students,
- List of ALIGN students, and
- `Map<String, CourseStatistics>`, where data type `CourseStatistics` is a custom class that you will have to develop in subsequent parts of this problem.

Include two constructors for class `CourseRecommender`:

1. A constructor that receives input arguments `List<Student>, List<Student>, Map<String, CourseStatistics>`, and uses those input arguments to initialize private fields, and
2. A constructor that only receives one input argument, `List<Student>`, and it then uses its own helper methods to initialize the other two fields. The helper methods' signatures are defined below.

(b) Within class `CourseRecommender`, implement helper method ***(5 pts)***
`private List<Student> filterOutAlignStudents(List<Student> allStudents)`. This methods takes a list of all students, and returns a filtered list, containing only ALIGN students. For the purpose of this problem, ALIGN students are those students who have taken **any of the courses** with course code `"CS 5001", "CS 5002", "CS 5004"` or `"CS 5008`.

(c) Design and implement class `CourseStatistics`, which keeps track of the following information: ***(5 pts)***

- The total number of students who have taken some course, with course name `courseA`.
- The number of ALIGN students who have taken some course, with course name `courseA`.
- The average grade of all the students who have taken a course with course name `courseA`.
- The average grade of all the ALIGN students who have taken a course with course name `courseA`.

(d) Within class `CourseRecommender`, implement helper method ***(10 pts)***
`private Map<String, CourseStatistics> computeCollegeCoursesStatistics(List<Student>)`.
The given method takes a list of all students of some college, and from it creates a `Map<String, CourseStatistics>`, where map key is the unique course name, and map values are objects of the data type `CourseStatistics`.
In other words, the method takes a list of students of some college, and returns a map, containing mapped information for every course:

- How many students in total have taken every course?
- How many AIGN students have taken every course?
- What was the overall average grade in the course?
- What was the average grade in the course for the ALIGN students?

(e) Within class `CourseRecommender`, implement public method ***(10 pts)***
`public List<Course> recommendCourses(Student)`.
The given method takes an object `Student` as an input argument, and recommends three courses a student should consider registering for. The list of recommended courses is generated based upon the following criteria:

- The first recommended course is the first course taken by the most students in the college, that the student hasn't yet taken. For example, let's assume the course taken by the most students in the college is a course with code `"CS 5500"`, and the course taken by the second most students is a course with code `"CS 5465"`. If the student has already taken `"CS 5500"`, your program should recommend taking `"CS 5465"`.
- The second recommended courses is the course with the highest average grade that the student hasn't already taken. For example, let's assume that the course with the highest average grade has course code `"CS 7445"`, and the course with the second highest average grade has code `"CS 5800"`. If the student has already taken `"CS 7445"`, your program should recommend `"CS 5800"`.
- Please propose and implement your own recommendation criteria for the third recommended course.

A List

You can use `ArrayList` as a concrete type that implements the `List` interface.

```java
package java.util;

public interface List<E> extends Collection<E> {

    /**
     * Returns the number of elements in this list.
     */
    int size();

    /**
     * Returns <tt>true</tt> if this list contains no elements.
     */
    boolean isEmpty();

    /**
     * Returns <tt>true</tt> if this list contains the specified element.
     */
    boolean contains(Object o);

    /**
     * Returns an iterator over the elements in this list in proper sequence.
     */
    Iterator<E> iterator();

    /**
     * Appends the specified element to the end of this list (optional operation).
     */
    boolean add(E e);

    /**
     * Removes the first occurrence of the specified element from this list,
     * if it is present (optional operation).
     */
    boolean remove(Object o);

    /**
     * Returns <tt>true</tt> if this list contains all of the elements of the
     * specified collection.
     */
    boolean containsAll(Collection<?> c);

    /**
     * Removes all of the elements from this list (optional operation).
     */
    void clear();

    /**
     * Returns the element at the specified position in this list.
     */
    E get(int index);

    /**
     * Inserts the specified element at the specified position in this list
     */
    void add(int index, E element);

    /**
     * Removes the element at the specified position in this list
     */
    E remove(int index);
}
```

B Map

You can use `HashMap` as a concrete type that implements the `Map` interface.

```
package java.util;

public interface Map<K,V> {
    /**
     * Returns the number of key-value mappings in this map.  If the
     * map contains more than <tt>Integer.MAX_VALUE</tt> elements, returns
     * <tt>Integer.MAX_VALUE</tt>.
     */
    int size();

    /**
     * Returns <tt>true</tt> if this map contains no key-value mappings.
     */
    boolean isEmpty();

    /**
     * Returns <tt>true</tt> if this map contains a mapping for the specified
     * key.
     */
    boolean containsKey(Object key);

    /**
     * Returns <tt>true</tt> if this map maps one or more keys to the
     * specified value.
     */
    boolean containsValue(Object value);

    /**
     * Returns the value to which the specified key is mapped,
     * or {@code null} if this map contains no mapping for the key.
     */
    V get(Object key);

    /**
     * Associates the specified value with the specified key in this map
     */
    V put(K key, V value);

    /**
     * Removes the mapping for a key from this map if it is present
     */
    V remove(Object key);

    /**
     * Removes all of the mappings from this map (optional operation).
     */
    void clear();

    /**
     * Returns a {@link Set} view of the keys contained in this map.
     */
    Set<K> keySet();

    /**
     * Returns a {@link Collection} view of the values contained in this map.
     */
    Collection<V> values();

    /**
     * Returns a {@link Set} view of the mappings contained in this map.
     */
    Set<Map.Entry<K, V>> entrySet();
```

```java
    /**
     * A map entry (key-value pair).
     */
    interface Entry<K,V> {
        /**
         * Returns the key corresponding to this entry.
         */
        K getKey();

        /**
         * Returns the value corresponding to this entry.
         */
        V getValue();

        /**
         * Replaces the value corresponding to this entry with the specified
         * value.
         */
        V setValue(V value);

        /**
         * Compares the specified object with this entry for equality.
         */
        boolean equals(Object o);

        /**
         * Returns the hash code value for this map entry.
         */
        int hashCode();
    }

    /**
     * Compares the specified object with this map for equality.
     */
    boolean equals(Object o);

    /**
     * Returns the hash code value for this map.
     */
    int hashCode();
}
```

C Stream

```java
/**
 * A sequence of elements supporting sequential and parallel aggregate
 * operations.  The following example illustrates an aggregate operation using
 * {@link Stream} and {@link IntStream}:
 *
 * <pre>{@code
 *     int sum = widgets.stream()
 *                      .filter(w -> w.getColor() == RED)
 *                      .mapToInt(w -> w.getWeight())
 *                      .sum();
 * }</pre>
 *
 * In this example, {@code widgets} is a {@code Collection<Widget>}.  We create
 * a stream of {@code Widget} objects via {@link Collection#stream Collection.stream()},
 * filter it to produce a stream containing only the red widgets, and then
 * transform it into a stream of {@code int} values representing the weight of
 * each red widget. Then this stream is summed to produce a total weight.
 *
 */
public interface Stream<T> extends BaseStream<T, Stream<T>> {
```

```
/**
 * Returns a stream consisting of the elements of this stream that match
 * the given predicate.
 *
 * @param predicate a <a href="package-summary.html#NonInterference">non-interfering</a>,
 *                  <a href="package-summary.html#Statelessness">stateless</a>
 *                  predicate to apply to each element to determine if it
 *                  should be included
 * @return the new stream
 */
Stream<T> filter(Predicate<? super T> predicate);

/**
 * Returns a stream consisting of the results of applying the given
 * function to the elements of this stream.
 *
 * @param <R> The element type of the new stream
 * @param mapper a <a href="package-summary.html#NonInterference">non-interfering</a>,
 *               <a href="package-summary.html#Statelessness">stateless</a>
 *               function to apply to each element
 * @return the new stream
 */
<R> Stream<R> map(Function<? super T, ? extends R> mapper);

/**
 * Performs an action for each element of this stream.
 *
 * <p>The behavior of this operation is explicitly nondeterministic.
 * For parallel stream pipelines, this operation does <em>not</em>
 * guarantee to respect the encounter order of the stream, as doing so
 * would sacrifice the benefit of parallelism.  For any given element, the
 * action may be performed at whatever time and in whatever thread the
 * library chooses.  If the action accesses shared state, it is
 * responsible for providing the required synchronization.
 *
 * @param action a <a href="package-summary.html#NonInterference">
 *               non-interfering</a> action to perform on the elements
 */
void forEach(Consumer<? super T> action);

/**
 * Returns an array containing the elements of this stream.
 *
 * @return an array containing the elements of this stream
 */
Object[] toArray();

 /**
 * Performs a <a href="package-summary.html#Reduction">reduction</a> on the
 * elements of this stream, using the provided identity value and an
 * <a href="package-summary.html#Associativity">associative</a>
 * accumulation function, and returns the reduced value.  This is equivalent
 * to:
 * <pre>{@code
 *     T result = identity;
 *     for (T element : this stream)
 *         result = accumulator.apply(result, element)
 *     return result;
 * }</pre>
 *
 * but is not constrained to execute sequentially.
 *
 * <p>The {@code identity} value must be an identity for the accumulator
 * function. This means that for all {@code t},
 * {@code accumulator.apply(identity, t)} is equal to {@code t}.
```

```
     * The {@code accumulator} function must be an
     * <a href="package-summary.html#Associativity">associative</a> function.
     *
     * <p>This is a <a href="package-summary.html#StreamOps">terminal
     * operation</a>.
     *
     * @apiNote Sum, min, max, average, and string concatenation are all special
     * cases of reduction. Summing a stream of numbers can be expressed as:
     *
     * <pre>{@code
     *     Integer sum = integers.reduce(0, (a, b) -> a+b);
     * }</pre>
     *
     * or:
     *
     * <pre>{@code
     *     Integer sum = integers.reduce(0, Integer::sum);
     * }</pre>
     *
     * <p>While this may seem a more roundabout way to perform an aggregation
     * compared to simply mutating a running total in a loop, reduction
     * operations parallelize more gracefully, without needing additional
     * synchronization and with greatly reduced risk of data races.
     *
     * @param identity the identity value for the accumulating function
     * @param accumulator an <a href="package-summary.html#Associativity">associative</a>,
     *                    <a href="package-summary.html#NonInterference">non-interfering</a>,
     *                    <a href="package-summary.html#Statelessness">stateless</a>
     *                    function for combining two values
     * @return the result of the reduction
     */
    T reduce(T identity, BinaryOperator<T> accumulator);
}
```