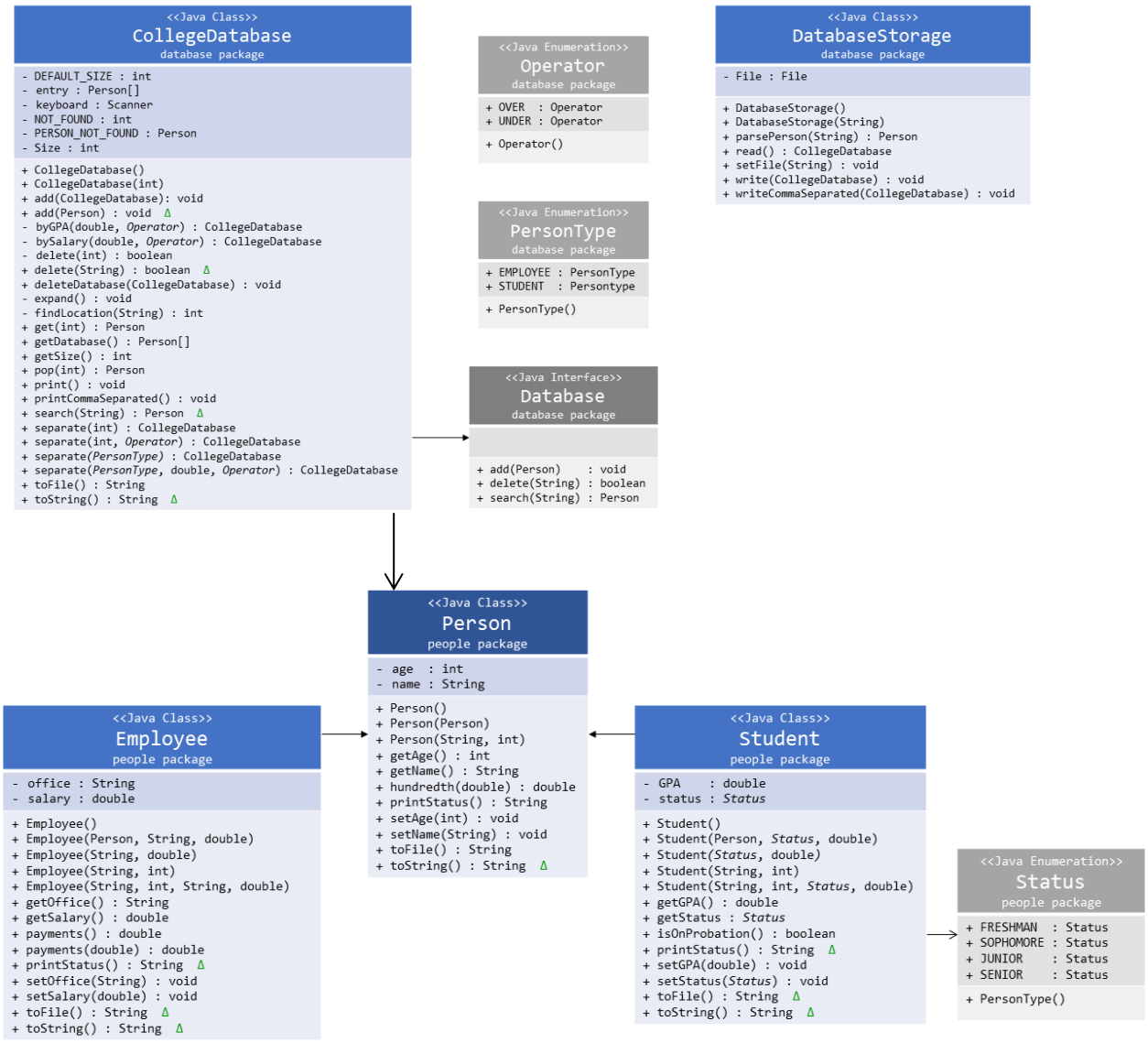
Keziah May
Fall 2018
CSC142
Final Project Report

Report: FinalProject

UML diagram of project:



Keziah May
Fall 2018
CSC142
Final Project Report

The purpose:

The purpose of this project is to take information read from a file, store it, and manipulate it in useful ways. The information in the files should represent people affiliated with a college, students or employees specifically. The information about each person should be represented on a single line with the individual data separated by commas. For students, you should provide their relationship to the college (student), their name, age, college status (Freshman, Sophomore, etc), and their GPA respectively. For Employees, provide their relationship to the college (Employee), their name, age, office location, and salary respectively. This information is all stored in a database. This data can be searched, separated into more specific databases (by relationship to college, GPA, salary, etc).

Applications:

This project could be paired with an online submission forum (or any other input or data collection software) if it could write submission data to a file in comma separated format as described above. It would allow data to be gathered about people affiliated with the college to perform statistic assessments.

Statistical information that could be gathered using this project:

- Number of students 21 years of age.
- GPA trends of students.
- Percent of students who are senior citizens (over 64 years of age).
- Percent of students who are minors (under 18 years of age).
- Percent of students on probation (GPA under 2.0).
- Number of employees paying 12% income tax (salary between $9,526 and $38,700)
- Total income tax due for current Employees in one year.
- Total number of Students.
- Total number of Employees.
- Ratio of Students to Employees.

Keziah May
Fall 2018
CSC142
Final Project Report

Important Definitions:

Java Project:

a phrase used to refer to this entire set of programs.

Package:

a word used to describe a set of programs that work together to perform specific tasks on a data set.

Class / object:

words used to describe a program that stores one or more pieces of information and all the ways you can manipulate or use that information. For example, if you store a birthdate and the current date (the information), you can identify the difference in years (the use or manipulation of that information) to learn their age.

Data:

information about an object. For example, a birthdate could be information about a person.

Method:

the way you manipulate given information to return new information.

Return:

to send information back.

Print:

to display information on the computer screen for a user to observe.

File Path:

The location and name of a file or folder, for example:

"C:\Users\E\Documents\School\Fall 2018\CSC_142\F18\CollegePersons.txt"

Is a valid file path on my computer (not on all computers).

Keziah May
Fall 2018
CSC142
Final Project Report

Java project breakdown:

This project contains two packages:

1. a people package, and a
2. database package.

People Package:

The people package contains three classes:

1. a Person Object class,
2. a Student Object class, and
3. an Employee Object class.

Person Objects:

Data: A Person Object holds two pieces of information:

1. a name, and
2. an age.

Constructors: A Person Object can be created one of three ways:

1. With no information: **`new Person()`**

   If a Person object is created with no information,
   - its name is blank (null), and
   - its age is set to zero.
2. Using another Person Object: **`new Person(`**`Person`**`)`**

   If a Person object is created using another Person object, it basically creates a copy.
   - The new Person's name is set to be the other Person's name, and
   - the new persons age is set to be the other Person's age.
3. Using a name and an age: **`new Person(`**`name, age`**`)`**

   If a Person object is created using a name and an age,
   - the name is set to the given name, and
   - the age is set to the given age.

Methods: A Person Object's stored information can be manipulated:

Note: to apply a method to a Person object: `PersonObject`**`.MethodName(`**`RequiredInformation`**`)`**

- **`getAge()`**
  Objective: access Person's age.

  Returns to the user the age stored in the Person object.

- **`getName()`**
  Objective: access Person's name.

  Returns to the user the name stored in the Person object.

- **`hundredth(`**`double`**`)`**
  Objective: round a decimal to the second place (ex. 43.3456 becomes 43.35).

- Requires you to send it a number (not a whole number, it should have a decimal attached. Ex. 43.3456). This method does not require access to information stored in a Person object to run successfully.
- The given number is multiplied by one-hundred (ex. 43.3456 x 100 = 4334.56).
- Then, the number is rounded to the nearest integer (ex. 4334.56 becomes 4335).
- Next, the number is divided by 100 (ex. 4335/100 = 43.35).
- This number is returned to the user (ex. 43.35).

➢ **`printStatus()`**
Objective: identify a Person's relationship to a college.
Returns the title "College Person" to the user.

➢ **`setAge(int)`**
Objective: change the age stored in the Person object.
- Requires you to send it an integer.
- Sets the Person object's age to the value of the given integer.

➢ **`setName(String)`**
Objective: change the name stored in the Person object.
- Requires you to send it a word or String of words (ex. "John" or "John Smith").
- Sets the Person object's name to the word(s) given.

➢ **`toFile()`**
Objective: To access all stored information about a Person, separated by comma's.
- Returns to the user the Person's relationship to a college, and
- the information stored in the Person object (name and age)
- separated by commas (ex. "`College Person,John Smith,32`").

➢ **`toString()`**
Objective: To access stored information about a Person with labels and formatting.
- Returns to the user the Person's relationship to a college, and
- the information stored in the Person object (name and age)
- formatted each on their own line

For example:

```
-------------------------------------------------
 COLLEGE PERSON

    Name: John Smith
     Age: 32                                    ).
```

Employee Objects:

Inheritance note:

An Employee is a type of Person. This means that they contain the same stored data as a Person object and the same methods of manipulating that data.

The following Person methods have different outputs for Employee objects:

- **`printStatus()`**
- **`toFile()`**
- **`toString()`**

Data: An Employee Object holds two pieces of information:

1. an office location, and
2. a salary.

Constructors: An Employee Object can be created one of five ways:

1. With no information: **`new Employee()`**

   If an Employee object is created with no information,
   - its name is blank (null),
   - its age is set to zero,
   - its office location is blank (null), and
   - its salary is set to zero.

2. Using a Person Object, an office location, and a salary: **`new Employee(`**`Person, office, salary`**`)`**

   If an Employee object is created using a Person object, an office location, and a salary,
   - it creates a copy of the given Person's information as described in the "Person Object" section.
   - The Employee's office location is set to the given office location, and
   - the Employee's salary is set to the value of the given salary.

3. Using an office location and a salary: **`new Employee(`**`office, salary`**`)`**

   If an Employee object is created using an office location and a salary,
   - the office location is set to the given location, and
   - the salary is set to the value of the given salary.

4. Using a name and an age: **`new Employee(`**`name, age`**`)`**

   If an Employee object is created using a name and an age,
   - the Employee's name is set to the given name, and
   - the age is set to the value of the given age.

5. Using a name, age, office location and salary: **`new Employee(`**`name, age, office, salary`**`)`**

   If an Employee object is created using a name, age, office location, and salary,
   - the Employee's name is set to the given name,
   - the age is set to the value of the given age,
   - the office location is set to the given location, and
   - the salary is set to the value of the given salary.

Methods: An Employee Object's stored information can be manipulated:

Note: to apply a method to an Employee type Person object use one of the following:

- ✓ `EmployeeObject.MethodName(RequiredInformation)`
- ✓ `((Employee)PersonObject).MethodName(RequiredInformation)`

- **`getOffice()`**
  Objective: access Employee's office location.
  Returns to the user the office location stored in the Employee object.

- **`getSalary()`**
  Objective: access Employee's salary.
  Returns to the user the salary stored in the Employee object.

- **`payments()`**
  Objective: access the Employee's payment installments.
  Sends the salary stored in the Employee object to the Employee object's **`payments(Double)`** method as the required information for that method (see next bullet point).

- **`payments(double)`**
  Objective: calculate payment installments.
  - Requires you to send it a number representing a salary. This method does not require access to information stored in an Employee object to run successfully.
  - The given salary is divided by 24, thus giving the individual payment amounts for an Employee whose salary is dispersed into 24 payments per year.
  - That payment amount is then sent to the **`hundredth(double)`** Person method to get the dollar amount rounded to the penny.
  - The value of this dollar amount is returned to the user.

- **`printStatus()`**
  Objective: identify a Person's relationship to a college.
  - Overrides the Person object printStatus() method.
  - Returns the title "Employee" to the user.

- **`setOffice(String)`**
  Objective: change the office location stored in the Employee object.
  - Requires a new office location.
  - Sets the Employee object's office location to the given office location.

- **`setSalary(Double)`**
  Objective: change the salary amount stored in the Employee object.
  - Requires you to give it a number representing the new salary amount.
  - Sets the Employee object's salary equal to the value of the given number.

- **`toFile()`**
  Objective: access all stored information about a Person, separated by comma's.
  - Overrides the Person object toFile() method.
  - Returns to the user the Person's relationship to a college, and
  - information stored in the Employee object (name, age, office location, and salary)
  - separated by commas (ex. "`Employee,John Smith,32,STEM 207,52125.0`").

- **`toString()`**
  Objective: access stored information about a Person with labels and formatting.
  - Overrides the Person object toString() method.
  - Returns to the user the Person's relationship to a college, and
  - information stored in the Employee object (name, age, office location, salary, and payment installments)
  - formatted each on their own line.
    For example:

```
--------------------------------------------------
 EMPLOYEE

    Name: John Smith
     Age: 32
  Office: STEM 207
  Salary: 52125.0
Payments: 2171.88
--------------------------------------------------
```

Keziah May
Fall 2018
CSC142
Final Project Report

Student Objects:

Inheritance note:

A Student is a type of Person. This means that they contain the same stored data as a Person object and the same methods of manipulating that data.

The following Person methods have different outputs for Student objects:

- **`printStatus()`**
- **`toFile()`**
- **`toString()`**

Data: A Student Object holds two pieces of information:

1. a GPA, and
2. a college Status, which can be FRESHMAN, SOPHOMORE, JUNIOR, SENIOR, or null *exclusively*.

Constructors: A Student Object can be created one of five ways:

1. With no information: **`new Student()`**

   If a Student object is created with no information,
   - its name is blank (null),
   - its age is set to zero,
   - its Status is blank (null), and
   - its GPA is set to zero.

2. Using a Person Object, a Status, and GPA: **`new Student(`**`Person, Status, GPA`**`)`**

   If a Student object is created using a Person object, a Status, and a GPA,
   - it creates a copy of the given Person's information as described in the "Person Object" section.
   - The Student object's college status is set to the Status, and
   - the Student object's GPA is set to the value of the given GPA.

3. Using a Status and a GPA: **`new Student(`**`Status, GPA`**`)`**

   If a Student object is created using a Status and a GPA,
   - the name is blank (null),
   - the age is set to zero,
   - the Status is set to the given Status, and
   - the GPA is set to the value of the given GPA.

4. Using a name and an age: **`new Student(`**`name, age`**`)`**

   If a Student object is created using a name and an age,
   - the Student object's name is set to the given name,
   - the age is set to the value of the given age,
   - the Status is blank (null), and
   - the GPA is set to zero.

5. Using a name, age, Status, and GPA: **`new Student(`**`name, age, Status, GPA`**`)`**

   If a Student object is created using a name, age, Status, and GPA,
   - the Student's name is set to the given name,
   - the age is set to the value of the given age,

- the Status is set to the given Status, and
- the GPA is set to the value of the given GPA.

Methods: A Student Object's stored information can be manipulated:

Note: to apply a method to a Student type Person object use one of the following:

- ✓ `StudentObject.MethodName(RequiredInformation)`
- ✓ `((Student)PersonObject).MethodName(RequiredInformation)`

- **`getGPA()`**
  Objective: access Student's GPA.
  Returns to the user the value of the GPA stored in the Student object.

- **`getStatus()`**
  Objective: access Student's Status.
  Returns to the user the Status stored in the Student object.

- **`isOnProbation()`**
  Objective: determine whether or not a Student is on probation.
  - Accesses the GPA stored in the Student object.
  - If this GPA is greater than or equal to 2.0,
    - the student is not on probation.
    - In this case, the value 'false' would be returned to the user.
  - If the GPA is greater than or equal to zero, but less than 2.0,
    - the student is on probation.
    - In this case, the value 'true' would be returned to the user.

- **`printStatus()`**
  Objective: identify a Person's relationship to a college.
  - Overrides the Person object method.
  - Returns the title "Student" to the user.

- **`setGPA(Double)`**
  Objective: change the GPA stored in the Student object.
  - Requires you to give it a number representing the new GPA.
  - Sets the Student object's GPA equal to the value of the given number so long as the number is positive.

- **`setStatus(Status)`**
  Objective: change the Status stored in the Student object.
  - Requires a new Status.
  - Sets the Student object's Status to the given Status.

- **`toFile()`**
  Objective: access all stored information about a Person, separated by comma's.
  - Overrides the Person object method.

- Returns to the user the Person's relationship to a college, and
- the information stored in the Student object (name, age, GPA, and Status)
- separated by commas (ex. "`Student,Jane Doe,21,SOPHOMORE,3.3`").

➢ **`toString()`**
Objective: access stored information about a Person with labels and formatting.

- Overrides the Person object method.
- Returns to the user the Person's relationship to a college, and
- the information stored in the Student object (name, age, GPA, and Status)
- formatted each on their own line.
  For Example:

```
-------------------------------------------------
 STUDENT

    Name: Jane Doe
     Age: 21
     GPA: 3.3
  Status: SOPHOMORE
-------------------------------------------------
```

Keziah May
Fall 2018
CSC142
Final Project Report

Database Package:

The database package contains two classes:

1. a DatabaseStorage Object class, and
2. a CollegeDatabase Object class.

   Note**:** CollegeDatabase is a type of Database, which means it is required to contain all Database methods as listed below:

   - **`add(Person)`**
   - **`delete(String)`**
   - **`search(String)`**

DatabaseStorage Objects:

Data: A DatabaseStorage Object holds one piece of information:

1. a file.

Constructors: A DatabaseStorage Object can be created one of two ways:

1. With no information: **`new DatabaseStorage()`**

   If a DatabaseStorage object is created with no information, it's file is blank (null).

2. Using a file path: **`new DatabaseStorage(filePath)`**

   If a DatabaseStorage object is created using a file path (file location and name),

   - it accesses the **`setFile(String)`** DatabaseStorage method described below
   - to set the DatabaseStorage object's file to be the file at the specified location with the specified name.

Methods: A DatabaseStorage Object's stored information can be manipulated:

Note: to apply a method to a DatabaseStorage object:

```
DatabaseStorageObject.MethodName(RequiredInformation)
```

- **`parsePerson(String)`**
  Objective: translate information on a single line of a file into a format that the program understands and can manipulate.
  - Requires a single line from a file.
  - It checks the content of the line to see if it describes an Employee or a Student.
  - It returns the information to the user as an Employee or a Student.

- **`read()`**
  Objective: translate each line in a file into useable and manipulatable information about Students and Employees and store it in a College Database.
  - Opens the file stored in the current DatabaseStorage object.
  - Sends each line into the **`parsePerson(String)`** method.
  - Stores Information from each line in a College Database.
  - Returns the database to the user.

- **`setFile(String)`**
  Objective: set the name and location of the file to store in the DatabaseStorage object.

- Requires you to send it a file path.
- It sets the name and location of the file stored in the DatabaseStorage object to the given file path.

➢ **`write(CollegeDatabase)`**
Objective: write to a file stored information about a database of Person's, with labels and formatting.

- Requires you to give it a College Database.
- Writes to the file stored in the DatabaseStorage object information about every Student and Employee in the database.
- The information will be clearly labeled using the **`toString()`** methods for Students and Employees.
  For example, if the database contained only the Employee and Student we discussed earlier, this method would write the following text to the file stored in the DatabaseStorage object:

```
-------------------------------------------------
 EMPLOYEE

    Name: John Smith
     Age: 32
  Office: STEM 207
  Salary: 52125.0
Payments: 2171.88
-------------------------------------------------

-------------------------------------------------
 STUDENT

    Name: Jane Doe
     Age: 21
     GPA: 3.3
  Status: SOPHOMORE
-------------------------------------------------
```

➢ **`writeCommaSeparated(CollegeDatabase)`**
Objective: To access all stored information about a Person, separated by commas.

- Requires you to give it a College Database.
- Writes to the file stored in the DatabaseStorage object information about every Student and Employee in the database.
- The information will be written as in the **`toFile()`** methods for Students and Employees For Example:

```
Student,Jane Doe,21,SOPHOMORE,3.3
Employee,John Smith,32,STEM 207,52125.0
```

Keziah May
Fall 2018
CSC142
Final Project Report

CollegeDatabase Objects:

Inheritance note:

A CollegeDatabase is a type of Database, which means it must meet the minimum requirements for a Database object.

The following methods are required for all Database Objects:

- **add(Person)**
- **delete(String)**
- **search(String)**

Data: A CollegeDatabase Object holds 6 pieces of information:

1. A database (list), of Person objects (Employees and Students),
2. The size or number of people in the database,
3. The DEFAULT_SIZE for a database (the number 20),
4. An value representing a person who was NOT_FOUND in the database (the number -1),
5. An immutable Person object: **new Person("NOT_FOUND", 0)** representing a PERSON_NOT_FOUND in the database.
6. and a program that can accept input submissions from a user (input scanner).

Constructors: A CollegeDatabase Object can be created one of two ways:

1. With no information: **new CollegeDatabase()**
   If a CollegeDatabase object is created with no information,
   - it sends the default size to the next option (See number two below).
2. Using an integer capacity: **new CollegeDatabase(capacity)**
   If a CollegeDatabase object is created with a capacity,
   - the capacity of the database inside this object is set to the given number.
   - The number of people stored in the database is set to zero.

Methods: A CollegeDatabase Object's stored information can be manipulated:

Note: to apply a method to a CollegeDatabase type object use the following:

- ✓ CollegeDatabaseObject.**MethodName(**RequiredInformation**)**

- **add(CollegeDatabase) : void**
  Objective: add another CollegeDatabase to the end of the current CollegeDatabase.
  - Requires you to send this CollegeDatabase method *another* CollegeDatabase.
  - It accesses every Person (Employee or Student) in the *other* CollegeDatabase and sends them one at a time to the **add(Person)** method for the current CollegeDatabase (See next bullet point).

- **add(Person) : void**
  Objective: add a Person to the database.
  - Requires you to send it a Person object (Employee or Student) and
  - if no one else in the database shares their name,

        - If the database is full, the database size is increased using the **`expand()`** method described below.
        - Next, it adds the Person to the database stored in the CollegeDatabase object, and
        - updates the size of the database, incrementing it by one.
    - If someone else in the database shares their name, the person is not added to the database, and their information is printed to the screen with a message explaining why they were not added to the list.
    - Will not add `CollegeDatabase.PERSON_NOT_FOUND` to the database.

- **`byGPA(double, Operator) : CollegeDatabase`**
  Objective: from a list of Students, separate a list of Students of a particular GPA range.
    - Note: this method is private, and cannot be accessed by a user directly. It is used exclusively by a method (**`separate`**) described later.
    - It requires the method which uses it to send it
        - a number (any positive real number representing a GPA), and
        - an Operator ("OVER" or "UNDER" exclusively).
    - It deletes (with the **`delete(int)`** method) Student objects from the current CollegeDatabase that have a GPA outside of the specified range.

- **`bySalary(double, Operator) : CollegeDatabase`**
  Objective: from a list of Employees, separate a list of Employees of a particular salary range.
    - Note: this method is private, and cannot be accessed by a user directly. It is used exclusively by a method (**`separate`**) described later.
    - It requires the method which uses it to send it
        - a number (any positive real number representing a salary), and
        - an Operator ("OVER" or "UNDER" exclusively).
    - It deletes (with the **`delete(int)`** method) Employee objects from the current CollegeDatabase that have a salary outside of the given range.

- **`delete(int) : boolean`**
  Objective: delete a person from the database.
    - Note: this method is private, and cannot be accessed by a user directly. It is used exclusively by methods described later.
    - It requires you to send it an integer representing a Person's position (index or location) in the database.
    - If the number is negative or larger than the size of the database,
        - the location does not exist, so
        - the program returns the value 'false' to the user.
    - If, instead, the number represents a real location,
        - It goes to that location and deletes its contents, and
        - shifts every Person following this position back one to fill in the empty space.
        - Then, it reduces the stored value for the size of the database by one.
        - When it is finished, it returns the value 'true' to the user.

- **`delete(String) : boolean`**
  Objective: delete a person from the database.
  - It requires you to send it a string of words representing a Person's name.
  - It sends that name to the **`findLocation(String)`** method described below to access the Person's location in the database.
  - It sends this location to the **`delete(int)`** method described above.
  - It checks to see if the return value for the **`delete(int)`** method was 'true' or 'false'.
    - If the value is 'true',
      - that same value, 'true' is returned to the user.
    - If, instead, the value is 'false',
      - It prints a message letting you know the person was not found in the database and
      - asks you to enter a new name.
        - If the person enters "q" to quit, the delete method returns the value it was passed from the **`delete(int)`** method, 'false'.
        - If instead, they enter a new string of words, the method restarts itself with the name given.

- **`deleteDatabase(CollegeDatabase) : void`**
  Objective: delete multiple people from the database.
  - It requires you to send it *another* CollegeDatabase.
  - For every Person stored in the *other* CollegeDatabase,
    - It uses the Person's **`getName()`** method to access their name.
    - It sends that name to the **`findLocation(String)`** method described below to access the Person's location in the database.
    - It sends that location to the **`delete(int)`** method.
    - If the current Person in the *other* CollegeDatabase is found in the current CollegeDatabase, they are deleted.
  - It then prints to the screen the successful, failed, and attempted deletions.

- **`expand() : void`**
  Objective: double the capacity of the current CollegeDatabase.
  - Note: this method is private, and cannot be accessed by a user directly.
  - It creates a new database with twice the capacity of the current CollegeDatabase.
  - It then copies the current CollegeDatabase to the new database and
  - sets the database stored in this CollegeDatabase object to be the new database.
  - The newly created database is now the database used in this CollegeDatabase object going forward.

- **`findLocation(String) : int`**
  Objective: find the location of a Person in a CollegeDatabase.
  - Note: this method is private, and cannot be accessed by a user directly.
  - Requires a method to send it a string of words representing a Person's name.
  - It then goes into every entry in the database,

        - accesses its name with the **`getName()`** Person method,
        - and compares each name with the given name.
    - If the names are an exact match,
        - an integer representing that entry's location is returned to the calling method.
    - If no names in the database match the given name,
        - The value of NOT_FOUND (-1) is returned to the calling method.

- **`get(int) : Person`**
  Objective: get information about a Person.
    - Requires you to send it an integer representing a location in the database.
    - If the location is valid,
        - It returns to the user the Person object stored at that location.
    - If the location is not valid,
        - It returns PERSON_NOT_FOUND.

- **`getDatabase() : Person[]`**
  Objective: access the full database.
    - Returns to the user the stored database of Person objects.

- **`getSize() : int`**
  Objective: find out how many Person objects are currently being stored in the database.
    - Returns to the user the number of person objects in the database.

- **`pop(int) : Person`**
  Objective: access a Person object and delete it from the database.
    - Requires you to send it an integer representing a location in the database.
    - If the location is a valid location in the database,
        - That Person object at that location is stored temporarily in the method,
        - deleted from the database using the **`delete(int)`** method,
        - and returned to the user.
    - If the location is NOT valid, the result from the **`get(int)`** method is returned to the user.

- **`print() : void`**
  Objective: print information about Person objects in the database for the user.
    - Prints to the screen the returned information from the CollegeDatabase **`toString()`** method defined below.

- **`printCommaSeparated() : void`**
  Objective: print information about Person objects in the database.
    - Prints to the screen the returned information from the CollegeDatabase **`toFile()`** method defined below.

- **`search(String) : Person`**
  Objective: find information about a Person in the database.
    - Requires you to send it a String of words representing a Person's name.
    - It uses the **`findLocation(String)`** method to find the Person's location in the database.

- If the person is not found in the database, a PERSON_NOT_FOUND Person object is returned to the user and a message explaining that is printed to the screen.
- If the Person is found, the Person object is returned to the user.

➢ **`separate(int) : CollegeDatabase`**
Objective: make a separate CollegeDatabase of Persons with a common age.
- Requires you to send it an integer representing an age.
- It creates a new CollegeDatabase object and
- Fills it with Person objects from the existing CollegeDatabase that have the given age.
- The new CollegeDatabase is returned to the user, and the original database is unchanged.

➢ **`separate(int, Operator) : CollegeDatabase`**
Objective: make a separate CollegeDatabase of Persons in a common age range.
- Requires you to send it an integer representing an age, and an *Operator* ("OVER" or "UNDER" exclusively).
- It creates a new CollegeDatabase object.
- If the *Operator* is "OVER",
  - it fills the CollegeDatabase with Person objects from the existing CollegeDatabase whose age is greater than the given integer.
- If the *Operator* is "UNDER",
  - it fills the CollegeDatabase with Person objects from the existing CollegeDatabase whose age is less than the given integer.
- The new CollegeDatabase is returned to the user, and the original database is unchanged.

➢ **`separate(PersonType) : CollegeDatabase`**
Objective: make a separate CollegeDatabase of Persons of a particular sub-type.
- Requires you to send it a *PersonType* ("STUDENT" or "EMPLOYEE" exclusively).
- It creates a new CollegeDatabase object.
- If the *PersonType* is "EMPLOYEE",
  - it fills the CollegeDatabase with Person objects from the existing CollegeDatabase who are of Employee type.
- Otherwise,
  - it fills the CollegeDatabase with Person objects from the existing CollegeDatabase who are of Student type.
- The new CollegeDatabase is returned to the user, and the original database is unchanged.

➢ **`separate(PersonType, double, Operator) : CollegeDatabase`**
Objective: make a separate CollegeDatabase of Persons in a common GPA or salary range.
- Requires you to send it a *PersonType* ("STUDENT" or "EMPLOYEE" exclusively), a number (representing either a GPA or a salary depending on the *PersonType*), and an *Operator* ("OVER" or "UNDER" exclusively).

- It creates a new CollegeDatabase equal to the returned CollegeDatabase of the **`separate(`***`PersonType`***`)`** method when sent the given PersonType.
- If the *PersonType* is Employee,
  - The new CollegeDatabase of Employee objects is then edited using the **`bySalary(`**`double, `***`Operator`***`)`** method.
- Otherwise,
  - The new CollegeDatabase of Student objects is then edited using the **`byGPA(`**`double, `***`Operator`***`)`** method.
- The new CollegeDatabase is returned to the user.

➢ **`toFile() : String`**
Objective: access all stored information about each Person stored in the CollegeDatabase, with data about a Person separated by comma's, and Person's separated each on their own line.

- It creates a place to temporarily store information in the method.
- For every entry in the CollegeDatabase,
  - it accesses that Person's **`toFile()`** method and adds the returned information to the temporary storage.
  - If the program finds a hole in the database where there is no Person object, the program stops and lets you know what happened.
- The information in the temporary storage is returned to the user.
  For Example:

```
Student,Jane Doe,21,SOPHOMORE,3.3
Employee,John Smith,32,STEM 207,52125.0
```

➢ **`toString() : String`**
Objective: access all stored information about each Person stored in the CollegeDatabase, with labels and formatting.

- It creates a place to temporarily store information in the method.
- For every entry in the CollegeDatabase,
  - it accesses that Person's **`toString()`** method and adds the returned information to the temporary storage.
  - If the program finds a hole in the database where there is no Person object, the program stops and lets you know what happened.
- The information in the temporary storage is returned to the user.
  (Example on next page)

For example:

```
-------------------------------------------------
 EMPLOYEE

    Name: John Smith
     Age: 32
  Office: STEM 207
  Salary: 52125.0
Payments: 2171.88
-------------------------------------------------

-------------------------------------------------
 STUDENT

    Name: Jane Doe
     Age: 21
     GPA: 3.3
  Status: SOPHOMORE
-------------------------------------------------
```

DatabaseClient testing report:

add(Person) method tests

Does not accept repeated names

Prints an error message to the user explaining which person could not be added.

Accurately populates an array

Creates an array stored in the CollegeDatabase object ‘database’ which accurately represents information read from a file.

Does not accept PERSON_NOT_FOUND (CollegeDatabase.PERSON_NOT_FOUND)

If the Person you are trying to add to the database is CollegeDatabase.PERSON_NOT_FOUND, the add method will ignore your request.

get(int) method tests

Handles a negative index (-9)

It reminds you of the index you sent it and informs you that it is not valid. If you try to send the Person it returns to you to the add method, it will ignore your request. If you try to print it, it will let you know that the Person object was not found.

Handles out of bounds index (size + 400)

Handled the same way as the previous test (-9).

Handles a valid index (0)

The Person object at the given (valid) index is returned.

search(String) method tests

Handles searching for Person at beginning of database (index 0)

Returns the correct Person object.

Handles searching for Person in the middle of the database (size / 2)

Same as previous test.

Handles searching for Person at the end of the database (size – 1)

Same as previous test.

Handles attempting to search for Person not in the database

It reminds you of the name you sent it and informs you that no such Person exists in the database. If you try to send the Person it returns to you to the add method, it will ignore your request. If you try to print it, it will let you know that the Person object was not found.

delete(String) method tests

Handles deleting Person at beginning of database (index 0)

Deletes Person and returns true.

Handles deleting Person in the middle of the database (size / 2)

Same as previous test.

Handles deleting Person at the end of the database (size – 1)

Same as previous test.

Handles attempting to delete a Person not in the database

It reminds you of the name that you gave it and informs you that a Person with that name is not stored in the database. It then allows you to correct the name (incase it was misspelled or something). You can correct the name infinitely unless you either enter a valid name, or enter “q” to quit. If you enter a valid name, the Person is deleted and it returns ‘true’. If you enter “q”, it returns ‘false’ and the database remains unchanged.

Separation Criteria Tests

Employee Separations:

Can separate a database into a database of only Employees

All Person objects of Employee type from the database are stored in a separate database and returned to the user.

Can separate a database into a database of Employees with a salary greater than $55,000.00

Successfully creates database and populates it with Employee objects with the given criteria.

Can separate a database into a database of Employees over 40 years of age

Successfully creates database and populates it with Employee objects with the given criteria.

Student Separations:

Can separate a database into a database of only Students

All Person objects of Student type from the database are stored in a separate database and returned to the user.

Can separate a database into a database of Students whose GPA is less than 2.0

Successfully creates database and populates it with Student objects with the given criteria.

Can separate a database into a database of Students whose GPA is less than 3.0

Successfully creates database and populates it with Student objects with the given criteria.

Can separate a database into a database of Students who are under 25 years of age

Successfully creates database and populates it with Student objects with the given criteria.

write() method test: **********

Can write to a file in a neat format that is easy for a user to read and understand

Note: I figured out how to make this method write properly to a file. If you recall, I had the issue where my entire string was writing to a single line. Every line of information about every Person in the database was being written to line one of the file.

How I figured it out: I ran some tests and realized that the writer was ignoring string newline characters like "\n" as well as print newlines such as System.out.println("").

I vaguely understand that Java is as popular as it is today because it has a virtual machine that helps it communicate with the operating system. I remember that you mentioned that your code wrote correctly to some operating systems and not to others. This made me wonder if the People who work for Java and wrote the code for the writer had found a way to communicate newlines to every operating system when writing to a file.

To test this, I separated information about each Person on its own writer.println(""). Alas, it worked! But, even having information about one Person on a single line looked, well, tacky. For this reason, I used the String.split("\n") method to split the toString() method for each Person at the "\n" newline character. I then had the writer.println("") access each index of the new String[ ] array and print it using the superior Java-employees' code.

On my operating system (windows), this writes to the file in the exact same format as it would print to the consol. I am curious if it does the same thing on your mac.

Keziah May
Fall 2018
CSC142
Final Project Report