Computer Science 21A (Spring, 2021)
Data Structures and Algorithms

Programming Assignment 0

- Your assignment should be submitted via Latte as an exported eclipse project (.zip file named LastnameFirstnamePA0.zip) by the date it is due.
- Name your eclipse project LastnameFirstnamePA0.
- Each class should have the proper header (see the Additional Notes at the bottom).
- Use the provided template. Make sure that you have the most recent version of java installed, since otherwise you might run into errors when trying to use the provided skeleton code. You can download the most recent JRE here.
- Late submissions will not receive credit.

Overview:

This assignment is meant to test the skills you learned in 12B and prepare you for 21A. This assignment will involve file reading, Scanners, one and two-dimensional arrays, object-oriented programming, and your ability to code modularly. You should read the entire PDF at least once before you start coding.

Background:

You've been hired by a company to try to pair a list of families with a list of available homes on the market. These families have a number of members, a budget for the cost of their future home, and they may or may not have pets. Correspondingly, the homes on the market have a total number of bedrooms, a cost, and may or may not allow pets. Your job is to assign these families into a home that can accommodate them.

Basic Idea:

You will be given two text files, one that has a list of all of the families and one that has a list of all of the available houses. Your job is to read these files, construct objects from these lists (as will be explained more in depth later), compare the properties of the objects (specifically between the families and the houses) and match the ones that have properties that align.

Requirements:

In the PA0.zip file provided, you will find 5 classes: Person.java, Pet.java, House.java, Family.java, and Main.java. Your job is to complete these classes using the provided skeletons. Below is an explanation of what fields and methods each class should have. You can add additional fields and helper methods if you feel they're needed, but **do not change the names of the given fields or the signatures of the given methods**. You may add `throws` clauses to methods that require them.

You should also write **Junit tests** to test the functionality of every method and class that you write, except for the methods in Main.java. The PersonTest class provides some example tests and directions that you can reference for formatting. Remember that the more extensive your tests are, the more likely you are to catch and fix any errors!

Person.java

- Fields
  - `private String name`
  - `private int age`
  - `private int salary`
- Constructor
  - `Person(String name, int age, int salary)` – this should initialize the three fields listed above
- Methods - be sure to add an appropriate visibility modifier for each method
  - `String getName()`
  - `int getAge()`
  - `int getSalary()`
  - `String speak()` – this should return a String representation of what the person says if they have been assigned to a house that does not have enough rooms for their family. This depends on whether the person is a child (age <= 18) or an adult (age > 18).
    - A child should say the following:
      "I want a bigger house!"
    - An adult should say the following:
      "This house does not have enough rooms to accommodate my family. I would like my family to be assigned to a house with more rooms."
  - `String toString()` – this should return a String representation of this Person. The format is up to you, but it must contain relevant information about this Person object.

Pet.java

- Fields
  - `private String name`
  - `private String species`
  - `private int age`
- Constructor
  - `Pet(String name, String species, int age)` – this should initialize the three fields listed above
- Methods (you may add helper methods, if you feel they are needed)
  - `String getName()`
  - `String getSpecies()`
  - `int getAge()`
  - `String makeSound()` – this should return a String representation of the sound this pet makes, based on the pet's species. A Cat should say "meow!", and a Dog should say "bark!". Remember to account for possible differences in capitalization. Feel free to add more sounds for other specific species. Any other animal should say "squak!".
  - `String toString()` – this should return a String representation of this Pet. The format is up to you, but it must contain relevant information about this Pet object.

House.java

- Fields
  - `private int rooms`
  - `private int price`
  - `private boolean petsAllowed`
- Constructor
  - `House(int rooms, int price, boolean petsAllowed)` – this should initialize the three fields listed above
- Methods (you may add helper methods, if you feel they are needed)
  - `int getRooms()`
  - `int getPrice()`
  - `boolean petsAllowed()`
  - `String toString()` – this should return a String representation of this House. The format is up to you.

Family.java

- Fields
  - `Person[] familyMembers`
  - `Pet[] familyPets`
- Constructor
  - `Family(int humans, int pets)` – this should initialize your two arrays
- Methods (you may add helper methods, if you feel they are needed)
  - `Person[] getPeople()`
  - `Pet[] getPets()`
  - `int getBudget()` – this should return the budget of your family, which is equal to the sum of all the salaries of the persons
  - `int numberOfPeople()` – this should return the *total* number of people that will be in the family once all of the people are added
  - `int numberOfPets()` – this should return the *total* number of pets that will be in the family once all of the pets are added
  - `boolean addMember(Person p)` – this should add the person to the family if this is possible, and return true if the person was added to the family successfully and false otherwise
  - `boolean addPet(Pet p)` – same idea as `addMember`
  - `String toString()` – this should return a String representation of this Family. The format is up to you. Hint: You should use the Person and Pet toString() methods that you've already written to help with this.

Main.java

- Fields
  - `Family[] f` – this should be an array of the families read from familyUnits.txt
  - `House[] h` – this should be an array of the houses read from housingUnits.txt

- `boolean[][] assignments` – this is a 2D matrix that represents the assignments of families to houses. Each row corresponds to a family, and each column corresponds to a house. The value in the cell at row `i` and column `j` should be `true` if family i has been assigned to house j, and `false` otherwise. Note that since each family can be assigned to at most one house and each house can be assigned to at most one family, each row and each column in this matrix should have at most one value equal to `true`.

e.g. In this example, there are three families and four houses. The `assignments` array indicates that Family1 was assigned to House4, Family2 was not assigned to a house, and Family3 was assigned to House1.

`f:`

0	1	2
Family1	Family2	Family3

`h:`

0	1	2	3
House1	House2	House3	House4

`assignments:`

	0	1	2	3
0	false	false	false	true
1	false	false	false	false
2	true	false	false	false

- Methods
  - `void createFamilies(Scanner s)` – the `Scanner` here should be reading from the familyUnits.txt file, and this method should populate the `Family` array with families and each family with family members.
  - `void createHomes(Scanner s)` – same as the previous method, but with the homes.
  - `void assignFamiliesToHomes()` – this is the real algorithm part of the assignment, where you will compare the fields of the families and homes and assign homes to families in the `assignment` 2D array (Hint: make sure that you aren't assigning two families to the same home or two homes to one family!)
    - Note: you should go through the Family[] and House[] in order (the order they were read in from the file). You must start with the first Family and assign it to the first House that can accommodate it, NOT the House that matches the Family's requirements most closely.
  - `void displayAssignments()` – this should print all of the assignments created by `assignFamiliesToHomes()`. The exact format is up to you, but we suggest printing each family, followed either by the home it was assigned to or by "not assigned to home" if it was not assigned to a home.
    (Hint: if you write the toString() methods for Family.java and House.java in informative ways and use them in this method, you can use this method to easily spot an error in your assignments (e.g. a family with 5 people being assigned to a home with 2 rooms) and use it to debug your assignment algorithm!)

- `void checkAssignment(int familyIndex)` – this method checks that the assignment given to the family at index `familyIndex` of the `f` array is valid.
    - If the house assigned to this family does not have enough rooms (i.e. it has fewer rooms than the family has people), then this method should call the `speak()` method on each family member, and print the resulting Strings to the console.
    - If the family has pets but the house assigned to it does not allow pets, then this method should call the `makeSound()` method on each family pet and print the resulting Strings to the console.
    - If the price of the House is greater than the budget of the Family, the message "house over budget" should be printed to the console.
    - If the Family has been assigned to more than one House, the message "family assigned to more than one house" should be printed to the console.
- The main method should call `createFamilies()` and `createHomes()` on the two provided .txt files. It should then call the `assignFamiliesToHomes()` method, followed by `displayAssignments()`. Finally, it should call `checkAssignment()` for each Family.

Text Files

You have been provided two text files. Feel free to change them to test your code's functionality. However, do not change the formats from the ones shown below, since this is the format of the text files we will use for testing.

The format for the familyUnits.txt file is:

```
1        //this is how many families are in the file, in this case, one

3             //this is how many Persons there are in the family
3             //this is how many Pets there are in the family
Bill 46 45000  //person 1 (name, age, salary)
Mary 47 45000  //person 2 (name, age, salary)
Kris 12 0      //person 3 (name, age, salary)
Bark Dog 3     //pet 1 (name, species, age)
Meow Cat 6     //pet 2 (name, species, age)
Gob Lizard 1   //pet 3 (name, species, age)
```

The format for the housingUnits.txt file is:

```
2          //this is how many homes are in the file, in this case, two

5          //this is the number of rooms in the first house
30000      //this is the cost of the first house
false      //this is whether pets are allowed or not (it is a boolean)

3
40000
true
```

This layout of the familyUnits.txt and housingUnits.txt files **is on purpose**. **Use it to simplify your file reading process!**

IMPORTANT

- This is **NOT** an optimization problem. You do not have to make sure the family goes into the "best fitting" home. You should put the family into the **first home that can accommodate them**. This is very important as it will change your matching output!
- Do not change any of the given method headers or parameters, other than by adding `throws` clauses to methods that need them.
- The only data structure you may use is **an array**. Do not use/implement Lists, Maps, etc.
- Please comment your code! Use JavaDoc method comments and additional comments on any particularly complex pieces of code.

Additional Notes

- Properly heading your classes – **this must be at the top of each class you write:**

```
/**
 * first_name last_name
 * email@brandeis.edu
 * Month Date, Year
 * PA#
 * Explanation of the program/class
 * Known Bugs: explain bugs/null pointers/etc.
 */

>Start of your class here<
```

- Java Docs
  - Before every one of your methods, you add a Java Doc comment. This is a very easy way to quickly comment your code while keeping everything neat.
  - The way to create a Java Doc is to type:

    ```
    /** + return
    ```
  - This should create:

    ```
    /**
     *
     */
    ```
  - Depending on your method, the Java Doc may also create additional things, such as `@param`, `@throws`, or `@return`. These are autogenerated, depending on if your method has parameters, throw conditions, or returns something. You should fill out each of those tags with information on parameter, exception, or return value.
  - If you want to read more on Java Docs, you can find that here.