CS2103 2018 B-Term -- Project 1 -- Facebuk

Prof. Jacob Whitehill

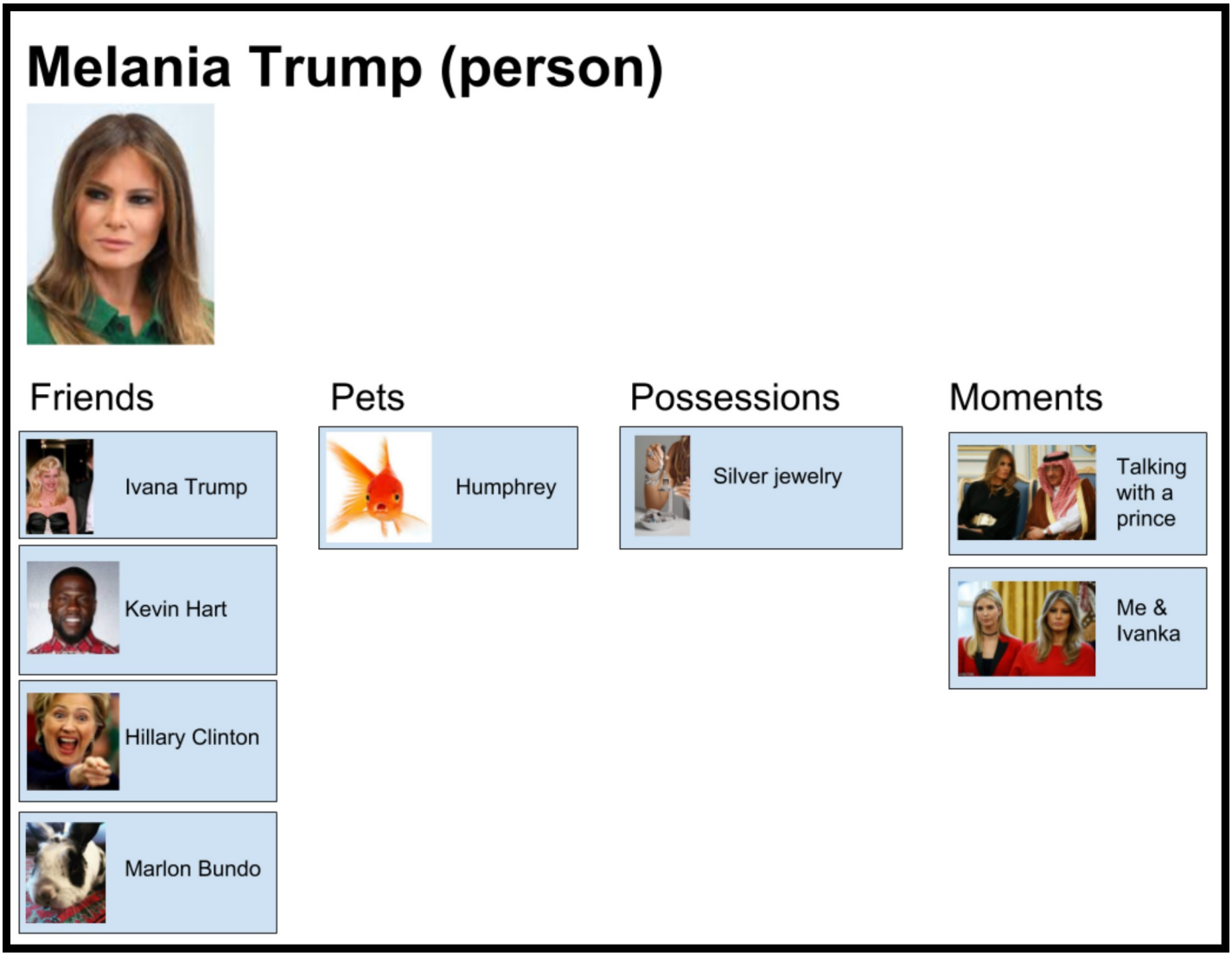
Introduction

The goal of this assignment is to practice thinking about designing software using object-oriented programming techniques in Java. Somewhat similar to our discussion during class of `match.com`, the theme of this programming project will also be social networks. In particular, this project asks you to think about how you would build a site called `Facebuk`, described in the next section.

Facebuk

People, Pets, & Possessions

Facebuk is a social networking site, somewhat akin to `Facebook`, that supports profiles not only for **people**, but also people's **pets** and **possessions**. People love their pets and love posting photos of their pets. Moreover, people sometimes want to sell their possessions online. As an example of how `Facebuk` might look, suppose that Melania Trump decided to join Facebuk:

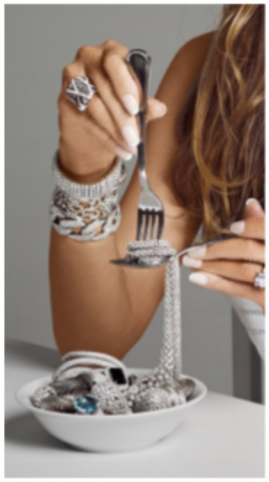


Melania's Facebuk page above shows that:

- Her **friends** consist of Ivana Trump (the ex-wife of Melania's current husband), Kevin Hart, Hillary Clinton, and Marlon Bundo (a bunny that belongs to a different family).
- Melania has one (somewhat reclusive) **pet**: a goldfish named Humphrey.
- Melania also owns some silver jewelry as a **possession**.
- Melania has, at certain **moments** (defined in the next section), appeared in photographs both with her daughter (Ivanka, not to be confused with Ivana) as well as a Saudi prince.

Below is the Facebuk page of Melania's silver jewelry (on sale):

Silver jewelry (possession)



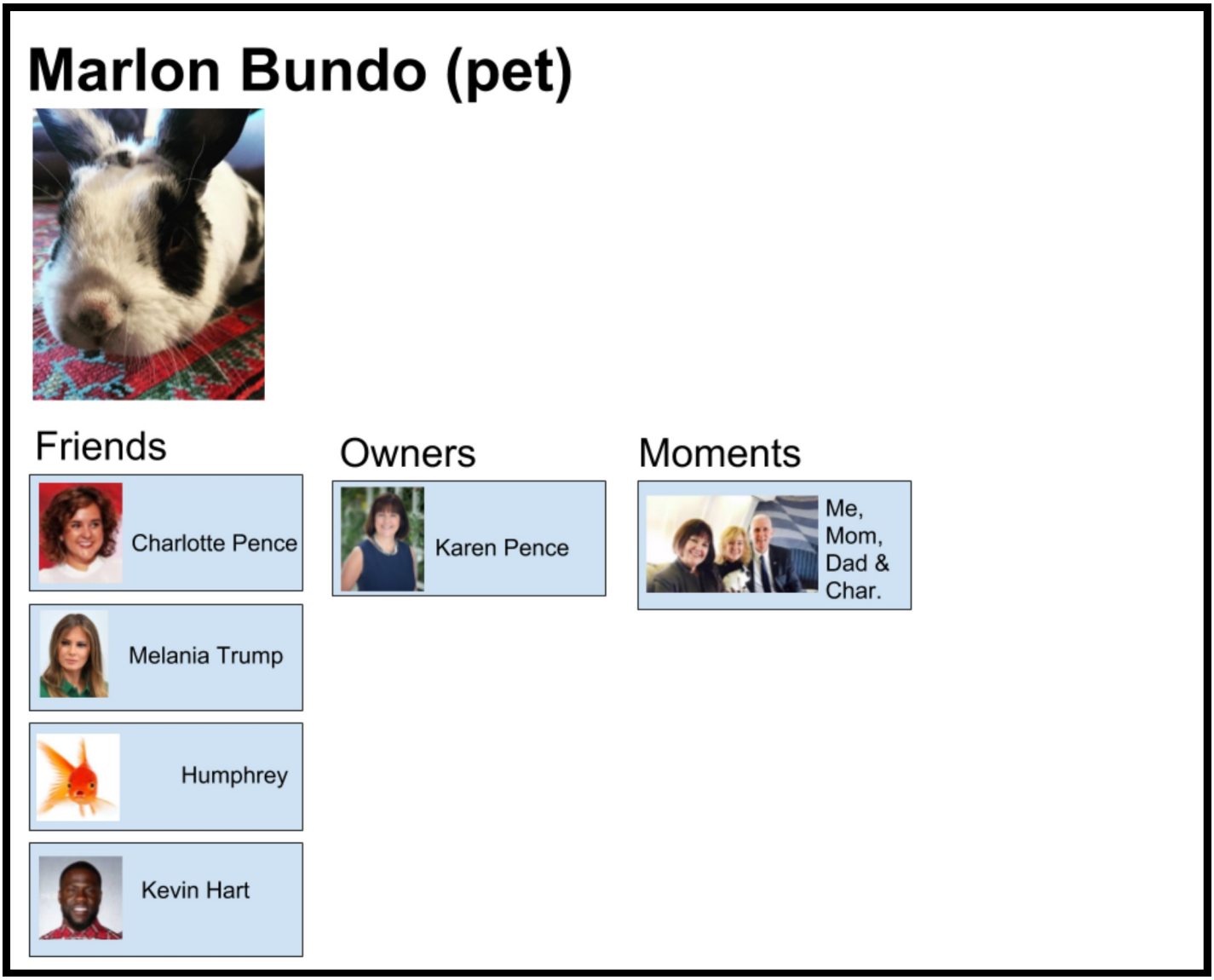
Owner

Melania Trump

Price

$19,999,999.98

Marlon Bundo is the pet bunny of Karen Pence. (Karen's husband is the Vice President of a largish country in North America.) Marlon has his own Facebuk page too, which shows that he has four friends of his own: Charlotte Pence (the daughter of Karen Pence), Melania Trump, Humphrey, and Kevin Hart.

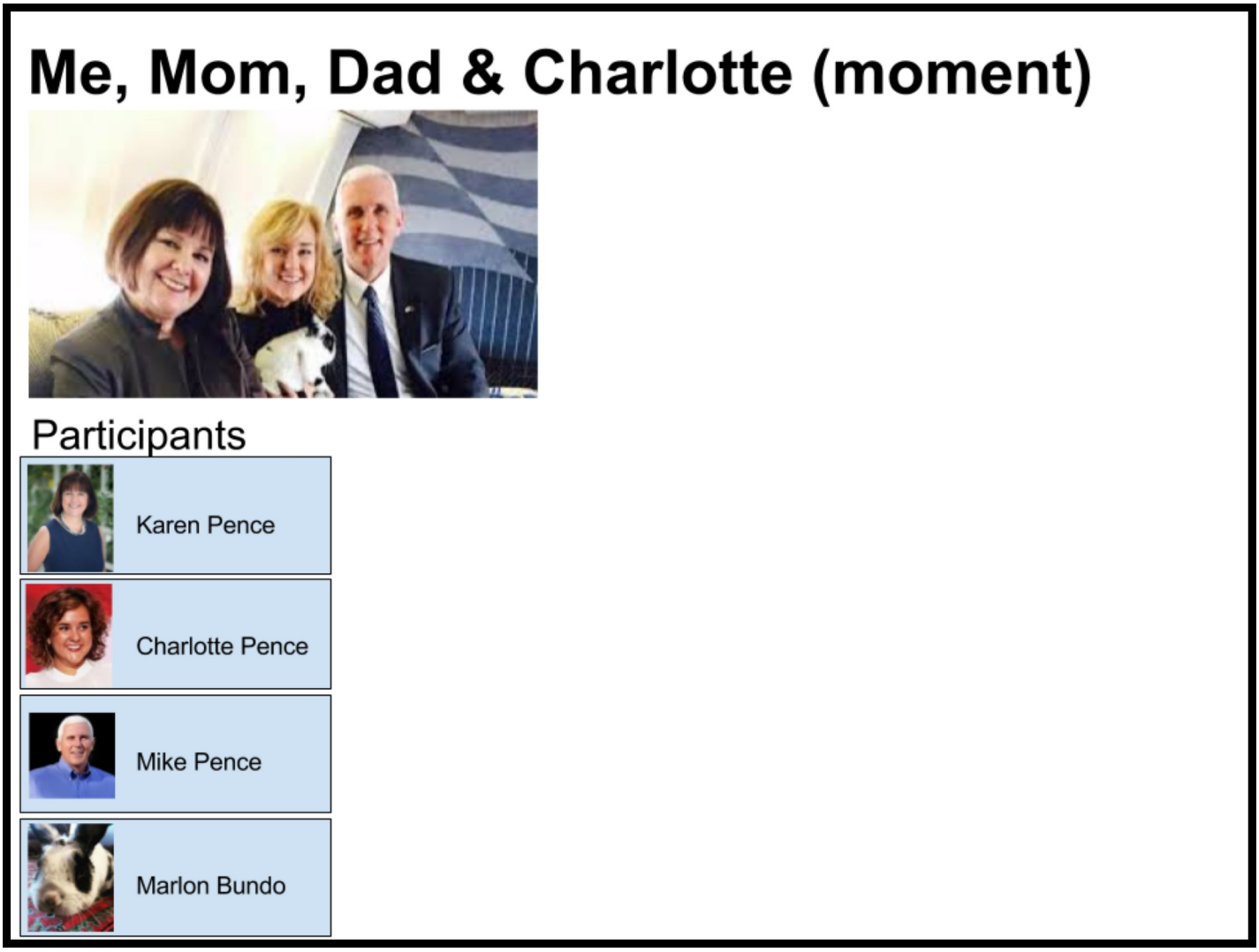


Friend Requests

Over the course of life, sometimes people (and pets) make new friends. In order to make this relationship official on Facebuk, a `FriendRequest` needs to be created, and then it must be approved by both friends. The `FriendRequest` stores both A and B, as well as whether A and/or B has approved the request. Once **both** parties have approved the request (but not before), then A is added to B's list of friends, and B is also added to A's list of friends. (Facebuk requires that both friends approve it because Facebuk itself might suggest two people to become friends.)

Moments

In addition to linking people, pets, and possessions, Facebuk also enables users to capture **moments** in which multiple **people** and **pets** -- but **not** possessions -- participated and are captured in a photograph. (Note that just because two participants are both in the same moment does not imply that they are friends.) See one of the Pence family's moments below, in which a list of the participants is displayed below the photo:



Happiness

Facebuk wants its users to be happy (isn't that nice?). For this reason, Facebuk provides its users with the ability to tag each moment with how happy each participant -- which can be either a person or a pet -- appears during that moment's image. These "happiness" tags could be provided by some sort of automatic happiness or smile detector (as we will discussed in class). However, in this project, for simplicity, they will just be input manually.

Using this "happiness" information, Facebuk can offer its users two special search functions:

1. For a specific person *p*, find the friend *f* -- who can be either a `Person` or a `Pet` -- with whom person *p* appears the **most happy on average**, over all the moments in which both *p* and her/his friend *f* participate. If person *p* has no friends with whom she/he appears in the same moment, then return `null`.
2. For a specific person *p*, find the moment *m* (in which *p* participated) in which the average happiness value -- over all participants in *m* (including both people and pets) -- is highest. If the person does not participate in any moments, then return `null`.

By "average" we mean the arithmetic mean.

Going Out with Friends

Sometimes it's nice to hang out with friends...especially when everybody in the group is friends with everybody else. What is the **largest** set of friends you can go to the movies with such that **everybody** in the set is friends with everybody else? In computer science and graph theory, this kind of set is called a **maximum clique** (which is different from a *maximal* clique). It isn't always easy to determine, but fortunately Facebuk offers a "Maximum Clique Finder" service that gives you the answer. As an example, suppose:

- **Melania**'s friends are: Ivana Trump, Kevin Hart, Hillary Trump, and Marlon Bundo.
- **Marlon**'s friends are: Melania Trump, Charlotte Pence, Kevin Hart, and Humphrey.
- **Ivana**'s friends are: Melania Trump, Tom Cruise, Hillary Clinton, and Kevin Hart.
- **Kevin**'s friends are: Melania Trump, Marlon Bundo, Robin Wright, Hillary Clinton, Humphrey, and Ivana Trump.
- **Hillary**'s friends are: Melania Trump, Ivana Trump, Kevin Hart, and Robin Wright.

In this case, the largest set -- i.e., the maximum clique -- of friends with whom Melania could go out is: Ivana, Kevin, and Hillary. Marlon cannot go because he is not friends with Hillary or Ivana :-(.

Requirements

This project consists of both *data modeling* and *algorithmic* challenges:

R1: Data modeling

Your job is to implement a set of classes to support the fundamental objects that constitute Facebuk, as well as the relationships that connect the different objects together. In particular, you must support the creation of objects of type:

- `Person`: Each person can have pets, friends (both people and pets), and possessions. Also, she/he can participate in moments.
- `Pet`: Each pet has a sole (i.e., just one) owner (who must be a person) and can have friends (both people and pets).
- `Possession`: Each possesion has a sole owner (who must be a person), and it has a price; it has no friends :-(.
- `Moment`: Each moment is associated with a list of participants along with their respective *happiness* values.

In addition, **all** objects have a name and an image. To keep things simple, we pretend to "load" an `Image` from disk by specifying a hypothetical filename, e.g., `new Image("Barack.png")`.

No matter how you design your set of classes, you are required to include the following constructors:

- `Person (String name, Image image)`
- `Pet (String name, Image image)`
- `Possession (String name, Image image, float price)`
- `Moment (String name, Image image, ArrayList participants, ArrayList smileValues)`, in which the array of participants should contain objects of either type `Person` or `Pet`, and the array `smileValues` should contain objects of type `Float` that correspond to the levels of happiness of each person/pet.

In addition, implement the following methods, which should be callable on objects of type `Person` **or** `Pet`:

- `void setFriends (ArrayList friends)` -- sets the list of friends for the target person/pet
- `void setMoments (ArrayList moments)` -- sets the list of moments for the target person/pet
- `ArrayList getFriends ()` -- returns an `ArrayList` of the person's/pet's friends
- `void addFriend (... friend)` -- adds a new friend to the list of friends of the target object. (Note that `...` is not real Java code -- you will need to change this to whatever type is appropriate given your implementation. Also, for simplicity, you do not need to check whether the specified friend was already a friend of the target object.)

The `FriendRequest` class should have the following methods:

- `FriendRequest (... entity1, ... entity2)` -- constructor that specifies the two "entities" (person or pet) of the friend request.
- `void approve (... entity)` -- called when the specified entity approves the request; once both entities of the request have approved it, then entity1 and entity2 should be added to their respective lists of friends. If the specified entity was not one of the two entities specified during construction, then this method should throw an `IllegalArgumentException`.

For `Person` but **not** `Pet`, implement:

- `void setPossessions (ArrayList possessions)` -- sets the list of possessions for the target person
- `void setPets (ArrayList pets)` -- sets the list of pets for the target person

For `Possession` and `Pet`, implement:

- `void setOwner (Person owner)` -- sets the sole owner for the target possession/pet

For `Possession`, implement:

- `float getPrice ()` -- returns the price of the possession

Finally, you should implement an `equals` method that is callable on objects of **all four main classes** (`Person`, `Pet`, `Possession`, and `Moment`):

- `boolean equals (Object o)` -- returns true if and only if the **name** of the target object is equal to the name of the specified object `o`.

In addition to writing Java code that models the objects and relationships described above, you also need to describe your *rationale* for how and why you created the classes that you chose. In particular, in a `README` file, for **every** class (abstract or concrete) that you use, describe **why** you use it and **what** purpose it serves within the greater project. Since `Person`, `Possession`, `Pet`, and `Moment` are all required, you do not need to justify their existence.

Design goals

The goal of this part of the assignment is to use classes (concrete or abstract) so as to (1) **minimize code redundancy** between different components (people, pets, etc.) of the project. At the same time, you need to (2) make sure that every class represents a **coherent** and **meaningful** entity -- it should not just be a "grab-bag" of methods. Both (1) and (2) are tenets of good object-oriented design.

`ArrayList`

In this project you will be using the ArrayList class. `ArrayList` is a data structure used to manage a **variable-**

length list -- similar in purpose to an array, but you don't have to resize it every time the array reaches capacity. `ArrayList` is an extremely useful class in many applications, and we will talk about several other useful data structures over the coming 1-2 weeks.

The proper way of using an `ArrayList` is to specify the **type** of object you want to store in it when you declare and instantiate it. **However**, this gets surprisingly complicated when you want to store objects that can derive from a hierarchy of classes and/or implement one or more interfaces (which we will discuss in class soon). We will cover this later in this course. **For now**, you should use `ArrayList` in a "type-unsafe" fashion: just declare, instantiate, and use it in the following way:

```
ArrayList list = new ArrayList();
list.add(person);
```

where `person` is an object of type `Person`. Because this is "type-unsafe", you will get a compiler warning:

```
make Note: FacebukTester.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

It's ok to ignore this warning for this assignment.

To later retrieve an object from the array, use: `list.get(idx);` where `idx` is the index of the element you want to retrieve. Note that, when you use `ArrayList` in this manner, the return value of `list.get(idx)` will be an `Object`; hence, to cast it back to a `Person`, use: `Person p = (Person) list.get(idx);` The cast will become unnecessary once we talk about *generic types* later in the course.

R2: Happiness computation

Your code should implement the following methods that can be called on objects of type `Person` **or** `Pet`:

- `getFriendWithWhomIAmHappiest ()` which returns either a `Pet` or a `Person` (hence, you will need to choose the return type to be sufficiently flexible)
- `getOverallHappiestMoment ()` which returns `Moment`

For a description of what these should do, see above section on happiness. We will test your code automatically and/or by manual inspection.

R3: Finding a Maximum Clique of Friends

Implement the following methods, both which can be called on objects of type `Person` **or** `Pet`:

- `ArrayList findMaximumCliqueOfFriends ()` -- return a list containing a maximum clique of friends with whom the target person/pet could go out, such that each of his/her friends is also friends with everyone else in the set.
- `static boolean isClique (ArrayList set)` -- returns true if and only if **all** the people/pets in the specified `set` are **all** friends with each other. (Note: if `set` is empty, then this method should return true since there is no one in the set who is *not* friends with another member of the set.) Since this is a helper method that should not depend on any instance variables of the target object, it can (and should) be declared `static`. Even though this is a helper method, **make sure it is declared `public` so we can test it!**

Details:

- The set can contain both people and pets.
- The set should not contain the target person/pet on which the method was called.
- If there are multiple such "largest sets" (aka maximum cliques) for the target person/pet, then it's fine to return any of them.
- The set of friends can be stored in the `ArrayList` in any order.

Teamwork

You may work as a team (2 people) on this project.

Getting started

Download this zip file that contains `FacebukTester.java` and `Image.java`. You should not need to modify the `Image` class, but you likely will want to add more tests to the tester.

Next, make sure you have downloaded and installed JUnit. In CS 2103, we will use version 4.12. There are **two** Jar files you need to download:

- junit-4.12.jar
- hamcrest-core-1.3.jar

See here for more information.

On the data modeling part of the project, consider what pets and people have in common, and then consider how they differ. "Factor out" the common functionality into a common (abstract) superclass (e.g., `LivingEntity` that can represent both pets and people).

Unit testing

We have given you starter code (`FacebukTester.java`) to help you with testing. We will use a larger version of this file to test your code; hence, **it is absolutely imperative that your code can compile against this tester**. If your code does not compile against our test code, you may unfortunately get 0 credit.

Grading

Grading will be based on (1) your code compiling against our test code; (2) your written description (in the `README` you write) and justification of the design choices you made; (3) your code passing our test cases; and (4) your adherence to consistent and appropriate style. In particular, make sure your code adheres to the style guidelines given during the first week of class. You additionally need to make sure that your use of **whitespace** is consistent -- no randomly ragged indentation is permitted.

In order to receive full-credit on the data modeling portion of this project, you need to make sure that you have used classes (concrete or abstract) so that (1) they represent coherent, meaningful objects in the Facebuk ecosystem you are modeling, and (2) so as to minimize code redundancy.

What to Submit

Create a Zip file containing all of your `.java` files (no class files), as well as a `README` file describing your object-oriented design rationale. (**Important note**: The file must be a Zip file, **not** a `gz`, `rar`, or `7z` file!) Submit the Zip file you created to InstructAssist. **Submission deadline**: Tuesday, October 30, at 11:59pm EDT.