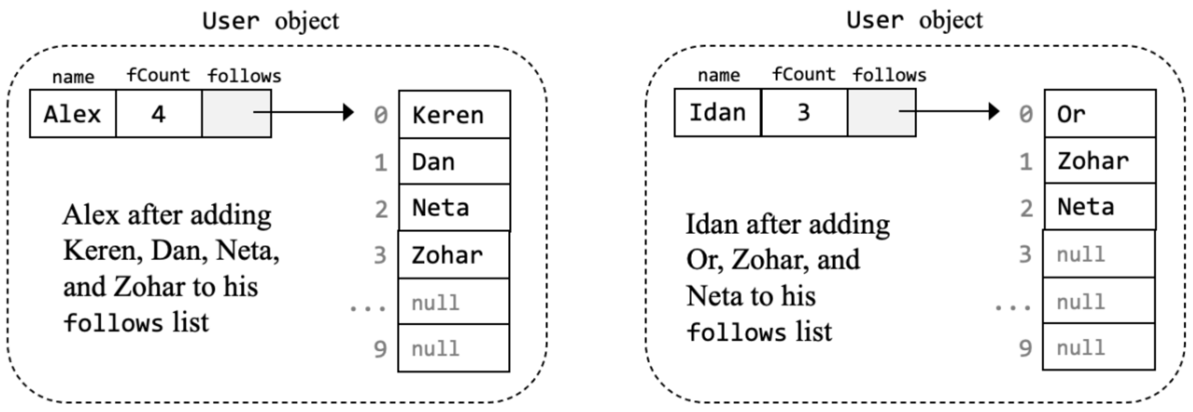
Homework 8: Social Network

In lectures 8-2 and 9-1 we began discussing *Object Oriented Programming*, introducing a new terminology and many new concepts. This exercise practices everything that was discussed in these lectures, and more. After completing it, you will have a solid grasp of object-oriented programming. The exercise implements a social network, using a more realistic architecture than the one used in the midterm exam.

Users

Each user in the network is represented as an instance of a `User` class. The `User` class has three *fields*: the user's `name` (a string), a `follows` array, representing a maximal number of names that the user can follow (each name is a string), and `fCount`, an `int` variable that stores the *actual* number of names that the user follows. Below are examples of two typical `User` objects:



Because arrays have a fixed size, we must decide in advance how many names each user can potentially follow. In this example we decide that `maxfCount` is 10, implying that each user can potentially follow at most 10 names. Here is the beginning of the `User` class declaration, specifying how `User` objects are represented, and how they are constructed:

```
/** Represents a user in a social network. A user is characterized by a name, a list of
 *   user names that the user follows, and the list's size. */
public class User {

    // Maximum number of users that a user can follow
    static int maxfCount = 10;

    private String name;        // name of this user
    private String[] follows;   // array of user names that this user follows
    private int fCount;         // actual number of followees (must be <= maxfCount)

    /** Creates a user with an empty list of followees. */
    public User(String name) {
        this.name = name;
        follows = new String[maxfCount]; // fixed-size array for storing followees
        fCount = 0;                       // initial number of followees
    }
// More User methods come here...
}  // End of the User class declaration
```

The requirement that each user follows at most 10 names sounds like an arbitrary and inflexible restriction. Indeed it is, and soon in the course we will learn how to overcome this limitation.

Back to our network architecture: To add a new user to the network, we construct a new `User` object. According to the rules of the game, the new object will include a fixed-sized array of 10 strings. Each element in the array will be automatically initialized (by Java) to the value `null`. The new `User` object will also include the number of names that the user *actually* follows, which is initialized to 0. Notice that `name`, `fCount`, and `follows` are all *fields*, also known as *private variables* (the fact that one of these fields is an array is not a problem at all).

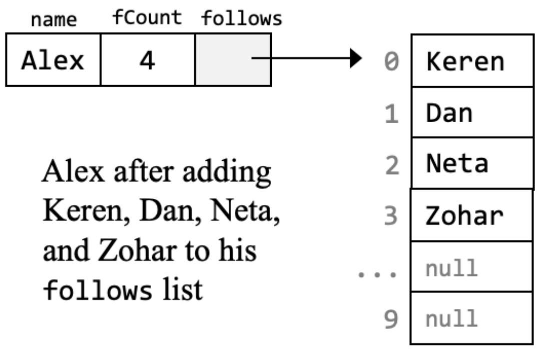
Below is the client code that created the two users shown in the previous page:

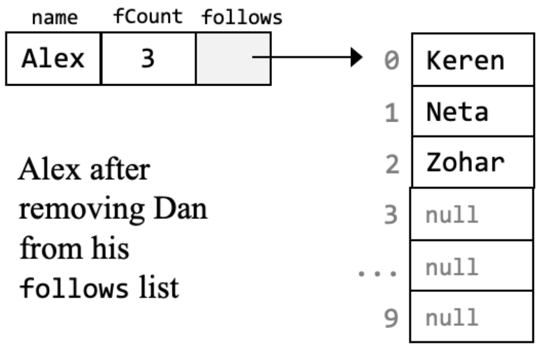
```
// Creates a new User
User alex = new User("Alex");
System.out.println(alex);     // Outputs: Alex ->
// Adds followees (names that the user follows)
alex.addFollowee("Keren");
alex.addFollowee("Dan");
alex.addFollowee("Neta");
alex.addFollowee("Zohar");
System.out.println(alex);     // Outputs: Alex -> Keren Dan Neta Zohar
// Creates a new User
User idan = new User("Idan");
System.out.println(idan);     // Outputs: Idan ->
// Adds followees
alex.addFollowee("Or");
alex.addFollowee("Zohar");
alex.addFollowee("Neta");
System.out.println(alex);     // Outputs: Idan -> Or Zohar Neta
...
```

The `toString()` method that produced the outputs shown above is part of the `User` class. Here is the code of this method, which is given:

```
/** Returns this user's name, and the names that s/he follows (part of the User class). */
public String toString() {
    String ans = name + " -> ";
    for (int i = 0; i < fCount; i++) {
        ans = ans + follows[i] + " ";
    }
    return ans;
}
```

Adding and removing followees: What happens when a user decides to start, or stop, following another user? To add a new followee, we store the followee's name in the next available element in the `follows` array. Importantly, we also have to remember to do `fCount++`. As a result of this storage strategy, the names of the followees are always stored one after the other, forming a contiguous block within the array. How to remove a name from the user's `follows` list? In that case we move all the array elements below this name one position "up". And, we must remember doing `fCount--`. The top of the next page depicts this technique.


name
fCount
follows
Alex
4
0
1
2
3
...
9
Keren
Dan
Neta
Zohar
null
null
Alex after adding Keren, Dan, Neta, and Zohar to his follows list



Notice, once again, that the names of the followees must always form a contiguous block within the `follows` array. That's why we have to work hard to "close the gap" whenever a name is removed from the list.

Implementing the User class

The skeletal `User` class is given, along with a complete `UserTest` class that tests it. Start by reviewing the code of the `User` class, following the guidelines below.

`The User(String,boolean) constructor:` This overloaded constructor creates a user and then fills its `follows` list with a few dummy names. This constructor is mostly useless, having one purpose only: Allowing testing the `toString` and `follows` methods before implementing anything else. Read and understand the constructor's code. Constructor overloading and the `this()` call are explained in Lecture 9-1, slide 30.

`getName`, `getFollows`, `getCount`: The code of these trivial getter methods is given, so there is no need to implement them. These getter methods are not used at all in the `User` class, so for now you can ignore them.

Getting started: Read the `UserTest` class code, up to, and not including, the "serious testing" part. Then compile the `User` class and the `UserTest` class, and execute `UserTest`. Notice that at this stage, only the beginning of the test makes sense; The "serious testing" produces mostly empty results, since it calls skeletal methods that do nothing.

`The toString()` method is the first method called by the `UserTest` class (it is called implicitly, when the statement `System.out.println(dummy)` is executed). Read and understand the `UserTest` code that creates and prints the dummy user. The `toString` method is given, so you don't have to implement it.

Comparing strings: To check if two string objects have the same contents, use the boolean method call `str1.equals(str2)`. This general practice must be used whenever you have to check the equality of two strings, throughout this exercise, and other exercises.

Implement the `User` class methods, in the order in which they appear below.

`follows`: This method searches a given string in an array of string values. It's a relatively simple method, so it's a good way to get you started. Implementation tips:(1) Suppose that `arr` is an array of `String` values. Following the advice just given: To compare the array's `i`-th element to some string `str`, use the method call `arr[i].equals(str)`. (2) The fact that the method name (`follows`) is the same as the name of the searched array (`follows`) is meaningless. It simply makes things more readable. (3) Read the beginning of the `UserTest` class and understand the code that creates the dummy user and tests the `toString` and `follows` methods.

`addFollowee`: The method's operation is described in its API documentation. In addition to the documented actions, the method displays relevant feedback messages, which are important for debugging purposes. These messages are not documented; To figure out what to print after each action, see the relevant outputs of the `UserTest` class. Implementation tip: In general, we cannot allow following "names" (arbitrary strings). Rather, we must first verify that the network includes users that have such names. We will worry about this later, when we implement the `Network` class. For now, don't worry about these validity tests, and happily add any given name.

`removeFollowee`: Everything that was said above about `addFollowee` is also applicable to this method. The main difference is that here we have to implement the "closing the gap" logic described earlier. Implementation tips: (1) Don't use the `follows` method; you will end up iterating the array twice, which makes no sense. Instead, use a single loop to search the `follows` array; when you'll reach the name that has to be removed, you will also have the `i`-th location at which the "closing the gap" logic must start. (2) After ending the "closing the gap" logic, set the array element that is no longer relevant to `null`. For example, in the figure shown above, notice that after closing the gap, `follows[3]` was set to `null`. In general: When an object (and strings are objects) is no longer needed, set the variable that refers (points) to it to `null`.

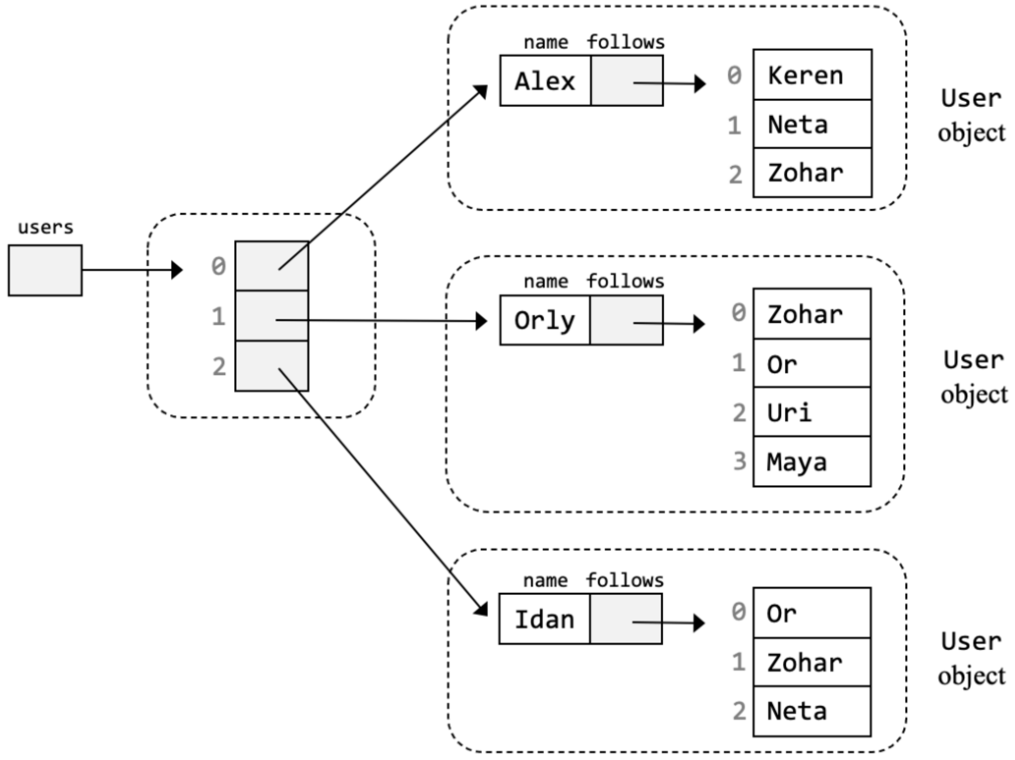
`countMutual`, **`isFriendOf`**: These methods are self-descriptive. Read their documentation, read carefully and understand the `UserTest` code that tests them, and implement them.

* * *

The homework description continues on the next page

Network

A social network consists of users who follow each other. In the midterm exam we described these relationships using a matrix. In this exercise we use a more realistic architecture. Here is an example of a social network that uses this architecture:



To create such a network, we use the following client code:

```
// Creates a network with a maximum capacity of 1000 users
Network net = new Network(1000);

// Adds Users and follows relationships
net.addUser("Alex");
net.addUser("Orly");
net.addUser("Idan");
net.addUser("Keren");
net.addUser("Neta");
...
net.addFollowee("Alex", "Keren");
net.addFollowee("Alex", "Neta");
net.addFollowee("Alex", "Zohar");
net.addFollowee("Orly", "Zohar");
...
```

`The Network fields and constructors`: The `users` list of this network is implemented similarly to the `follows` arrays implementation, with one difference: Instead of fixing the array size for all possible social networks, we allow the code that creates a new network to determine the maximal number of users in the network. This logic is implemented by the `Network` constructors. Similarly to the `User` class, we provide a constructor for creating a dummy network. This constructor is used only for testing purposes.

Implementing the Network class

Getting started: The skeletal `Network` class is given, along with a complete `NetworkTest` class that tests it. Start by reviewing the code of these classes. Then compile the two classes, and execute `NetworkTest`. At this stage the tests will produce mostly empty results, since they call skeletal methods that do nothing.

The `toString()` method is the first method called by the `NetworkTest`. Read the code that creates and prints the dummy network, and understand it. Then implement the `toString` method. Since the users in the dummy network follow no one, expect to see this output:

```
Network:
Foo ->
Bar ->
Baz ->
```

Tip: In the implementation of the `toString`() method of the `Network` class, have each user print itself. This is the spirit of object-oriented programming: Objects should take care of themselves.

Implement the rest of the `Network` class methods, in the order in which they appear below.

`getUser`: Read the beginning of the `NetworkTest` class, and understand the code that tests the `getUser` method (the "ternary if" used in this test is explained in Lecture 2-1, slide 14). Implementation tip: The logic of this method is similar to that of the `User` class's `follows` method, with one major difference: `follows` returns a boolean value; `getUser` returns a `User` object, or `null`.

`addUser`: The logic of this method is similar to that of the `User` class's `addFollowee` method.

`addFollowee`: This method gets two parameters: `name1` and `name2`. Implementation tips: Call the `User` class's `addFollowee(name2)` on the user whose name is `name1`. Notice that at least three things can go wrong: `name1` is not a user in this network, `name2` is not a user in this network, or trying to make the former follow the latter fails.

`recommendWhoToFollow`: This method iterates through all the users in the network, searching for a most recommended user to follow. Implementation tips: (1) Since we have to return a `User` object, you can get started with a statement like `User mostRecommendedUserToFollow = null;` (2) The method processes each user in the network, except for the user on which it was called. The processing of this user must be skipped. How to skip an iteration in a loop? One way to do it is using Java's `continue` statement. This statement is described here, along with a nice usage example.

`mostPopularUser`: Implementation tip: Start by implementing and testing the helper method `followeeCount`. Then use it for implementing the `mostPopularUser` method.

Endnote

If this homework assignment will make you sick each time you see a social network again, then, Mazal Tov. There are better things to do in life than following social networks, like climbing mountains and reading books.