BBM 104 – Introduction to Programming II Lab.
Programming Assignment 3 – Object-Oriented Programming with Java
Developing Mini *Facebook* system – a prototype[1]

Developed for the Spring 2017 term by Research Assistants Selman Bozkır, Selim Yılmaz and Selma Dilek

Submission:

- **Project Folder Zipped as b<StudentID>.zip**
- **Deadline: 28.04.2017 23:59**

Table of Contents

1 INTRODUCTION

Object-oriented programming (OOP) is more than just adding a few new features to programming languages. It is rather a new *way of thinking* about decomposing and modeling problems with less complexity and more code reuse when

[1] **Covered subjects**: OOP Basics, Abstraction, Encapsulation, Inheritance, Polymorphism, Abstract Class, Interfaces, Java Collections, UML class diagrams, Javadoc

developing programming solutions. In OOP, a program is viewed as a collection of loosely connected objects, each of which is responsible for specific tasks. It is through the interaction of these objects that computation proceeds and the model works as a whole.

In this assignment, you will work with the concepts of OOP in order to practice them and observe their advantages. By the end of this assignment, you will have learnt the concepts of relationships among classes, encapsulation, abstraction, inheritance and polymorphism, as well as how to and why use Javadoc in your Java projects.

1.1 LEARNING OBJECTIVES

The learning objective of this programming assignment is to let students practice the following concepts of Java programming language in particular (and OOP in general):

- Object-oriented programming
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Abstract Class
- Java Interface
- Java Collections
- UML class diagrams
- Javadoc

1.2 MARKING SCHEME

The marking scheme is shown below. Carefully review each mark component.

Component	Mark %
Creation of Base Classes with the requested class structure (the main class of your application must be named **`Main.java`**)	35%
Successful instantiation of all initial user objects from the input file	10%
Correct execution of scenarios as requested in the input file with commands	40%
Drawing of UML class diagram saved as **`uml.jpg`**	5%
Javadoc that gives a detailed description of the classes in **`javadoc`** folder	10%
Total	**100%**

Note: In your solutions, your algorithm and code should be as simple as possible.

2 USEFUL INFORMATION

In this section you can find some useful beginner's level information that you will need for this project. For more information on each subject you need to do additional research and consult other resources (e.g. lecture notes, textbook, Internet resources, etc.).

2.1 ABSTRACTION

Abstraction is a general concept that denotes the progress of modeling "real things" into programming language. For example, when you write a class named Person, you abstract a real person into a type (class). In OOP, abstraction is a process of hiding the implementation details from the user, and only providing the functionality. In other words, the user will have the information on what the object does instead of how it does it.

In Java, the process of abstraction is done using interfaces, classes, abstract classes, fields, methods and variables. It is the fundamental concept on which other things rely on such as encapsulation, inheritance and polymorphism.

2.1.1 Abstract Class

In Java, a class which contains the **`abstract`** keyword in its declaration is known as abstract class. Abstract classes may or may not contain abstract methods, i.e., methods without body (e.g. `public void get();`). If a class has at least one abstract method, then the class must be declared abstract.

Abstract classes **cannot be instantiated**. To use an abstract class, you have to inherit it from another class, and provide implementations to the abstract methods in it. If you inherit an abstract class, you have to provide implementations to all the abstract methods in it. Properties of an abstract class are inherited just like the properties of any inherited concrete class.

2.1.2 Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract method using the **`abstract`** keyword (e.g. `public abstract int compute();`). An abstract method contains a method signature, but no method body. Instead of curly braces, an abstract method will have a semicolon (;) at the end.

Declaring a method as abstract has two consequences:

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Finally, a child class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

2.2 JAVA INTERFACE

An interface is a reference type in Java. It is similar to a class. Think of it as a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. A class describes the attributes and behaviors of an object, but an interface contains behaviors that a class implements. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a .java extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a .class file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

An interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

To declare an interface, the keyword **`interface`** is used. A class uses the **`implements`** keyword to implement an interface. When a class implements an interface, that class agrees to implement the specific behaviors of the interface. If a class does not implement all the behaviors of the interface, the class must declare itself as abstract.

An interface can extend another interface in the same way that a class can extend another class. The **`extends`** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface. Unlike classes, an interface can extend more than one parent interface. The **extends** keyword is used once, and the parent interfaces are declared in a comma-separated list.

2.3 ENCAPSULATION

Encapsulation is one of the fundamental OOP concepts. It is a technique of wrapping (packaging) the related data (attributes) and behavior (code - methods) together into a single unit. It also provides a way to hide and control data by protecting it from misuse by the outside world. This is achieved by making data members private, and using public helper methods through which we can decide the level of access we want to provide (e.g. read-only, write-only). A fully encapsulated Java class is a class whose all variables are private.

The advantages of encapsulation include: flexibility in modifying code in case requirements change, reusability of encapsulated code throughout multiple applications if necessary, and reducing the time of maintenance (update) process.

2.3.1 Visibility Modifiers

In Java, there are four access modifiers which provide various access levels: **private** (visible inside of the class only), **default/package** (visible inside of the package), **protected** (visible inside the package and to all subclasses) and **public** (visible to everyone).

2.4 INHERITANCE

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another class. It is achieved with the keyword `extends`. The class which inherits the properties of another class is known as subclass (derived class, child class), and the class whose properties are inherited is known as superclass (base class, parent class).

Inheritance enables managing information in a hierarchical order, code reuse, extending a class by adding new features instead of writing a new class from scratch, and maintainability of the code (e.g. updating the code in one place – superclass, instead of updating every single class).

2.5 POLYMORPHISM

Polymorphism is the ability of an object to take on many forms. In OOP, polymorphism means a type can point to different object at different time. In other words, the actual object to which a reference type refers, can be determined at runtime. Any Java object that can pass more than one IS-A test is considered to be polymorphic. Thus, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. A reference variable can refer to any object of its declared type or any subtype of its declared type.

Polymorphism increases code reuse, allows for flexibility in running code and code extensibility.

In Java, polymorphism is based on inheritance and overriding.

2.5.1 Method Overriding

When a class extends another class, it can use its super class's methods. However, sometimes the subclass may need a different behavior of a method provided by its superclass. The method implementation in the subclass overrides (replaces) the method implementation in the superclass. In this case, the subclass method and superclass method have the same name, parameters and return type, but different implementation. This is called method overriding.

2.6 METHOD OVERLOADING

Method overloading should not be confused with method overriding. When we need to have more than one method with the same functionality within the same class, we don't have to declare new methods with different names for each one. Method overloading feature allows us to declare multiple methods with the same name, but different signatures

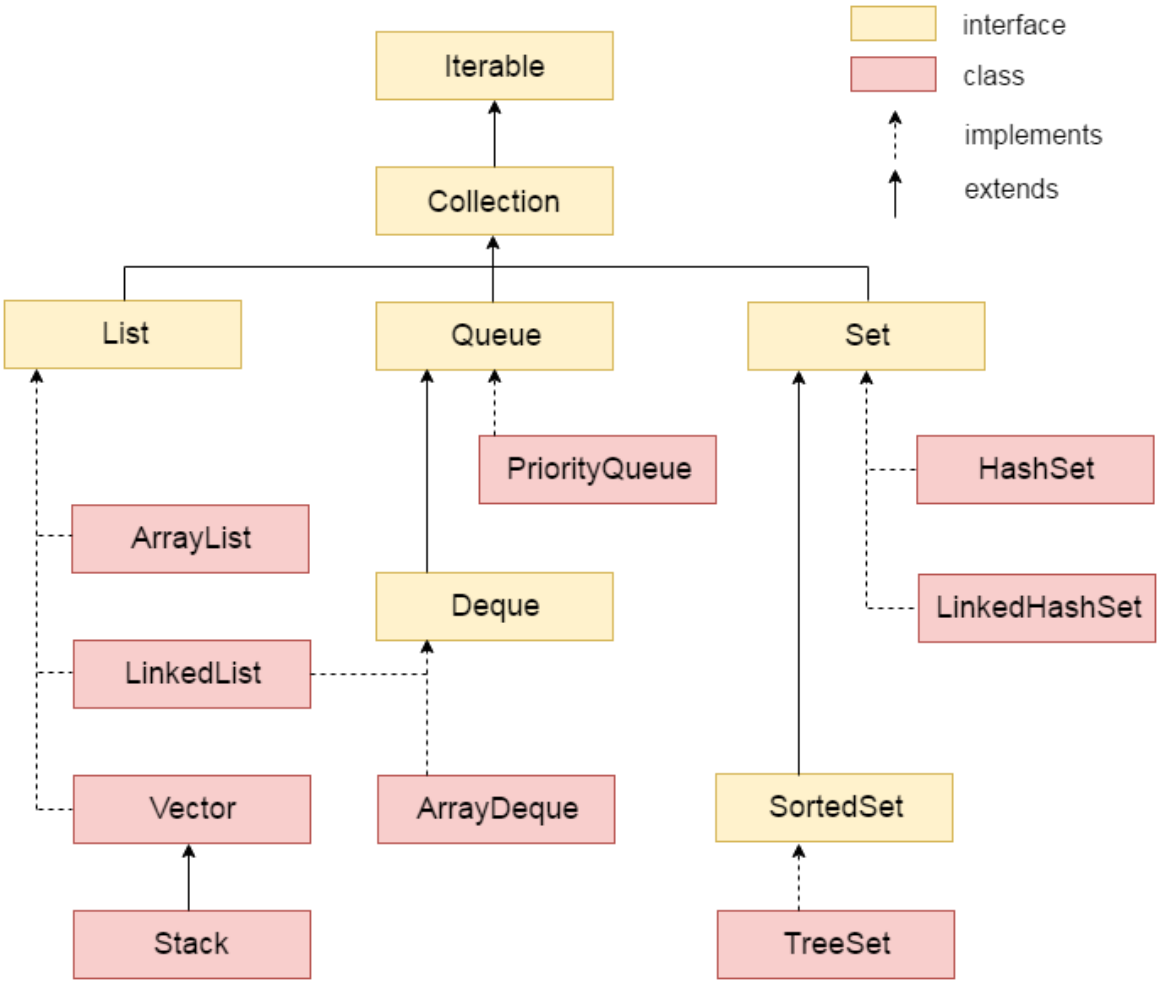
(e.g. different argument list, types or order). `System.out.println()` is an example of method overloading in Java. It takes float, int, double or String types as arguments.

2.7 JAVA COLLECTIONS

Collections in Java is a framework that provides an architecture to store and manipulate a group of objects. A Collection represents a single unit of objects i.e. a group. All the operations that you perform on data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

Java Collection simply means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque etc.) and a set of standard collection classes that implement Collection interfaces (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet etc).

Hierarchy of Java Collection Framework:



2.8 UML CLASS DIAGRAMS

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, which is intended to provide a standard way to visualize the design of a system. A UML class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects. The class diagram is the main building block of object-oriented modeling.

In a UML diagram, classes are represented with boxes that contain three compartments:

- The top compartment contains the name of the class in bold and centered, and the first letter is capitalized.
- The middle compartment contains the attributes of the class. They are left-aligned and the first letter is lowercase.

- The bottom compartment contains the operations the class can execute. They are also left-aligned and the first letter is lowercase.

To specify the visibility of a class member (i.e. any attribute or method), these notations must be placed before the member's name:

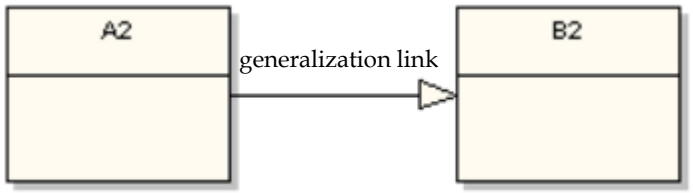
+	Public
-	Private
#	Protected
/	Derived (can be combined with one of the others)
~	Package

2.8.1 UML Class Relations

A relationship is a general term covering the specific types of logical connections found on class and object diagrams. UML defines several relationships, and two of them are Association and Generalization (Inheritance). In this assignment, you are expected to show generalization, realization, and association relationships between your classes.

2.8.1.1 Generalization & Realization

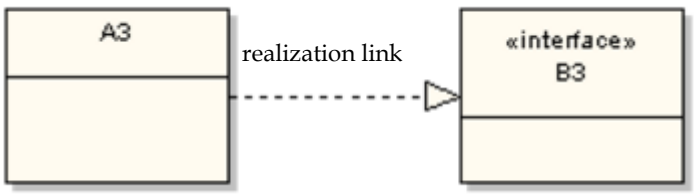
Generalization defines a taxonomic relationship between two classifiers: a more general one and a more specific one (remember inheritance). For example, : when one class groups common data and behavior of other classes, this is called generalization. It is shown as a line with a hollow triangle that connects two classifiers. In contrast to generalization, *specialization* means creating new subclasses from an existing class. If it turns out that certain attributes, associations, or methods only apply to some of the objects of the class, a subclass can be created. The most inclusive class in a generalization/specialization is called the superclass and is generally located at the top of the diagram. The more specific classes are called subclasses and are generally placed below the superclass. A UML representation of generalization is given below:




```
public class A2 extends B2 - B2 is a generalization for A2
```

Realization, on the other hand, is a specialized abstraction relationship between two sets of model elements, one representing a specification and the other representing an implementation. A Realization dependency is shown as a dashed line (see UML diagram fragment below) with a triangular arrowhead at the end that corresponds to the realized element.

This diagram fragment given below states that class A3 implements or *realizes* the interface defined by B3.

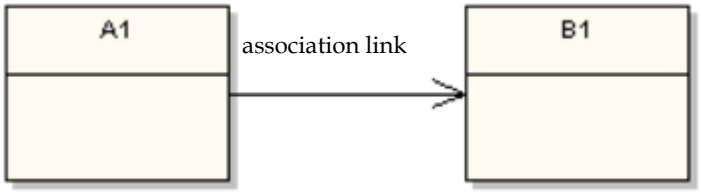



```
public class A3 implements B3 - A3 realizes behavior specified by B3
```

2.8.1.2 Association

Association is a relationship between classifiers which is used to show that instances of classifiers could be either linked to each other or combined logically or physically into some aggregation. UML illustration indicating association between two classes is given below.

The arrowhead means that there is a one-way relationship. In this example it means that class A1 is associated with class B1. In other words, class A1 uses and contains one instance of class B1, but B1 does not know about or contain any instances of class A1. This example manifests itself as the following Java code:




```
public class A1 {
  private B1 b1;
  public B1 getB1() {
    return b1;
  }
}
```

For this assignment, you are expected to draw a basic UML diagram for your classes, which will include class names, all attributes and methods with the specified visibility, and show all relationships between the classes. You may use any tool for this task. For example, you can look up available UML generator tools for eclipse at https://marketplace.eclipse.org/.

2.9 JAVADOC

Javadoc is a documentation generator for Java, which Generates HTML pages of API documentation from Java source files. The basic structure of writing document comments is to embed them inside `/** ... */`. You need to submit Javadoc for your code as well. For details refer to:

http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

3 AN OVERVIEW OF THE APPLICATION

In this programming assignment, you will develop a very simplified version of *Facebook*, well-known social media platform. We will call it Mini-Facebook. Note that Programming Assignments 3 and 4 are consecutive and that this assignment is the first part and basis for the Assignment 4.

3.1 PROBLEM DEFINITION

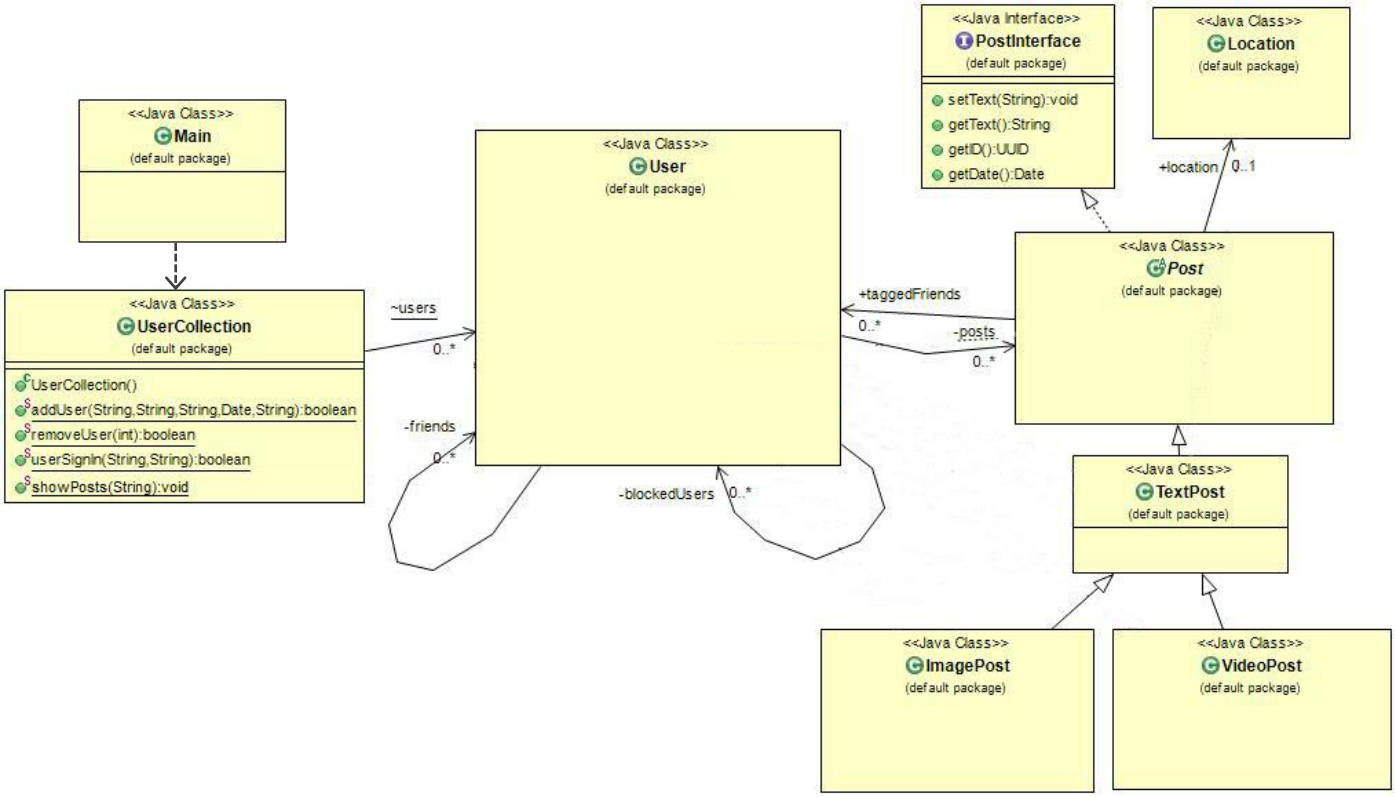
First, we need to review the requirements (specifications) of the system we want to develop. The basic components of Mini-Facebook application include:

- **users** (people with a Mini-Facebook account),
- **posts** (classified according to a post type),
- **user collection** (a class that adds or removes users, lists users' posts, and controls user log-ins).

You will be given an input file that contains information necessary for instantiating the initial users of the system. Furthermore, another input file containing commands to test various execution scenarios will be provided, which you will use to test your application.

The necessary components of the system are explained in detail in the following sections.

A simplified UML class diagram of our Mini-Facebook application is shown in the figure below:



3.1.1 Users

Users are people who have an account on our Mini-Facebook system. Users will not have different roles (e.g. admin or system manager vs. other users); that is, they will all be peers. User's attributes and behaviors are explained below:

- Every **User** in the system has a unique **userID**, a **name**, a unique **username**, a **password**, a **date of birth**, information about **school from which the user graduated**, a **last log-in date**, a **collection of friends**, a **collection of blocked users**, and a **collection of posts**.
- **userIDs** should be assigned as integers in order, starting from 1 for the first user that gets added to the system. These IDs cannot be changed later. A newly added user should never get an ID that has been assigned before to some other user, even in cases when other users get removed from the system.
- Users' dates of birth and last log-in dates should be stored as Date variables.
- All collections must be implemented as Java Collections.
- Users must be able to
  - **Sign in,**
  - **Sign out,**
  - **Update their user profile info (name, date of birth, and school graduated),**
  - **Change their passwords,**
  - **Add friends to their friend lists,**
  - **Remove friends from their friend lists,**
  - **Add posts,**
  - **Remove posts,**
  - **Block users,**
  - **Unblock users,**
  - **List their friends,**
  - **List all users,**
  - **List blocked friends,**
  - **List blocked users.**
- When a user is not logged-in, he/she should be allowed only to perform a sign-in. After user log-in, other actions should be permitted.
- Note that you are required to hide sensitive personal user info and implement setter and getter methods as necessary.

3.1.2 Posts

Posts represent textual or multimedia content shared by users. You are expected to design your post classes in a hierarchical structure. The necessary information about posts is as follows:

- First of all, you need to implement an **interface** which will specify behavior (methods) that the Post class will implement.
- Secondly, you need to implement an **abstract Post class** which will implement the post interface and be a superclass to all other post classes.
- This abstract Post class should define post attributes, implement the methods defined in the interface, and define two additional abstract methods that will **show tagged users** and **show post location**. These methods should be implemented in the immediate subclasses of the Post class.
- Posts are categorized as either **TextPosts**, **ImagePosts**, or **VideoPosts**. But **be careful**: all posts have a textual part (all posts are actually text posts), and can optionally contain image or video (but not both!).
- All posts have a unique **postID** which should be assigned as a random UUID (immutable universally unique identifier), a **text**, a **date** when the post originated, a **location**, and a **collection of tagged friends**.
- **Location** should be implemented as a separate class with the following attributes: **latitude** and **longitude** (stored as double values).
- **TextPost** class should implement abstract methods from the abstract Post class. To show the tagged friends, it is enough to print their names (if any). To show the post location, both latitude and longitude should be printed.
- **ImagePosts** have the **image filename**, and the **image resolution** (width and height in pixels).
- **VideoPosts** have the **video filename**, and a constant attribute that specifies the **maximum video length** in minutes (the maximum allowed video duration is 10 minutes).
- Note that you are required to implement setter and getter methods as necessary.

3.1.3 User Collection

The main class in your program should delegate managing to a class named ***UserCollection*** and call it as necessary. In Assignment 4, you will add a Graphical User Interface to your Mini-Facebook application, and this class will play the role of an intermediary between your backend and frontend. Thus, you need to design it accordingly. ***UserCollection*** class needs to allow for the following actions:

- Keeping track of all users,
- Adding new users to the system (by providing new user information),
- Removing existing users from the system (by providing the userID),
- User sign-in (with username and password),
- Displaying users' posts.

3.1.4 Scenarios of Execution

i. **Add New User**: At the initial stage, your program should retrieve the user information from the file users.txt and add them to the system. Besides, the system should also add new users while running. The program should add a new customer (a functionality of the *UserCollection* class) if the syntax of the command is as provided below:

ADDUSER<TAB>**name**<TAB>**userName**<TAB>**password**<TAB>**dateofBirth**<TAB>**schoolGraduated**

Usage Illustration:

ADDUSER<TAB>**Adnan**<TAB>**adnan**<TAB>**adnan1**<TAB>**01/01/1991**<TAB>**Selcuk Universitesi**
Adnan has been successfully added.

ii. **Remove Existing User**: The system should also remove an existing user (a functionality of the *UserCollection* class). It takes one parameter which indicates

REMOVEUSER<TAB>**userID**

Usage Illustration:

REMOVEUSER<TAB>**1**
User has been successfully removed.
No such user!

iii. **Show Posts**: The system should also be able to display users' posts (a functionality of the *UserCollection* class). The parameter for this action is a username:

SHOWPOSTS<TAB>**userName**

Usage Illustration:

SHOWPOSTS<TAB>**adnan**
adnan does not have any posts yet.
No such user!
********** adnan's Posts ********** This is my 1st text post Date: 14.04.2017 Location: 39.8833431, 32.7381663 ---------- This is my 1st text post Date: 14.04.2017 Location: 39.8833431, 32.7381663 Friends tagged in this post: ahmet, demet, adnan ---------- This is my 1st image post Date: 14.04.2017 Location: 42.8833431, 32.6381533 Image: image.png Image resolution: 135x250 Friends tagged in this post: demet, gizem ---------- This is my 1st video post Date: 14.04.2017 Location: 39.8833431, 32.6381533 Video: myvideo.avi Video duration: 8 minutes Friends tagged in this post: utku ----------

iv. **Sign-In**: This is the base action **for users to perform** before they are allowed to interact with the system and execute other commands. As in a real application, this evaluator takes two parameters that are a **username** and a **password**. The *UserCollection* class should also provide means for a user sign-in given the same parameters.

SIGNIN<TAB>**userName**<TAB>**password**

Usage Illustration:

SIGNIN<TAB>**adnan**<TAB>**adnan1**
You have successfully signed in.
Invalid username or password! Please try again.
No such user!

v. **Sign-Out**: It prevents the user from executing any scenarios other than sign-in. No argument needed to perform this command since it is assumed only one user can log in to the system at a particular time.

SIGNOUT

Usage Illustration:

SIGNOUT
You have successfully signed out.

vi. **Update Profile**: User who has already logged in may change his/her personal information (name, date of birth, and school from which graduated). It takes three arguments and these are assumed to be the updated values.

UPDATEPROFILE<TAB>**name**<TAB>**dateofBirth**<TAB>**schoolGraduated**

Usage Illustration:

UPDATEPROFILE<TAB>**Adnan**<TAB>**07/01/1991**<TAB>**schoolGraduated**
Your user profile has been successfully updated.
Error: Please sign in and try again.

vii. **Change Password**: A user may want to change his/her password. The system should first check if the provided password matches the current password.

CHPASS<TAB>**oldPassword**<TAB>**newPassword**

CHPASS<TAB>**adnan1**<TAB>**adnan123**
Password mismatch! Please, try again.
Error: Please sign in and try again.

viii. **Add New Friend**: Friendship is the most important point in any social media platform and without which probably it would not gain much attention. The system you will develop should allow a user who has signed in to add an existing user as a friend. It takes only one argument, which indicates user name of the user who will be added.

ADDFRIEND<TAB>**userName**

ADDFRIEND<TAB>**ahmet**
ahmet has been successfully added to your friend list.
This user is already in your friend list!
No such user!
Error: Please sign in and try again.

ix. **Remove Friend**: A user can also remove a user from his/her friend list. This operator takes one argument that is a username.

REMOVEFRIEND<TAB>**userName**

REMOVEFRIEND<TAB>**ahmet**
ahmet has been successfully removed from your friend list.
No such friend!
Error: Please sign in and try again.

x. **Add Text Post**: User may create a new text post by specifying the text content, location (longitude & latitude), and username of the friends who will be tagged to this post, each of which is separated by TAB.

ADDPOST-TEXT<TAB>**textContent**<TAB>**longitude**<TAB>**latitude**<TAB>
userName1<:>**userName2**<:>..<:>**userNameN**

ADDPOST-TEXT<TAB>**This is my 1st text post**<TAB>**39.8833431**<TAB>**32.7381663**<TAB>**ahmet:demet:adnan**
The post has been successfully added.
ahmet is not your friend, and will not be tagged! The post has been successfully added.
ahmet is not your friend, and will not be tagged! demet is not your friend, and will not be tagged! The post has been successfully added.
Error: Please sign in and try again.

xi. **Add Image Post**: Not only can a user create text post, but he/she can also create an image or a video post. In this scenario, the image path and resolution information should be provided to the system as well.

ADDPOST-IMAGE<TAB>**textContent**<TAB>**longitude**<TAB>**latitude**<TAB>
<TAB>**userName1**<:>**userName2**<:>..<:>**userNameN**<TAB>**filePath**<TAB>**resolution**

ADDPOST-IMAGE<TAB>**This is my 1st image post**<TAB>**42.8833431**<TAB>**32.6381533**<TAB>**demet**<:>**gizem**<TAB>**image.png**<TAB>**135**<x>**250**
The post has been successfully added.
demet is not your friend, and will not be tagged! The post has been successfully added.
demet is not your friend, and will not be tagged! gizem is not your friend, and will not be tagged! The post has been successfully added.
Error: Please sign in and try again.

xii. **Add Video Post**: User should only provide the file path of the video.

ADDPOST-VIDEO<TAB>**textContent**<TAB>**longitude**<TAB>**latitude**
<TAB>**userName1**<:>**userName2**<:>..<:>**userNameN**<TAB>**filePath**<TAB>**videoDuration**

ADDPOST-VIDEO<TAB>**This is my 1st video post**<TAB>**39.8833431**<TAB>**32.6381533**<TAB>**utku**<:>**gizem**<TAB>**myvideo.avi**<TAB>**8**
The post has been successfully added.
gizem is not your friend, and will not be tagged! The post has been successfully added.
utku is not your friend, and will not be tagged! gizem is not your friend, and will not be tagged! The post has been successfully added.
Error: Your video exceeds maximum allowed duration of 10 minutes.
Error: Please sign in and try again.

xiii. **Remove Post**: The system should allow user to remove a post that has been created the latest. Because of the random unique identifier, you are expected to remove the last created post rather than a particular one.

REMOVELASTPOST

Usage Illustration:

REMOVELASTPOST
Error: You don't have any posts.
Your last post has been successfully removed.
Error: Please sign in and try again.

xiv. **Block User**: In most of the real world social platforms, a user might want to block another user due to several reasons (disturbing posts, etc.). In this assignment, your implementation should also allow users to block other users given a username as the argument.

BLOCK<TAB>**userName**

BLOCK<TAB>**ahmet**
ahmet has been successfully blocked.
No such user!
Error: Please sign in and try again.

xv. **Unblock User**: The system should also allow a user to unblock another user who has been blocked by him/her.

UNBLOCK<TAB>**userName**

UNBLOCK<TAB>**ahmet**
ahmet has been successfully unblocked.
No such user in your blocked users list!
Error: Please sign in and try again.

xvi. **List Friends**: Users should be able to lists their friends, blocked or not.

LISTFRIENDS

LISTFRIENDS
You haven't added any friends yet!
Error: Please sign in and try again.
Name: Adnan Username: adnan Date of Birth: 01/01/1991 School: Selcuk Universitesi --------------------------- Name: Ahmet Username: ahmet Date of Birth: 04/25/2001 School: Meram Anadolu Lisesi ---------------------------

xvii. **List Users**: Users should be able to view all users in the system.

LISTUSERS

LISTUSERS
Error: Please sign in and try again.
Name: Adnan Username: adnan Date of Birth: 01/01/1991 School: Selcuk Universitesi --------------------------- Name: Ahmet Username: ahmet Date of Birth: 04/25/2001 School: Meram Anadolu Lisesi ---------------------------

xviii. **Show Blocked Friends**: Users should be able to lists their friends whom they have blocked.

SHOWBLOCKEDFRIENDS

SHOWBLOCKEDFRIENDS
You haven't blocked any users yet!
You haven't blocked any friends yet!
Error: Please sign in and try again.
Name: Adnan User Name: adnan Date of Birth: 01/01/1991 School: Selcuk Universitesi ---------------------------

xix. **Show Blocked Users**: Similar to the previous execution scenario, users should be able to lists all users whom they have blocked, both friends and non-friends.

SHOWBLOCKEDUSERS

SHOWBLOCKEDUSERS
You haven't blocked any users yet!
Error: Please sign in and try again.
Name: Adnan User Name: adnan Date of Birth: 01/01/1991 School: Selcuk Universitesi --------------------------- Name: Ahmet Username: ahmet Date of Birth: 04/25/2001 School: Meram Anadolu Lisesi ---------------------------

4 INPUT FILES

You will be provided with two (tab-separated) text input files:

- **Users** in the system,
- **Commands** that you need to execute in order to test your application.

4.1 USERS INPUT FILE

List of all users in the system will be given in the (tab-separated) `users.txt` file:

The format of user's data will be given as follows:

name<TAB>**username**<TAB>**password**<TAB>**dateofBirth**<TAB>**graduatedSchool**

A sample `users.txt` input file is given below:

```
Ahmet ahmet ahmet123    04/25/2001  Meram Anadolu Lisesi
Demet demet demet00     01/16/1999  Ankara Fen Lisesi
Zeki  zeki  zeki01      08/16/1987  Kadıkoy Lisesi
Gizem gizem gizem1      12/09/1997  Hacettepe Universitesi
Utku  utku  utku99      10/06/1999  Bilkent Universitesi
Hakan hakan hakan81     03/01/1981  Orta Dogu Teknik Universitesi
```

You are expected to create an instance for each user in this file.

4.2 COMMANDS INPUT FILE

List of all commands, which will be given to test your program and which you are expected to execute correctly and in order, will be given in the (tab-separated) `commands.txt` file. A detailed explanation, format and expected output for each command is given in the 'Scenarios of Execution' section (Section 3.1.4) of this document.

```
ADDUSER       Adnan  adnan  adnan1 01/01/1991    Selcuk Universitesi
REMOVEUSER    1
SIGNIN cemil  cemil1
SIGNIN adnan  adnan1
LISTUSERS
UPDATEPROFILE Adnan  07/01/1991    Gazi University
CHPASS adnan123      adnan1234
CHPASS adnan1 adnan123
ADDFRIEND     ahmet
ADDFRIEND     demet
ADDFRIEND     demet
ADDFRIEND     gizem
ADDFRIEND     utku
ADDFRIEND     ziya
REMOVEFRIEND  zeki
REMOVEFRIEND  ziya
REMOVEFRIEND  utku
LISTFRIENDS
ADDPOST-TEXT  This is my 1st text post   39.2   32.81  ahmet
ADDPOST-IMAGE This is my 1st image post  37.87  32.46  demet:gizem   image.png     135x250
ADDPOST-VIDEO This is my 1st video post  40.87  29.24  utku:gizem    myvideo.avi   8
ADDPOST-TEXT  This is my 2nd text post   38.35  33.1   demet:gizem:utku
REMOVELASTPOST
SHOWPOSTS     adnan
BLOCK  demet
BLOCK  gizem
BLOCK  ahmet
SHOWBLOCKEDFRIENDS
UNBLOCK       ziya
UNBLOCK       gizem
UNBLOCK       gizem
SHOWBLOCKEDFRIENDS
SHOWBLOCKEDUSERS
SIGNOUT
```

5 JAVADOC

Javadoc is the JDK tool that generates API documentation from documentation comments. You are expected to document every method and class using Javadoc. Documentation comments (doc comments) are special comments in the Java source code that are delimited by the /** ... */ delimiters. These comments are processed by the Javadoc tool to generate the API docs.

The Javadoc tool can generate output originating from four different types of "source" files. In this assignment, the source code files we are interested in are your Java classes (.java) - these contain class, interface, field, constructor and method comments.

A format of a doc comment is as follows:

- A doc comment is written in HTML and must precede a class, field, constructor or method declaration. It is made up of two parts -- a description followed by block tags. In this example, the block tags are `@param`, `@return`, and `@see`.

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argument
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
       try {
           return getImage(new URL(url, name));
       } catch (MalformedURLException e) {
           return null;
       }
}
```

- The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the API item. This means the first sentence of each member, class, interface or package description. The Javadoc tool copies this first sentence to the appropriate member, class/interface or package summary. This makes it important to write informative initial sentences that can stand on their own. E.g.

```
/**
    * Class constructor.
    */
   foo() {
     ...

   /**
    * Class constructor specifying number of objects to create.
    */
   foo(int n) {
     ...
```

Tag conventions:

- Order of tags should be:
  - `@author` (classes and interfaces only, required)
  - `@version` (classes and interfaces only, required)
  - `@param` (methods and constructors only)
  - `@return` (methods only)
  - `@exception` (@throws is a synonym added in Javadoc 1.2)
  - `@see`
- Required tags:
  - An `@param` tag is "required" (by convention) for every parameter, even when the description is obvious.
  - The `@return` tag is required for every method that returns something other than `void`, even if it is redundant with the method description. (Whenever possible, find something non-redundant (ideally, more specific) to use for the tag comment.)

For more information on Javadoc and examples of doc comments, visit:
http://www.oracle.com/technetwork/articles/java/index-137868.html

Important note: DO NOT use Turkish characters in your code (anywhere, not even comments)!

6 CONSTRAINTS

- **Update Profile**: `updateProfile` method should perform this operation by invoking relevant mutator (setter) methods within the User class.
- **Add New Friend**: The system should not allow users to add a friend who is already in his/her friend or does not exist in the system.
- **Remove Friend**: The system should not allow user to remove a friend who does not exist or who is not his/her friend. Also, when a friend is removed from a user's friend list, he/she should not be removed from the system!
- **Add Video Post**: the maximum allowed video length is 10 minutes. If a user tries to add a video that lasts longer than that, post should not be added, and an error message should be displayed as illustrated in Section 3.
- **General constraints**:
  - When a user is not logged-in, he/she should be allowed only to perform a sign-in. Only after the user signs in, other actions should be permitted.
  - Users should not be able to tag other users in their posts if they are not their friends.

7 SUBMISSION

- Submissions will be accepted only electronically via submit.cs.hacettepe.edu.tr.
- The deadline for submission is **28.04.2017 until 23:59**.
- The submission format is:

  b<student id>.zip
  - javadoc.zip
  - src.zip
  - uml.jpg
- **Your main class must be named Main.java**! All of your classes should be in the **src** folder (they may, however, be further classified into other subfolders in case you decide to make packages). Note that only zipped folders can be submitted.
- **All work must be individual**! Plagiarism check will be performed! Duplicate submissions will be graded with 0 and disciplinary action will be taken.

8 COMPILE & RUN

javac Main.java

java Main users.txt commands.txt