Fakebook

Object Oriented Programming
2nd Project, version 1.2 – 2020-05-06
Contact: mgoul@fct.unl.pt

Important remarks

Deadline until 23h55 (Lisbon time) of May 31, 2020.

Team this project is to be MADE BY GROUPS OF 2 STUDENTS.

Deliverables: Submission and acceptance of the source code to **Mooshak** (score > 0). See the course web site for further details on how to submit projects to Mooshak.

Recommendations: We value the documentation of your source code, as well as the usage of the best programming style possible and, of course, the correct functioning of the project. Please carefully comment both interfaces and classes. The documentation in classes can, of course, refer to the one in interfaces, where appropriate. Please comment the methods explaining what they mean and defining preconditions for their usage. Students may and should discuss any doubts with the teaching team, as well as discuss this project with other students, but may not share their code with other colleagues. This project is to be conducted with full respect for the Code of Ethics available on the course web site.

1 Development of the application *Fakebook*

1.1 Problem description

The goal of this project is to develop an application that simulates a social network called *fakebook*. Whenever a *user* posts a new *message*, the message becomes available to that user's *contacts*. Users can react to messages sent to them by sending *comments*. Some of these messages are *honest*, while others are *fake*. The comments can be either *positive*, reinforcing the message, or *negative*, denying it. The network offers several analytics.

Fakebook users have a unique id, a collection of contacts, which are bidirectional, and we can also retrieve a collection of posts made by that user, a collection of posts sent to that user and a collection of comments made by that user. There are several kinds of users: *self-centered*, *naïve*, *liar* and *fanatic*. *Self-centered* users can post honest and fake messages. Because they are self-centered, they only write positive comments on their own messages, and never comment on someone else's posts. *Naïve* users can post post honest and fake messages, and can only post positive comments (*i.e.* they always agree with whatever the post they are commenting says, regardless of the post author and honesty). *Liar* users can only post fake messages. Because they are liars, they only post positive comments on fake messages and negative comments on honest messages. *Fanatic* users have a list of topics they are fanatic about and only post and comment messages about subjects they are fanatic about. These topics are defined via hashtags (*e.g.* `♯flatearth`, `♯cutecat`, ...). A fanatic always has a stance concerning the topics (s)he

is fanatic about. If the fanatic has a positive stance on a topic (s)he can only make positive comments on it, if the post is honest, or negative comments, if the post is fake. Conversely, if the fanatic has a negative stance on that topic, (s)he will only make negative comments on honest posts and positive comments on fake posts.

Each post has a single author, an id which is unique for that author, can be either honest or fake and has an associated list of hashtags fixed by the author. Whenever a user writes a new post, Fakebook sends the post to all that author's contacts. Those users can then comment on the post. Each user can make as many comments as they wish on a received post (with the already explained restrictions). The post has a single thread of comments, which may be empty, if the post is not that popular. The author of the post can also comment her own posts.

Posts and comments cannot be deleted. This application is about fake posts. We don't want dishonest users to erase their mischievous past, do we? Or honest ones to erase their past just to look cool to new friends. Once on Fakebook, forever on Fakebook. ;-)

We will also have several convenient queries we can do on Fakebook, to learn who are the most prolific fanatics, which are the most popular posts, and so on. Please check the detailed commands list for further details on this.

Last, but not the least, please do not take any of the following examples of posts and comments seriously. Any similarity between the fictional users in those examples and real people with sort of similar names is just for the sake of making the examples easier to remember. They never said or wrote most of these quotes.

2 Commands

In this section we present all the commands that the system must be able to interpret and execute. In the following examples, we differentiate `text written by the user` from the feedback written by the program in the console. You may assume that the user will make no mistakes when using the program other than those described in this document. In other words, you only need to take care of the error situations described here, in the exact same order as they are described.

Commands are case insensitive. For example, the `exit` command may be written using any combination of upper and lowercase characters, such as `EXIT`, `exit`, `Exit`, `exIT`, and so on. In the examples provided in this document, the symbol ↵ denotes a change of line.

If the user introduces an unknown command, the program must write in the console the message Unknown command. Type help to see available commands. For example, the non existing command `someRandomCommand` would have the following effect:

```
someRandomCommand↵
Unknown command. Type help to see available commands.↵
```

Several commands have arguments. You may assume that the user will only write arguments of the correct type. However, some of those arguments may have an incorrect value. For that reason, we need to test each argument exactly by the order specified in this document. Arguments will be denoted **`with this style`**, in their description, for easier identification.

2.1 `exit` command

Terminates the execution of the program. This command does not require any arguments. The following scenario illustrates its usage.

```
exit↵
Bye!↵
```

This command always succeeds. When executed, it terminates the program execution.

2.2 `help` command

Shows the available commands. This command does not require any arguments. The following scenario illustrates its usage.

```
help↵
register - registers a new user↵
users - lists all users↵
addfriend - adds a new friend↵
friends - lists the user friends↵
post - posts a new message↵
userposts - lists all posts by a user↵
comment - user comments on a post↵
readpost - prints detailed info on a post↵
commentsbyuser - shows all the comments by a user on a given post↵
topicfanatics - shows a list of fanatic users on a topic↵
topicposts - shows a list of posts on a given topic↵
popularpost - shows the most commented post↵
topposter - shows the user with more posts↵
responsive - shows the user with a higher percentage of commented posts↵
shameless - shows the top liars↵
help - shows the available commands↵
exit - terminates the execution of the program↵
```

This command always succeeds. When executed, it shows the available commands.

2.3 `register` command

Registers a new user on fakebook. The command receives as arguments the **`user kind`**, which can be one of **`naive`**, **`liar`**, **`fanatic`**, or **`selfcentered`** and the unique **`user id`**. If the user is a **`fanatic`**, there are some additional arguments: an integer representing the **`number of fanaticisms`** that user has, followed by a **`sequence of fanaticisms`**. The sequence of fanaticisms consists of a sequence of pairs made of either **`loves`** or **`hates`** and an **`hashtag`**.

In the following example, we create one of each user kind: naïve Forrest Gump, liar Pinocchio, fanatic Pete, who loves red, hates blue and green, and self-centered Dim Cardashian.

```
register naive Forrest Gump↵
Forrest Gump registered.↵
register liar Pinocchio↵
Pinocchio registered.↵
register fanatic Pete↵
3 loves #red hates #blue hates #green↵
Pete registered.↵
register selfcentered Dim Cardashian↵
Dim Cardashian registered.↵
```

If some parameter is incorrect, the user is not registered and an adequate error message is presented:

- (1) If the user kind is unknown, the error message is (<user kind> is an invalid user kind!).
- (2) If there is already a user with the same id, the error message is (<user id> already exists!).
- (3) If the user is a fanatic but there are repeated fanaticisms in the user's list, the error message is (Invalid fanaticism list!).

The following example illustrates an interactive session where these error messages would be generated. The problem with the input is highlighted in **red**.

```
register honest Abe↵
honest is an invalid user kind!↵
register naive Forrest Gump↵
Forrest Gump registered.↵
register selfcentered Forrest Gump↵
Forrest Gump already exists!↵
register fanatic Pygmy↵
3 hates ♯cat hates ♯you loves ♯cat↵
Invalid fanaticism list!↵
```

2.4 `users` command

Lists all registered users on fakebook. This command does not require any parameters and always succeeds. It lists all the registered users in alphabetic order. For each user, the list presents his id, kind, total number of friends, posts and comments produced by that user, with the format (<user id> [<user kind>] <friends count> <posts count> <comments count>).

Suppose you have successfully registered several users and by now they have a few friendships, posts and comments produced (e.g. Forrest Gump is naive and has 2 friends, wrote 3 posts and produced 4 comments; Dim Cardashian is selfcentered, has 0 friends but has written 5 posts and 23 comments on her own posts; Pete is a fanatic, has 1 friend and made 0 posts and 4 comments; Pinocchio is a liar, has 1 friend, and has not posted or commented anything yet). The output of the `users` command would be like this:

```
users↵
Forrest Gump [naive] 2 3 4↵
Dim Cardashian [selfcentered] 0 5 23↵
Pete [fanatic] 1 0 4↵
Pinocchio [liar] 1 0 0↵
```

If there are no registered users, the program presents the message (There are no users!).

```
users↵
There are no users!↵
```

2.5 `addfriend` command

Adds a new friend to a user. The command receives 2 parameters: the **`first user id`** and the **`second user id`** and creates a bidirectional relationship between the first and the

second user. In case of success, the program presents the feedback message (<first user id> is friend of <second user id>.).

In the following example, **Pinocchio** becomes a friend of **Dim Cardashian**.

```
addfriend Pinocchio↵
Dim Cardashian↵
Pinocchio is friend of Dim Cardashian.↵
```

If some parameter is incorrect, the program is unable to add a friend to the user:

- (1) If some of the users does not exist, the adequate error message is (<user id> does not exist!). If none of the users exists, the <first user id> is used in the output.
- (2) If the first and second users are the same, the adequate error message is (<first user> cannot be the same as <second user>.)
- (3) If the first user is already a friend of the second user, the adequate error message is (<first user id> must really admire <second user id>!)

The following scenario illustrates these issues:

```
addfriend Some bloke who never registered↵
Some other bloke who did not register either↵
Some bloke who never registered does not exist!↵
addfriend Some bloke who never registered↵
Pinocchio↵
Some bloke who never registered does not exist!↵
addfriend Pinocchio↵
Some bloke who never registered↵
Some bloke who never registered does not exist!↵
addfriend Dim Cardashian↵
Dim Cardashian↵
Dim Cardashian cannot be the same as Dim Cardashian!↵
addfriend Pinocchio↵
Dim Cardashian↵
Pinocchio is friend of Dim Cardashian.↵
addfriend Pinocchio↵
Dim Cardashian↵
Pinocchio must really admire Dim Cardashian!↵
addfriend Dim Cardashian↵
Pinocchio↵
Dim Cardashian must really admire Pinocchio!↵
```

2.6 `friends` command

Lists all the friends of a user. The command receives as parameter the **user id** and lists all that user's friends in a comma separated list. Friends are sorted in alphabetic order.

In the following example, we list the three friends of the **Big Bad Wolf**, who are, of course, the three little pigs (Pip, Pat and Pam).

```
friends Big Bad Wolf↵
Pam, Pat, Pip.↵
```

If the **user id** is unknown, or the user does not have any friends, the following rules apply:

- (1) If the **user id** is unknown the adequate error message is (<user id> does not exist!).
- (2) If the user has no friends the adequate message is (<user id> has no friends!).

The following scenario illustrates these issues:

```
friends Rainbow Unicorn↵
Rainbow Unicorn does not exist!↵
friends Dim Cardashian↵
Dim Cardashian has no friends!↵
```

2.7 `post` command

User posts a new message. The command receives several parameters. The first one contains the **user id**. The second argument is the **number of hashtags** added to the post. This is followed by a sequence of hashtags. The next argument contains the truthfulness of the post to be added. It can be either **honest** or **fake**. Finally the last argument is the **message** itself. The output acknowledges the new post with the message (<user id> sent a <honest | fake> post to <number of friends> friends. Post id = <postId>.). postId is a serial number, starting at 1, for all the posts made by the corresponding user.

The following scenario illustrates a post by **Dim Cardashian** and another one by **Anthony**. Assume **Dim Cardashian** has a couple of friends (**Forrest Gump** and **Pete**) and **Anthony** has 5 (including **Pete**). This was **Dim Cardashian**'s post 3, and **Pete**'s post 7, in the example.

```
post Dim Cardashian↵
4 ♯AwesomeKim ♯red ♯blue ♯whatever↵
honest Well, a bear can juggle while standing on a ball and he is
talented, but he is not famous.  You understand what I am saying?↵
Dim Cardashian sent a honest post to 2 friends. Post id = 3.↵
post Anthony↵
2 ♯MentalSamurai ♯red↵
fake I am pretty sure there are 27 letters in the Portuguese alphabet.
abcdefghijklmnopqrstuvwxyz.  27!  Block the answer!↵
Anthony sent a fake post to 5 friends. Post id = 7.↵
```

If a parameter is incorrect, the post is not sent and the following occurs:

- (1) If the **user id** is unknown the adequate error message is <user id> does not exist!
- (2) If the number of hashtags is not greater or equal to 0 (where 0 would match an empty list of hashtags), or there are repeated hashtags, the adequate error message is (Invalid hashtags list!).
- (3) If the post stance contradicts the user's **stance** (e.g. because the user is a **liar** and the post is **honest**, or because the post is **honest** about something a **fanatic** hates), the adequate error message is (Inadequate stance!).

The following scenario illustrates these error messages. **Rainbow Unicorn** does not exist. An hashtag (in this case, ♯**flatearth**) can only be used once in the topics list of a publication. **Pinocchio** is a liar, so he cannot make an **honest** post. **Pete** hates ♯**blue**, so he cannot make an honest post with that hashtag.

```
post Rainbow Unicorn↵
1 ♯cute↵
honest Be a unicorn in a field full of horses!↵
Rainbow Unicorn does not exist!↵
post Pinocchio↵
3 ♯flatearth ♯soLongUncleNosy ♯flatearth↵
fake Just the other day my uncle was minding his own business and fell
off the edge of the world!↵
Invalid hashtags list!↵
post Pinocchio↵
0↵
honest Gepeto is my dad!↵
Inadequate stance!↵
post Pete↵
1 ♯blue↵
honest The sky is blue and I just love it!↵
Inadequate stance!↵
```

2.8 `userposts` command

Lists all posts by a user. The command receives as parameter the **`user id`** and lists all the posts by that user. The list is presented in order of introduction in the system.

```
userposts Dim Cardashian↵
Dim Cardashian posts:↵
1. [fake] Last few years I have been too cool for duck face. So that is not going to happen. [4
comments]↵
2. [honest] I would, like, died to be in Twilight. [1 comments]↵
3. [honest] Well, a bear can juggle while standing on a ball and he is talented, but he is not famous.
You understand what I am saying? [8 comments]↵
```

If the parameter is incorrect, the post is not sent and the following occurs:

- (1) If the **`user id`** is unknown the adequate error message is <user id> does not exist!
- (2) If the user exists, but has no posts, the adequate error message is <user id> has no posts!

The following scenario illustrates these error messages.

```
userposts Rainbow Unicorn↵
Rainbow Unicorn does not exist!↵
userposts Pete↵
Pete has no posts!↵
```

2.9 `comment` command

comment - user comments on a post. The command receives the **`user id`** of the user who is going to comment, the **`user id`** of the author of the commented post, the **`id`** of the commented post, the **`stance`** of the comment which can be one of **<positive|negative>**, and the **`comment`** itself. Note that it is possible for a user to comment on a post authored by

Table 1: Can a fanatic user make a comment on this post?

Post is Honest	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE
Comment is positive	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
Fanaticism is positive	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE
Can make comment?	**FALSE**	**TRUE**	**TRUE**	**FALSE**	**TRUE**	**FALSE**	**FALSE**	**TRUE**

that user. The rules that decide whether a fanatic can make a comment on any given post are tricky, so decision table 1 to help deciding when (s)he can make a comment.

In the following example, **Forrest Gump** makes a **positive** comment on a post (**3**) made by **Dim Cardashian**. **Anthony** wrote a fake post (his **7**th post), which includes the **♯blue** hashtag. Now **Pete**, who is fanatic for **♯red**, one of the hashtags used by **Dim Cardashian** on her post, feels the urge to support **♯red** and so makes a **positive** comment. And, of course, he makes a positive comment on **Anthony**'s mistaken post on the alphabet because **Pete** hates **♯blue** and the post is **fake**.

```
comment Forrest Gump↵
Dim Cardashian↵
3 positive So true, I know what you mean!↵
Comment added!↵
comment Pete↵
Dim Cardashian↵
3 positive Yeah, you would never see a bear on the RED carpet!  LOL!↵
Comment added!↵
comment Pete↵
Anthony↵
7 positive Let me tell you something...  Anybody with a true heart knows
there are 27 letters in the Portuguese alphabet.  I challenge you to
prove otherwise!↵
Comment added!↵
```

If some parameter is incorrect, the comment is not stored and an adequate error message is returned:

- (1) If the **user id** of either the author of the comment, or the author of the post is unknown, the adequate error message is (<user id> does not exist!). If none of the two users exists, the **user id** used is that of the user making the comment.
- (2) If the post **id** does not exist for that author the adequate error message is (<user id> has no post <post id>!).
- (3) If the user cannot comment that post, because he has not received it from the author (this happens if they were not friends when the post was made), the adequate error message is (<user id> has no access to post <post id> by <user id>!).
- (4) If the user has access to the post, but is not allowed to comment on that post (e.g. because (s)he is self-centered), the adequate error message is (<user id> cannot comment on this post!).
- (5) If the comment **stance** is invalid for that **user id** and that **post** the adequate error message is (Invalid comment stance!).

Let us have a look at a few examples.
(1) Neither **Rainbow Unicorn** nor **Santa Claus** exist.

```
comment Rainbow Unicorn↵
Dim Cardashian↵
3 positive Nobody shines as much as you, Dim!↵
Rainbow Unicorn does not exist!↵
comment Rainbow Unicorn↵
Santa Claus↵
2 positive I am so glad to hear you are coming to town!↵
Rainbow Unicorn does not exist!↵
comment Pete↵
Santa Claus↵
1 positive I love your RED uniform.↵
Santa Claus does not exist!↵
```

(2) **Pinocchio** fails to comment on a post by **Dim Cardashian** because she never wrote it in the first place.

```
comment Pinocchio↵
Dim Cardashian↵
42 negative I am just like you, Dim.  I would rather be really smart
than to be an actor!↵
Dim Cardashian has no post 42!↵
```

(3) **Pinocchio** fails to comment on a post by **Big Bad Wolf** because they are not friends on fakebook.

```
comment Pinocchio↵
Big Bad Wolf↵
2 positive Little pigs are vegetables, Wolfie!↵
Pinocchio has no access to post 2 by Big Bad Wolf!↵
```

(4) **Dim Cardashian** fails to comment a post which is not her own because she is **selfcentered**.

```
comment Dim Cardashian↵
Pinocchio↵
2 positive White is, in fact, one of my favorite colors.  I have a↵
white car.  I love white.↵
Dim Cardashian cannot comment on this post!↵
```

(5) **Pete** fails to make a **positive** comment on an **honest** post with the **♯blue** hashtag because he has a **fanatic** hate for **♯blue**.

```
comment Pete↵
Forest Gump↵
1 positive Lovely blue sky, is it not?↵
Invalid comment stance!↵
```

(5) **Pinocchio** fails to make a **negative** comment on a **fake** post by **Anthony** because **Pinocchio** is a **liar**.

```
comment Pinocchio↵
Anthony↵
7 negative I am pretty sure they are 26, Anthony!↵
Invalid comment stance!↵
```

(5) This last one is a bit trickier. **Pete** loves **red**, but hates **♯blue**. So he tries to make a **negative** comment, including an insult, for good measure, to **Dim Cardashian**'s **4**th post because the post is **♯blue**. However, that post is also **red**. What to do? Because Pete is fanatic for **♯red** before being fanatic for **♯blue**, **♯red** is higher in his priorities. So, the rule is that he program will analyse each of the user's hashtags **in order of insertion** and check them against the hashtags in the post. The user will be positive, or negative, depending on the first match to be found. In this case, he will be positive about the post because his **♯red** love comes before his **♯blue** hate and therefore is not allowed to make a **negative** post.

```
comment Pete↵
Dim Cardashian↵
4 negative What about Winnie the Poo, you mindless socialite?  And why
are you wearing that hideous BLUE dress?↵
Invalid comment stance!↵
```

2.10 `readpost` command

Prints detailed information on a post. The command receives receives two parameters: the **user id** of the post author and the **post id**. It then prints the post contents followed by all the comments, sorted by insertion in the system. The following example shows comments on one of **Anthony's 7th post**. Yep, the one with the alphabet ended up generating quite a heated debate. Have a look.

```
readpost Anthony↵
7↵
[Anthony fake] I am pretty sure there are 27 letters in the Portuguese alphabet.
abcdefghijklmnopqrstuvwxyz. 27! Block the answer!↵
[Forrest Gump positive] Never been good with letters and stuff! You are probably right.↵
[6-year old kid negative] No way. 26!↵
[Pete positive] Let me tell you something... Anybody with a true heart knows there are 27 letters in
the Portuguese alphabet. I challenge you to prove otherwise!↵
[6-year old kid negative] Isa from TV School told me there are 26 and that I should wash my hands!↵
[Pete positive] Grow up, kid! I dare you to prove they are 26! Gosh! The nerve!↵
[6-year old kid negative] But... look: abcde... fghij... oopsie daisy... toes. Ok. klmno... pqrst...
Moooom, I ran out of fingers, can I borrow your hands? Thanks. Where was I? Oh, yes, 20. Silly me.
uvwxy...z. Look, its 26!↵
[Pete positive] Put some socks on and go wash your hands, little brat!↵
[Anthony positive] There must be some mistake, did I just lose 1250 euros?↵
[Pete positive] I demand a recount!↵
```

If some parameter is incorrect, the list is not produced, and an adequate error message is presented:

- (1) If the **user id** is unknown, the adequate error message is (<user id> does not exist!).
- (2) If the user has no message with the given **id** the adequate error message is (<user id> has no post <id>!).

- (3) If there are no comments yet, the post is printed, followed by the error message (No comments!).

In the following example, we see each of these situations. **Santa Claus** does not exist, Anthony has no **13**th post, and nobody commented on Dim Cardashian's **4**th post, yet.

```
readpost Santa Claus↵
1↵
Santa Claus does not exist!↵
readpost Anthony↵
13↵
Anthony has no post 13!↵
readpost Dim Cardashian↵
4↵
[Dim Cardashian honest] I feel blessed because I honestly enjoy the whole process of doing makeup
and hair.↵
No comments yet!↵
```

2.11 `commentsbyuser` command

Shows all the comments by a user on a particular topic. The comment receives two parameters, the **user id** and the **topic id**. It then presents all the comments of that user about that topic. Please note that there can be potentially thousands of different topics addressed by a user, as these topics are the union of all the hashtags of all the posts commented by that user. Comments are **presented by order of introduction**, within that specific topic. As we already saw, the same comment can be associated to more than one topic. In the following example, assume Pete is participating in more than one online discussion at the same time, so messages get interleaved. He first participates in a discussion started by Anthony, then by Dim Cardashian, then back to Anthony's discussion, and so on.

```
commentsbyuser Pete↵
#red↵
[Anthony fake 7 positive] Let me tell you something... Anybody with a true heart knows there are 27
letters in the Portuguese alphabet. I challenge you to prove otherwise!↵
[Dim Cardashian honest 3 positive] Yeah, you would never see a bear on the RED carpet! LOL!↵
[Anthony fake 7 positive] Grow up, kid! I dare you to prove they are 26! Gosh! The nerve!↵
[Anthony fake 7 positive] Put some socks on and go wash your hands, little brat!↵
[Anthony fake 7 positive] I demand a recount!↵
```

If some parameter is incorrect, the list is not produced, and an adequate error message is presented:

- (1) If the **user id** is unknown, the adequate error message is (<user id> does not exist!).
- (2) If the user has not made any comments (No comments!).

In the following example, **Santa Claus** does not exist, and the **Big Bad Wolf** has no comments.

```
commentsbyuser Santa Claus↵
♯cocacola↵
Santa Claus does not exist!↵
commentsbyuser Big Bad Wolf↵
♯constructionStandards↵
No comments!↵
```

2.12 `topicfanatics` command

Shows a list of fanatic users on a given topic. The list is presented in alphabetic order of fanatic **`user id`**. In the following example, **♯red** fanatics are shown.

```
topicfanatics ♯red↵
Anthony, Pete.↵
```

If the parameter is incorrect, an adequate error message is presented.

- (1) If the kind of fanaticism is unknown, the adequate error message is (Oh please, who would be a fanatic of <fanaticism id>?).

In the following example the topic **♯SittingInAWaitingRoom** is so booooring that none of the users of fakebook has enlisted as a fanatic of it. Not even as a fanatic against it. It is just too boring to care about.↵

```
topicfanatics ♯SittingInAWaitingRoom↵
Oh please, who would be a fanatic of ♯SittingInAWaitingRoom?↵
```

2.13 `topicposts` command

Shows a list of posts on a given topic. The command receives two arguments: the **`topic`** and the **`maximum number of posts to list`**. The list is presented in reverse number of comments per post and, when ties occur, by alphabetic order of author id, followed by decreasing id for the post (so, more recent posts are presented first). Each line has the format (<user id> <post id> <number of comments on post>: <message>). The printed list will have up to the `maximum number of posts to list` elements. It may have less elements, if not enough posts were produced (e.g. if we ask for a list of the top 7 posts on ♯cutecats and there are only 4 posts on that topic, only 4 posts will be shown; on the other hand, if there are 34 posts on that topic, only the top 10 will be listed).

```
topicposts ♯beautytips 2↵
Dim Cardashian 5 9: I will cry at the end of the day. Not with fresh makeup.↵
Pete 2 6: I always look smart with a red tie!↵
```

If a parameter is incorrect, an adequate error message is presented.

- (1) If the maximum number of posts to list is not greater or equal to 1, the adequate error message is (Invalid number of posts to present!).
- (2) If the **<topic id>** is unknown, the adequate error message is (Oh please, who would write about <topic id>?).

In the following example the topic **♯SittingInAWaitingRoom** is so booooring that nobody wrote about it.↵

```
topicposts ♯SittingInAWaitingRoom 3↵
Oh please, who would write about ♯SittingInAWaitingRoom?↵
```

2.14 `popularpost` command

Shows the most commented post, including post author id, post id, and number of comments. When ties occur with the number of votes, the alphabetic order of author id is used, giving priority to lower order names. If there is still a tie, the next and final criterion is the id for the post, so that more recent posts are considered more popular than older posts. In the following example, assume Dim Cardashian's post 5 is the most popular, with 9 comments. Actually, it was tied with her own post 2, but post 5 is more recent, so 5 is the most popular post. The output format is (<user id> <post id> <number of comments on post>: <message>).

```
popularpost↵
Dim Cardashian 5 9: I will cry at the end of the day. Not with fresh makeup.↵
```

Even without parameters, things may go wrong:

- (1) If there are no posts on fakebook, the adequate error message is (Social distancing has reached fakebook. Please post something.).

```
popularpost↵
Social distancing has reached fakebook. Please post something.↵
```

2.15 `topposter` command

Shows the id of the top poster, the count of the posts made by this user and the count of comments. In case of ties, the total number of comments by the user will be used to break the tie, favouring the user who wrote more comments. If there is still a draw, growing alphabetic order should be used. The output line has the format (<user id> <posts> <comments>.).

In the following example, assume Anthony is the user with more posts.

```
topposter↵
Anthony 7 4.↵
```

Even without parameters, things may go wrong:

- (1) If there are no posts on fakebook, the adequate error message is (Social distancing has reached fakebook. Post something to become the king of posters.).

```
topposter↵
Social distancing has reached fakebook. Post something to become the king of posters.↵
```

2.16 `responsive` command

Shows the id of the user with the higher percentage of commented posts. This is the percentage of posts the user has access to in which the user wrote at least one comment (including posts authored by that user). If there is still a draw, growing alphabetic order of user id should be used. After the user id, the command prints the number of commented posts, followed by the number of available to that user. Note that the posts available to a user are the union of posts published by that user with the posts published by that user's friends.

In the following example, assume Pete is the most responsive user, with comments on 8 out of the 9 posts available to him. Of these 9 posts, 3 are his own posts, while the other 6 were written by friends of Pete. The output line has the format (<user id> <comments> <posts>).

```
responsive↵
Pete 8 9.↵
```

Even without parameters, things may go wrong:

- (1) If there are no posts on fakebook, the adequate error message is (Social distancing has reached fakebook. Post something and then comment your own post to become the king of responsiveness.).

```
responsive↵
Social distancing has reached fakebook. Post something and then comment your own post to
become the king of responsiveness.↵
```

2.17 `shameless` command

Shows the id of the top liars as well as statistics on how many lies each of them made in the network. Lies include both fake posts and comments which resulted in a lie, either by reinforcing a fake news or by giving a negative comment to an honest post.

In the following example, assume Pete is the most liar user, with a total of 15 lies between his posts and his comments. Note that a user may post several comments on the same post, so there are more lies than post threads in which Pete participated (in case you were wondering how he could lie 15 times about only 8 posts). The line has the format (<user id> <lies>).

```
shameless↵
Pete 15.↵
```

Even without parameters, things may go wrong:

- (1) If there are no posts on fakebook, the adequate error message is (Social distancing has reached fakebook. Post a lie and become the king of liars.).

```
shameless↵
Social distancing has reached fakebook. Post a lie and become the king of liars.↵
```

3 Developing this project

Your program should take the best advantage of the elements taught in the Object-Oriented programming course. You should make this application as **extensible as possible** to make it easier to add, for instance, new kinds of users.

You can start by developing the main user interface of your program, clearly identifying which commands your application should support, their inputs and outputs, and error conditions. Then, you need to identify the entities required for implementing this system. Carefully specify the **interfaces** and **classes** that you will need. You should document their conception and development using a class diagram, as well as documenting your code adequately, with Javadoc.

It is a good idea to build a skeleton of your `Main` class, to handle data input and output, supporting the interaction with your program. In an early stage, your program will not really do much. Remember the **stable version rule**: do not try to do everything at the same time. Build your program incrementally, and test the small increments as you build the new functionalities in your new system. If necessary, create small testing programs to test your classes and interfaces.

Have a careful look at the test files, when they become available. You should start with a really bare bones system with the `help` and `exit` commands, which are good enough for checking whether your commands interpreter is working well, to begin with. Then, add users, and the users listing operation, so that you can create users and then check they are ok. Then add friends and their listing capabilities. And so on. Step by step, you will incrementally add functionalities to your program and test them. **Do not try to make all functionalities at the same time. It is a really bad idea.**

In this project you will handle several collections. These offer you several opportunities for reusing code, rather than repeating it over and over again. The same holds for iterators. Use the Java Collections, implemented with generics, as this will significantly speed up your project and increase the code reliability, as these collections are well-tested.

Last, but not the least, **do not underestimate the effort for this project**.

4 Submission to Mooshak

To submit your project to mooshak, please register in the mooshak contest POO1920-TP2 and follow the instructions that will be made available on the moodle course website.

4.1 Command syntax

For each command, the program will only produce one output. The error conditions of each command have to be checked in the exact same order as described in this document. If one of those conditions occurs, you do not need to check for the other ones, as you only present the feedback message corresponding to the first failing condition. For example, if you attempt to create a post with an invalid user id, the remaining conditions do not need to be checked. However, the program does need to consume all the remaining input parameters, even if they are to be discarded.

4.2 Tests

The Mooshak tests verify incrementally the implementation of the commands. They will be be made publicly available on May 11 2020. When the sample test files become available, use them to test what you already have implemented, fix it if necessary, and start submitting your

partial project to mooshak. Do it from the start, even you just implemented the exit and help commands. By then you will probably have more than those to test, anyway. Good luck!