Progetto Intermedio – SocialNetwork

È richiesto di progettare un SocialNetwork (MicroBlog) dove l'utente può pubblicare e lasciare like ai vari Post presenti nella rete sociale. Il concetto di "like" è legato a quello del "follow", comunemente usato sui social. Di fatto, mettendo like ad un Post, in automatico, "seguiamo" anche l'autore del suddetto Post. Come richiesto dalle specifiche, la classe Post deve essere immutable e per mettere like è sufficiente pubblicare un Post scrivendo "like:" con l ID del Post in esame (es. "like:1234" → l'autore di questo Post "seguirà" l'autore del Post il cui ID è 1234).

Classe Post

```
private final int id;
private final String author;
private final String text;
private final Timestamp timestamp;
```

Le variabili usate sono quelle richieste dalla specifica:

- **ID** -> ID univoco del Post
- **author** -> autore del Post
- **text** -> testo del Post (max 140 caratteri, causa SocialPostArgumentException)
- **timestamp** -> data e ora della pubblicazione del Post

Essendo la classe immutable, gli unici metodi presenti sono dei semplici getter e il costruttore, strutturato nel seguente modo:

```
public Post(int id, String author, String text) throws SocialPostArgumentException {
    this.id = id;
    this.author = author;
    this.text = text;
    if(text.length() > 140)
        throw new SocialPostArgumentException("Testo troppo lungo");
    this.timestamp = new Timestamp(System.currentTimeMillis());
}
```

Classe SocialNetwork

La classe SocialNetwork, com'è facile intuire, è il fulcro del progetto. Per la realizzazione, ho adoperato le seguenti variabili di istanza:

```
private final Map<String, Set<String>> social;      //utenti registrati + seguiti
private final HashMap<String, Integer> influencers; //utenti registrati + numero di followers
private final HashMap<Integer, Post> posts;          //post approvati
private final Set<String> metionedUsers;            //utenti che hanno postato almeno una volta
```

- **social** -> richiesta dalla specifica, serve per tenere traccia degli utenti registrati e dei relativi seguiti
- **influencers** -> non indispensabile, ma necessaria per evitare doppi cicli che incrementerebbero la complessità del programma. Serve per tenere traccia di quanti followers ha uno specificato utente
- **Posts** -> elenco di tutti i Post approvati (quindi tutti quelli che hanno un ID univoco e che hanno un testo con meno di 140 caratteri)

- **metionedUsers** -> non indispensabile, ma necessaria per evitare cicli inutili. Serve per tenere traccia degli utenti che hanno Postato almeno una volta.

Metodo: GuessFollowers(List<Post> ps) → Map<String,Set<String>>

Questo metodo analizza i Post passati come parametro e, in base a quelli, ne costruisce la relativa rete sociale.

In pratica se un utente U1 pubblica un Post e un altro utente U2 gli mette like, nella Map di ritorno deve risultare che l'utente U2 "segue" l'utente U1. Quindi, l'output deve essere del tipo {U1 = [U2]}

Attenzione: l'ordine non è importante! Anche se io nella lista ho messo come primo valore il like di U2 e come secondo valore il Post di U1, di fatto, il like di U2 è stato messo DOPO il Post di U1, quindi la guessFollowers deve tener conto di questo.

L'algoritmo usato è molto semplice:

a) separo i like dai Post normali
b) per ogni like, prendo il corrispondente Post
c) se un Post p1 è stato trovato, allora controllo che l'autore del like e del Post p1 non corrispondano e, tramite la timestamp fornita dalla classe Post, mi assicuro che il like sia stato messo dopo la pubblicazione del Post
d) se tutto va bene, allora prendo i followers dell'autore del Post p1 e ci aggiungo l'autore del like

a)

```
for (Post post : ps){   //separo i likes dai Post
    if(posts.containsKey(post.getId()) || likes.containsKey(post.getId()))
        throw new SocialDuplicatePostException();
    reteSociale.put(post.getAuthor(), new HashSet<>()); //registro l utente nella rete sociale fittizia

    if(post.getText().contains("like:")){
        likes.put(post.getId(),post);
    }else {
        posts.put(post.getId(),post);
    }
}
```

b)

```
for (Post like : likes.values()) {
    try {
        String[] splittedText = like.getText().split( regex: ":");
        int idPost = Integer.parseInt(splittedText[1].trim());
        Post likedPost = posts.get(idPost);   //prendo il Post a cui like.author ha messo like
```

c)

```
if (likedPost != null && like.getTimestamp().after(likedPost.getTimestamp())) {//se il like è stato messo dopo il Post
    if (likedPost.getAuthor().equals(like.getAuthor()))
        throw new SocialFollowBackException("Non ci si può seguire da soli su questo Social!");
```

d)

```
//metto like.author tra i followers dell autore del Post a cui lui ha messo like
if (!reteSociale.containsKey(likedPost.getAuthor())) {
    reteSociale.put(likedPost.getAuthor(), new HashSet<>(Collections.singleton(like.getAuthor())));
} else {
    reteSociale.get(likedPost.getAuthor()).add(like.getAuthor());
}
```

Metodo: influencers () → List<String>

L'obbiettivo di questo metodo è restituire gli utenti più "influenti" nella rete sociale (ovvero quelli che hanno più followers). Avendo già una Map contenente, per ogni utente, il corrispettivo numero di seguaci, basta fare un ordinamento per ordine decrescente in base al numero di followers.

N.B. la Map influencers non è essenziale, ma togliendola avrei dovuto crearla internamente alla funzione, ciclare sulla Map social e calcolare, per ogni utente, il corrispettivo numero di followers. Essendo, tra l'altro, una map contenente {utenti → seguiti} avrei dovuto ciclare ulteriormente dentro i seguiti e, per ogni utente, aggiungere la key della map come follower dell'utente. La complessità sarebbe aumentata ad un O (n^2).

Creandola globalmente, ho la possibilità di aggiornare il numero di followers di un utente ogni volta che gli viene messo un like. Nell'esempio qui sotto, "post.getAuthor()" è l'autore del Post a cui è stato messo like e author è l'utente che ha messo il like al suo Post.

```
social.get(author).add(post.getAuthor());
influencers.put(post.getAuthor(), influencers.get(post.getAuthor())+1); //aggiorno il numero di followers
                                                                         //di post.getAuthor()
```

L'algoritmo è piuttosto semplice:

Sfrutto l interfaccia Map.Entry<Username,Followers> e il metodo entrySet() per iterare sugli elementi. In pratica Map.Entry è un'interfaccia che mi permette di avere Key e Value in un'unica classe; questo mi evita di dover ciclare su tutte le Key di influencers, per poi prenderne i rispettivi valori.

Facendo un "comparingByValue", ordino gli utenti in ordine crescente in base al numero di followers. A questo punto mi basta solo invertire la lista con un ciclo e prendere solo i nomi degli utenti.

```
public List<String> influencers() {
    List<Map.Entry<String, Integer>> sorted_influencers = new ArrayList<>(influencers.entrySet());
    sorted_influencers.sort(Map.Entry.comparingByValue());  //ordino gli utenti per ordine crescente
    // in base al numero di follower che hanno
    List<String> result = new ArrayList<>();

    for (int i = sorted_influencers.size() - 1; i >= 0; i--) {
        result.add(sorted_influencers.get(i).getKey());
    }

    return result;
}
```

Metodi:

getMentionedUsers () → Set<String> / getMentionedUsers (List<Post> ps) → Set<String>

Questo metodo deve restituire tutti gli utenti che hanno Postato almeno una volta sulla rete sociale.

N.B. il Set metionedUsers non è essenziale, ma togliendolo avrei dovuto crearlo internamente alla funzione, ciclare sulla Map Posts e prendere, per ogni Post, il corrispondente autore. Creandolo globalmente, invece, ogni volta che un utente pubblica un Post, l'autore viene inserito all'interno del Set (e siccome, per definizione, un Set non può contenere duplicati, non importa nemmeno che controlli se l'utente è già stato inserito o no). Infine, mi basta solamente un getter (così il costo rimane costante).

```
metionedUsers.add(author);
```
(→ sta dentro il metodo post (String author, String text))

Il secondo metodo si limita a prendere, per ogni Post nella lista ps, il corrispondente autore.

```
public Set<String> getMentionedUsers() {
    return metionedUsers;
}

public Set<String> getMentionedUsers(List<Post> ps) {
    Set<String> list = new HashSet<>();
    for (Post post : ps) {                                    //per ogni post in ps
        list.add(post.getAuthor());
    }

    return list;
}
```

Metodi:

- writtenBy () → List<Post>
- writtenBy (List<Post> ps, String username) → List<Post>
- containing (List<String> words) → List<Post>

Questi metodi sono molto simili tra loro, di fatto la logica è analoga:

- ciclare su una struttura dati alla ricerca di uno o più elementi

Nel caso della writtenBy devo cercare tutti i Post (dati o come parametro o in this) scritti da un utente passato come parametro.

Nel caso della containing devo cercare tutti i Post, presenti nel social, che contengono almeno una delle parole presenti nella lista, data come parametro del metodo.

```
public List<Post> writtenBy(String username) throws SocialUserArgumentException {
    if (username.isEmpty() || username.trim().length() == 0)
        throw new SocialUserArgumentException("username non può essere vuota o contenere solo spazi vuoti");

    List<Post> list = new ArrayList<>();
    for (Post post : posts.values()) {                      //per ogni post in posts<Integer,Post>
        if (post.getAuthor().equals(username)) {
            list.add(post);                                 //lista dei post scritti da username
        }
    }
    return list;
}

public List<Post> writtenBy(List<Post> ps, String username) throws SocialUserArgumentException {
    if (username.isEmpty() || username.trim().length() == 0)
        throw new SocialUserArgumentException("username non può essere vuota o contenere solo spazi vuoti");

    List<Post> list = new ArrayList<>();
    for (Post post : ps) {
        if (post.getAuthor().equals(username)) {
            list.add(post);
        }
    }
    return list;
}

public List<Post> containing(List<String> words) {
    List<Post> result = new ArrayList<>();

    for (Post post : posts.values()) {
        for (String word : words) {
            if (post.getText().contains(word)) {
                result.add(post);
                break;
            }
        }
    }

    return result;
}
```

Metodi aggiuntivi (ma essenziali):

Post (String author, String text) → Post

Questo metodo permette la pubblicazione di un Post all'interno della rete sociale. È infatti possibile pubblicare un Post SOLO attraverso questo metodo. Questo mi permette di evitare controlli ausiliari, quali per esempio Post duplicati o like messi a Post che ancora non esistono (utilizzando magari anche la timestamp).

Pubblicare un Post, significa semplicemente associare un ID (generato incrementando un intero di 1) al Post vero e proprio se e solo se author e text non sono vuoti (causa SocialPostArgumentException). Successivamente aggiungerlo alla Map Posts, creata globalmente.

```
Post newPost = new Post(id_posts, author, text);
posts.put(id_posts, newPost);                                    //pubblico il post
metionedUsers.add(author);
id_posts++;
return newPost;
```

- Se il Post contiene la dicitura "like:ID", allora viene preso il Post con quell'ID specificato

```
if (text.startsWith("like:")) {     //se il post comincia con "like:", author viene inserito tra i followers
                                    //dell autore del Post a cui ha messo like (post.getAuthor())
    //(quindi usando la HashMap social)
    String[] splittedText = text.split( regex: ":");
    try {
        int idPost = Integer.parseInt(splittedText[1].trim());
        Post post = posts.get(idPost);        //ritorna null se get non trova nulla
```

- e, se esiste, viene aggiunto l'autore del Post tra i seguiti dell'autore del like

```
if (post != null) {                 //se io pubblico like:1234 ma 1234 non corrisponde a nessun post
                                    //viene comunque pubblicato, ma viene scartato dall analisi dei like
    if (post.getAuthor().equals(author))
        throw new SocialFollowBackException("Non ci si può seguire da soli su questo Social!");

    social.get(author).add(post.getAuthor());
    influencers.put(post.getAuthor(), influencers.get(post.getAuthor())+1); //aggiorno il numero di followers
                                                                            //di post.getAuthor()
}
```

(es. Se utente1 pubblica un Post con scritto "hello world" e l utente2 gli mette like, allora l utente2 avrà tra i suoi seguiti l utente1)

Le eccezioni SocialUserException, SocialPostArgumentException, SocialFollowBackException sono lanciate rispettivamente se:

- l'utente non è registrato nel social (quindi se la Map social non contiene la key author)
- se author e text sono vuoti
- se l'autore del like e del Post a cui si vuole mettere like corrispondono

PROBLEMA: se nella batteria di test viene creato un Post e un like relativo a quel Post, nel seguente modo

```
Post post = new Post( id: 1, author: "autore1",  text: "hello world");   //id del Post = 1
Post like = new Post( id: 2,  author: "utente2",  text: "like:1");       //like al Post con id = 1
```

Può verificarsi il seguente caso:

```
post creato: 2020-11-24 11:50:55.126
like messo:  2020-11-24 11:50:55.126
```
(ovvero stesso timestamp)

Questo perché il tempo che intercorre tra la creazione dei due Post è minore di 1 ms, cioè la minima differenza di tempo rilevata da System.currentTimeMillis(), usata dalla Timestamp.

Questo può creare problemi nella guessFollowers, in quanto viene confrontato il timestamp dei Post per vedere se il like è stato messo prima o dopo la pubblicazione del Post. Per ovviare a questo problema, basta "rallentare" di pochi ms la creazione di un Post.

```
try {
    Thread.sleep( millis: 10);    //per non avere timestamp uguali alla creazione di un Post
}catch(InterruptedException ignored){}
```

createUser (String author) → String

Questo metodo serve per registrare un utente sul social network.

```
public String createUser(String username) throws SocialDuplicateUserException, SocialUserArgumentException {
    if (username.isEmpty())
        throw new SocialUserArgumentException("Il campo username, non può essere vuoto");

    if (social.containsKey(username)) {
        throw new SocialDuplicateUserException();
    }

    social.put(username, new HashSet<>());
    influencers.put(username, 0);   //all inizio, l utente appena registrato non segue nessuno
    return username;
}
```

Le eccezioni **SocialDuplicateUserException**, **SocialPostArgumentException**, sono lanciate rispettivamente se:

- Esiste già un utente sul social con quello username
- Se username è passata come stringa vuota

Custom Exceptions:

- **SocialDuplicatePostException** → se esistono nel Social Network, due Post con il solito ID
- **SocialDuplicateUserException** → se esistono nel Social Network, due Utenti con il solito username
- **SocialFollowBackException** →Se un utente sta cercando di mettersi like da solo
- **SocialPostArgumentException** → se un Post contiene parametri sbagliati (testo troppo lungo, author vuoto...)
- **SocialPostException** → errori generici che non sono legati ai parametri
- **SocialUserArgumentException** → se un utente contiene parametri sbagliati (username vuoto...)
- **SocialUserException** → errori generici che non sono legati ai parametri

---

Parte 3

Nella parte tre del progetto, viene richiesto un modo per poter segnalare eventuali Post offensivi. È richiesto inoltre di dover usare la classe SocialNetwork come superclasse (o classe padre). La scelta utilizzata da me comporta l'implementazione di un metodo protetto nella classe SocialNetwork (quindi solo chi estende la classe, può averne accesso).

```java
protected HashMap<Integer, Post> getPosts() {
    return posts;
}
```
(→ sta dentro la classe SocialNetwork)

La classe FilteredSocialNetwork sovrascrive il metodo "post", in quanto la segnalazione avviene solo tramite di esso. Se il testo del Post da pubblicare comincia con "rep:ID", allora viene preso il Post con quell'ID specificato dalla classe padre (quindi dal SocialNetwork) e viene messo in una blacklist.

```java
@Override
public Post post(String author, String text) throws SocialPostArgumentException, SocialUserException, SocialFollowBackException {
    Post post=super.post(author, text);
    if(text.startsWith("rep:")){    //controllo se il Post da segnalare esiste,
                                    // e solo in quel caso lo metto nella blacklist
        String[] splitted_text=text.split( regex: ":");
        try {
            int reportedPostID = Integer.parseInt(splitted_text[1].trim());
            Post repPost= getPosts().get(reportedPostID);
            if(repPost != null)
                blacklist.add(repPost);
        }catch(NumberFormatException ignored){}
    }
    return post;
}
```

Nel caso in cui si decidesse di non rimuovere il Post (perché magari l'amministratore ha deciso così), è possibile rimuovere il Post dalla blacklist mediante il seguente metodo:

```java
public void removePostfromBlacklist(Post p){
    blacklist.remove(p);
}
```

Variabili globali:

- **blacklist** → lista contenente tutte le reference ai Post della rete principale che sono stati segnalati

Altri modi per poter segnalare un Post, potrebbero essere:

- utilizzare una lista contenente parole offensive e, se un Post contiene una di quelle parole, mettere quel Post dentro la blacklist.
  ➔ Questo è il metodo più facile e barbaro, di contro però la lista sarebbe davvero troppo lunga e, in termini di memoria e time-complexity, il costo sarebbe davvero troppo alto.
- Implementare dentro la classe SocialNetwork un metodo protetto che, dato l ID del Post da segnalare, lo si metta dentro una lista privata. Così facendo, tutte le classi che estendono SocialNetwork hanno accesso a quella lista e ne possono controllare il contenuto (e quindi eliminare il Post o lasciarlo nella rete)
  ➔ Così facendo però, si perde il senso dell'ereditarietà e non avrebbe senso estendere una classe solo per accedere ad un metodo specifico della classe padre.

La scelta implementata da me permette di avere due tipi di SocialNetwork (uno che permette il filtraggio dei Post e uno no) il che è coerente con il concetto di ereditarietà: di fatto la classe FilteredSocialNetwork è un tipo di Social dove i Post possono essere filtrati.

Main

```
    public static void main(String[] args) throws Exception {
        Post p = new Post( id: 10,  author: "utente1",  text: "like:1");
        String crash = socialNetwork.createUser( username: "crash");
        String coco = socialNetwork.createUser( username: "coco");
        String cortex = socialNetwork.createUser( username: "cortex");
        String brio = socialNetwork.createUser( username: "brio");
        String gabry1 = socialNetwork.createUser( username: "gabry");
        //String gabry2 = "gabry";    //questa è la solita persona creata prima

        Post cortexPost = socialNetwork.post(cortex,  text: "Ti prenderò Crash!");
        Post crashPost = socialNetwork.post(crash,  text: "Ok!");

        Post like1 = socialNetwork.post(crash,  text: "like:" + cortexPost.getId());
        Post like3 = socialNetwork.post(cortex,  text: "like:abcd");//viene preso come Post normale
        Post like4 = socialNetwork.post(brio,  text: "like:" + crashPost.getId());
        Post like5 = socialNetwork.post(coco,  text: "like:" + crashPost.getId());

        System.out.print("Social: ");
        socialNetwork.printSocial();  //stampa tutti gli utenti con i relativi followers
        System.out.println("(influencers) Lista delle persone più influenti: "+socialNetwork.influencers());
                                                                                    //deve stampare
                                                                                    //1.crash
                                                                                    //2.cortex
                                                                                    //brio
                                                                                    //coco
        System.out.println("(metionedUsers) Hanno postato almeno una volta: "+socialNetwork.getMentionedUsers());
        System.out.println("(writtenBy) crash ha scritto: "+socialNetwork.writtenBy(crash));
        ArrayList<String> words = new ArrayList<>();
        words.add("like");
        System.out.println("(containing) i Post con \"like:\" "+socialNetwork.containing(words));

//======================================================================================================================
        List<Post> list = new ArrayList<>();
        Post like_guess = socialNetwork.post(gabry1,  text: "like:" + cortexPost.getId());
        Post followBackException = new Post( id: 100, cortex,  text: "like: "+cortexPost.getId());

        list.add(cortexPost);
        list.add(crashPost);
        list.add(like1);        //crash mette like a cortex
        list.add(like3);        //cortex scrive like:abcd
        list.add(like4);        //brio mette like a crash
        list.add(p);            //viene scartato, in quanto il like è stato messo prima del Post
        list.add(like_guess);   //gabry mette like a cortex
        Collections.shuffle(list);  //mescolo tutto

//        list.add(followBackException);

        System.out.println("(guessFollowers) rete sociale derivata: "+socialNetwork.guessFollowers(list));
        System.out.println("[metionedUsers(ps)] utenti mezionati in list: "+ socialNetwork.getMentionedUsers(list));
//======================================================================================================================
        FilteredSocialNetwork filteredSocialNetwork = new FilteredSocialNetwork();
        String rillaRoo=filteredSocialNetwork.createUser( username: "rilla_roo");
        String tiny=filteredSocialNetwork.createUser( username: "tiny");

        Post tinyOffensivePost=filteredSocialNetwork.post(tiny,  text: "aaarrghh!");
        filteredSocialNetwork.post(rillaRoo,  text: "rep:"+tinyOffensivePost.getId());

        System.out.println("(filteredSocialNetwork) Post offensivi: "+filteredSocialNetwork.getBlacklist());
        filteredSocialNetwork.removePostfromBlacklist(tinyOffensivePost);
    }
```

Output:

```
Social: {cortex=[], coco=[crash], gabry=[], brio=[crash], crash=[cortex]}
(influencers) Lista delle persone più influenti: [crash, cortex, brio, gabry, coco]
(metionedUsers) Hanno postato almeno una volta: [cortex, coco, brio, crash]
(writtenBy) crash ha scritto: [com.SocialNetwork.Post@7699a589, com.SocialNetwork.Post@58372a00]
(containing) i Post con "like:" [com.SocialNetwork.Post@58372a00, com.SocialNetwork.Post@4dd8dc3, com.SocialNetwork.Post@6d03e736, com.SocialNet
(guessFollowers) rete sociale derivata: {cortex=[gabry, crash], gabry=[], utente1=[], brio=[], crash=[brio]}
[metionedUsers(ps)] utenti mezionati in list: [cortex, gabry, utente1, brio, crash]
(filteredSocialNetwork) Post offensivi: [com.SocialNetwork.Post@2d98a335]
```

Riassunto

Per poter utilizzare la classe SocialNetwork come richiesto basta:

- Creare un SocialNetwork
- Utilizzare il metodo ".createUser(String username)" per registrare un utente con quello *username* dentro il Social
- Utilizzare il metodo ".post(String username, String text)" per far postare all'utente *username* un Post con *text* come testo del Post
- Per mettere like, utilizzare il metodo .post(String username, String text) dove *text* comincia con "like:" e l ID del Post a cui username vuole mettere like

A questo punto è possibile utilizzare i metodi richiesti dalla specifica.