Primo Progetto Intermedio

Leonardo Scoppitto

September 2020

Compilazione ed esecuzione

Struttura del codice

Post

Nella classe `Post`, definita dall'interfaccia `PostInterface`, definisco la struttura che avranno i post all'interno di *MicroBlog* e i metodi per recuperare informazioni e visualizzarli. In particolare, rispetto alle informazioni richieste nella consegna del progetto, ho aggiunto: 1. `TreeSet<String> followers`: Questo è il set che salva gli utenti che seguono il post. Ho scelto un `TreeSet` per avere sempre gli utenti ordinati in ordine lessicografico una volta che adrò a stampare a schermo il post, rendendo quindi più fruibile quest'informazione (che alla fine è una di quelle che interesserà all'utilizzatore di MicroBlog). Essendo le interaizoni sotto al singolo post contenute, Non avrei un beneficio prestazionale apprezzabile rispetto all'usare un `HashSet`, inoltre non ho scelto una lista così da non dover gestire i duplicati.

2. `boolean familyFriendly`: è un booleano che indica se il contenuto è adatto a un pubblico sensibile (`true`) o no (`false`). Verrà utilizzato una volta implementata la sottoclasse `FamilyFriendlySocialNetwork` che fornisce le strutture e i metodi per segnalare e oscurare i post che possono risultare offensivi per qualcuno.

I metodi implementati nella classe `Post` sono quindi:

- `getAuthor`: restituisce la stringa contenente l'autore del post
- `getText`: restituisce il testo del post
- `getId`: restituisce l'ID del post
- `getTime`: restituisce il Timestamp della creazione del post
- `getFollowers`: restituisce il Set contenente i followers del post
- `addFollow`: riceve in input una stringa contenente il nome del follower e lo aggiunge al Set `Followers`
- `printPost`: stampa a schermo il post formattato
- `compareTo`: ridefinisce la funzione presente nell'interfaccia `comparable` per l'ordinamento secondo l'ID dei post
- Metodi *setter*:
    1. `setFamiliFriendlyOn()`: modifica `this.familyFriendly` settandolo su `true`.
    2. `setFamilyFriendlyOff()`: modifica `this.familyFriendly` settandolo su `false`.

SocialNetwork

La classe `SocialNetwork` è il fulcro del progetto e contiene tutto il necessario per creare e far funzionare una rete sociale. Nel corso della relazione si parlerà di `follower` e mai di `like`, questo per mantenere una coerenza di termini durante tutto lo sviluppo del progetto e della relazione, usando quindi la convenzione per cui se una persona *segue un post*, *segue l'utente*. Le strutture dati che ho scelto di usare per l'implementazione di un SocialNetwork sono:

1. `HashMap<String, Set<String>> followers`: è una hashmap che ha come `Keys` gli utenti che hanno postato sul SocialNetwork, e come `Values` i followers di quell'utente.
2. `HashMap<String, Set<String>> followed`: è una hashmap che ha come `Keys` gli utenti che hanno avuto interazioni con gli altri utenti, anche se non hanno postato. Le `Values`, infatti sono `Set` di utenti che vengono seguiti dall'utente nella chiave, così da avere sia una mappa `utente -> persone che lo seguono` che una mappa `utente -> persone che sono seguite da utente`. Questo mi permette di implementare efficientemente il metodo `influencers`, come vedremo dopo e di avere una panoramica dettagliata sulle interazioni che sono avvenute all'interno del SocialNetwork istanziato.
3. `HashSet<Post> postSet`: è un set contenente tutti i post pubblicati nel SocialNetwork. Ho scelto un `TreeSet` poiché in questo modo ho la garanzia che l'ordinamento sia mantenuto nel tempo, al prezzo di un inserimento in tempo logaritmico invece che costante (come negli `HashSet`).
4. `int idCounter`: è semplicemente un contatore inizializzato a 0 che viene incrementato ogniqualvolta viene inserito un post.

Metodi per l'interazione col SocialNetwork:

`void addPost()`

Aggiunge un post all'interno di `postSet`, lanciando un'eccezione nel caso in cui il testo fosse più lungo di 140 caratteri (`IllegalLengthException`), nel caso il testo sia vuoto (`EmptyTextException`) o anche nel caso l'username inserito sia `null` o non contenga caratteri (`IllegalArgumentException`).

`void follow()`

Aggiunge l'autore del post che si vuole seguire (chiave) e il follower (valore) all'hashmap `followers`, lanciando l'eccezione `AutoFollowException` se l'utente prova a seguirsi da solo o `IllegalArgumentException` nel caso l'id o l'utente non rispettino i parametri della specifica. Questo metodo (pubblico), ne chiama uno privato (`void addFollowed()`), che popola la mappa `followed`, in modo che siano entrambe sempre pronte a essere utilizzate.

Metodi per la formattazione dell'output

Sono principlamente 3 metodi che stampano a schermo delle informazioni utili:

1. `printSocialNetwork()`: stampa la rete sociale contenuta in `this.followers`.
2. `printSocialNetworkbyFollowed()`: stampa la rete sociale contenuta in `this.followed`.
3. `printAllPosts()`: stampa tutti i post contenuti in `this.postSet`.

Metodi *getter*:

I metodi *getter* sono 3:

1. `getPostSet()`: restituisce `this.postSet`.
2. `getFollowed()`: restuisce `this.followed`.

3. `guessFollowers()`: restutuisce `this.folloers`. Questo metodo non si chiama `getFollowers()` poiché era richista l'implementazione di un metodo `Map<String, Set<String>> guessFollowers(List<Post> ps)` che ha la stessa funzione ma a partire da una lista di post data, mentre per come ho costruito io la classe `SocialNetwork`, ho già una mappa conle stesse proprietà pronta da essere restituita.

Metodi di cui era richiesta l'implementazione

`Map<String, Set<String>> guessFollowers(List<Post> ps)`

Prima di tutto creo una mappa dove salvare l'output (`networkByFollowers`, sempre un'HashMap per motivi di prestazioni), dopodiché per ogni post all'interno di `ps` controllo se l'autore del post è già stato inserito all'interno di `networkByFollowers`: se è già stato inserito, aggiungo i follower del post al `Set` associato all'autore; se non è stato inserito inizializzo un `HashSet` e inserisco la coppaia chiave (autore del post) e valore (set contenente i followers) alla mappa.

`List<String> influencers(Map<String, Set<String>> followers)`

Questo metodo restituisce una lista di utenti che hanno più followers che followed a partire da una mappa `followers` fornita come parametro. Per verificare quali utenti soddisfino questa condizione, costruisco una `followedMap` con la stessa proprietà di `this.followed` e poi confronto per ogni utente (`Key`) la grandezza dei `Set` contenente i *seguiti* e i *seguaci*. Chi soddisfa la condizione viene inserito nella lista `influecers`, che verrà poi ordinata e restituita. Di seguito il codice dell'implementazione:

```
List<String> influencers = new ArrayList<>();
for (Map.Entry<String, Set<String>> entry : followers.entrySet()) {
  for (String people : entry.getValue()) {
      if (followedMap.get(people) == null) {
          toadd = new HashSet<>();
          toadd.add(entry.getKey());
          followedMap.put(people, toadd);
      } else
          followedMap.get(people).add(entry.getKey());
  }
}
// Abbrevio followedMap con fMap altrimenti non entra nei margini
for (Map.Entry<String, Set<String>> entry : followers.entrySet()) {
  if ((fMap.get(entry.getKey()) == null && !entry.getValue().isEmpty())
  || entry.getValue().size() > fMap.get(entry.getKey()).size())
    influencers.add(entry.getKey());
}
Collections.sort(influencers);
return influencers;
```

Ho poi definito un metodo `List<String> influencers()` che opera solo con `this` e non riceve parametri in input, così da poter calcolare i followers dell'istanza di `SocialNetwork` con cui stiamo lavorando. Il codice è molto simile, infatti manca solo la parte della costruzione della mappa `followedMap` di supporto in quanto in questo caso abbiamo già tutto quello che ci serve per il calcolo:

```
List<String> influencers = new ArrayList<>();
for (Map.Entry<String, Set<String>> entry : followers.entrySet()) {
  if ((followed.get(entry.getKey()) == null && !entry.getValue().isEmpty())
          || entry.getValue().size() > followed.get(entry.getKey()).size())
      influencers.add(entry.getKey());
}
Collections.sort(influencers);
```

```
return influencers;
```

Come possiamo vedere, se abbiamo già le mappe a disposizione, l'operazione viene svolta molto più velocemente ed efficientemente.

`Set<String> getMentionedUser(List<Post> ps)`

Questo metodo restituisce la lista degli utenti che hanno contribuito al SocialNetwork scrivendo almeno un post. Il metodo quindi prende una lista di post e la scorre aggiungendo l'autore a un `Set` di stringhe che verrà restituito al termine dell'operazione. Il funzionamento è molto semplice:

```
Set<String> mentionedUsers = new TreeSet<>();
for (Post post : ps)
    mentionedUsers.add(post.getAuthor());
return mentionedUsers;
```

Anche in questo caso ho definito un altro metodo `Set<String> getMentionedUser()` che non prende in input nessun parametro, ma esegue la stessa operazione su `this.postSet`, restituendo la lista di tutti gli utenti che hanno postato sul SocialNetwork.

`List<Post> writtenBy(String username)`

Questo metodo prende come parametro un nome utente e restituisce la lista dei post pubblicati da quell'utente, opera come `getMentionedUser()`, ma salva l'intero post e non solo il nome utente:

```
if (username.isBlank())
    throw new IllegalArgumentException("Username non valido");
List<Post> wroteBy = new ArrayList<>();
for (Post post : this.postSet)
    if (post.getAuthor().equals(username))
        wroteBy.add(post);

if (wroteBy.isEmpty()) {
    System.out.println("Nessun post trovato :(");
    return wroteBy;
}
Collections.sort(wroteBy); // Ordino in ordine cronologico inverso i post (dal più recente al p
return wroteBy;
```

Il metodo `List<Post> writtenBy(List<Post> ps, String username)` funziona nello stesso modo, semplicemente esegue la ricerca su `ps` e non su `this.postSet`.

`List<Post> containing(List<String> words)`

Questo metodo ricerca una lista di parole all'interno dei post pubblivati all'interno del SocialNetwork. Il suo funzionamento consiste nel fare un parsing del testo di ogni post e confrontare ogni parola con quelle prestenti in `word`, restituendo una lista di post tali che contengano almeno una parola definita dall'espressione regolare `word.toLowerCase()[a-z]*`. inoltre, uso un flag `boolean found` per indicare quando una parola è stata trovata, così da interrompere la ricerca e passare al post successivo. Di seguito il codice:

```
List<Post> contains = new LinkedList<>();
String[] parsedText;
boolean found;
for (Post toScan : this.postSet) {
    found = false;
    for (String word : words) {
        if (word.isBlank())
```

```
                throw new IllegalArgumentException("Stringa non valida");
            parsedText = toScan.getText().split("[^a-zA-Z]+");
            for (String parsedWord: parsedText) {
                if (parsedWord.toLowerCase().matches(word.toLowerCase() + "[a-z]*")) {
                    contains.add(toScan);
                    found = true;
                    break;
                }
            }
            if (found)
                break;
        }
    }
    if (contains.isEmpty()) {
        System.out.println("Non ho trovato risultati :(");
        return contains;
    }
    return contains;
```

FamilyFriendlySocialNetwork (Parte 3)

Ho pensato di estendere la classe `SocialNetwork` con la sottoclasse `FamilyFriendlySocialNetwork` introducendo dei metodi che permettessero di segnalare i post sia direttamente tramite il loro ID, che tramite una lista di *badWords*, così da popolare un dizionario, inizialmente vuoto, di parole non adatte a un pubblico sensibile. Ho scelto di implementare questo metodo di segnalazioni progressive in modo che sia l'utenza stessa a decidere cosa è ammesso e cosa no all'interno della loro rete sociale.

Funzionamento

Ogni volta che viene aggiunta una parola al dizionario, la lista dei post ammessi viene aggiornata e verrà impedita la pubblicazione di nuovi post non conformi alle regole. Rimane comunque possibile ripristinare un post segnalato per errore e visualizzare l'id dei post che sono stati rimossi oltre a rimuovere una parola dal dizionario e ripristinare di conseguenza tutti i post che contenevano quella parola. Inoltre, ho effettuoato un override dei metodi di `SocialNetwork` che richiedono di interagire con i post, così da bloccare i contenuti offensivi ogniqualvolta vengono chiamati tali metodi. In particolare non sarà possibile seguire un post segnalato (`follow()`) e `writtenBy` e `containing` restituiranno solo i post in cui il flag `familyFriendly` è `true`.