Mail Service

We want to code in Java a mailing system where different users can send messages to each other. Messages will be saved in a message store and users can send or receive messages to the store. We will also need some administrative operations to manage mailboxes.

Review notes:

- The reviews will value both correctness and quality of the code.
- All classes should be documented (javadoc).
- Please create unit test to validate your code (JUnit).

Create a package for each part. Parts are incremental, so they should use the classes in packages from previous parts. If a part requires to add new features to a class, try to extend the one from the other package with the new code and simply use the new class in the main program of the part. Each part has one or more main programs that use the functionality of that part.

PART 1 - OOP

This problem includes different entities: Users, Messages, Mailbox, and Mail Store.

Each user contains a username, name, and year of birth.

Messages are composed of a subject and a body (text) and contain a sender (from) and a receiver (to) (both using the username); additionally, we will store the message creation timestamp, so that we can sort them by new.

All messages from all users will be stored on a backend service: a Mail Store. We want to be able to use different kinds of mail stores, so that we can change the backend service in the future. The stores need to provide the following operations:

1. Send mail: Send a new message to a user.
2. Get mail: Retrieve all the messages that are intended for a certain user.

For now, we will have two types of mail store:

1. In memory: Keeps the messages in memory using Java objects. It will store messages on a structure that allows to directly obtain the list of messages for each receiver.
2. On a file: Messages are appended to a file when sent and read from the file when retrieved, filtered for the user that is retrieving them. One message per line and each message field separated with “;”. We assume messages will not contain this delimiter.

A Mailbox represents a mail account and implements the user mailing operations. It contains the user and an internal list of the messages that has received. It also has a reference to the mail store to which it will ask to retrieve the user's messages. The user tells the mailbox to update its messages by retrieving them from the mail store.

A mailbox has the following operations:

1. Update mail: The mailbox connects to the mail store and retrieves all messages whose receiver is the user. Depending on the backend used, the mail store could return the mail in arbitrary order. Sort it by sent time, newer first, before saving it to the internal list. (streams)
2. List mail: Return the list of messages that has been previously retrieved from the mail store.
3. Send mail: Send a message to the mail store. This operation needs a destination, a subject and a body since the sender is the mailbox's user.

4. Get the mail sorted by some given method. (streams)
5. Filter user mail: Apply a filter to the user list of messages. This operation returns a list of messages that fulfill a given condition. (streams)

Make the mailbox iterable through its messages. So that we can use the following constructions:

```
for(Message m: mailbox){
    System.out.println(m);
}
mailbox.forEach(System.out::println);
```

The system will have an administrative component that manages the system (MailSystem). This component is responsible for the mail store and the lists of users and mailboxes. It can be useful to quickly obtain users and mailboxes by their usernames. The system operations are (most can be performed with Java 8 streams):

- Create new user: Creates a new user, their mailbox, and puts them on the administrative data-structures. Creating a user returns its mailbox, that contains user operations.
    - The mailbox can be retrieved later by giving the username (log in).
- Get all messages in the system.
- Get all users in the system.
- Filter messages globally: Get all messages in the system that fulfill a condition.
- Count total number of messages.
- Average messages per user.
- Group messages per subject. Any user.
- Count the words of all messages from users with a particular name.
- Get messages to users born before a certain year.

To test the system, create a main code that:

1. Initialize the mail system with an in-memory mail store.
2. Create at least 3 users, two have the same name but different username.
3. Then, use the mailboxes to send a few emails between them. Make some of them share the same subject and make enough so that the following tests have results.
4. Get one of the mailboxes and update its mail.
5. List the mailbox messages in the console. (Sorted by newer first.) Use the iterable capabilities of the mailbox!
6. Now list the messages by sender username using the mailbox feature.
7. Filter the messages with the following conditions:
    - The message subject contains a certain word.
    - The message sender is a certain user.
8. Use the mail system object to retrieve all messages and print them.
9. Filter messages globally that fulfill the following conditions:
    - The message subject is a single word.
    - The sender was born after year 2000.
10. Get the count of messages in the system and print it.
11. Get the average number of messages received per user and print it.
12. Group the messages per subject in a `Map<String, List<Message>>` and print it.
13. Count the words of all messages sent by users with a certain real name.
14. Use the name that you used on two users. Print the result.
15. Print the messages received by users born before year 2000.
16. Now change the mail store to the file implementation.

Develop a CLI (command line interface) with the following operations (system operations):

1. createuser <username> <args>... : Create a new user as admin
2. filter <...> : Filter at a system level. The program has implemented several conditions for filtering messages and can be referenced from the command. For instance:
   - contains <word> : filters all messages that contain the word in the body or subject.
   - lessthan <n> : filters messages that contain less than n words in the body.
3. logas <username> : Log in as a user. No passwords.

Logged as a user (mailbox operations):

1. send <to> "subject" "body" : send a new message.
2. update : retrieve messages from the mail store.
3. list : show messages sorted by sent time.
4. sort <> : sort messages by some predefined comparators.
5. filter : same options that for system, but just for the user messages.

Create another main program that starts the CLI. You can populate the system with users and messages before opening the CLI.

[OPTIONAL] Make the CLI a GUI using Swing.

PART 2 – PATTERNS

Automatic message filters

Use the Observer pattern to create an automatic mail filter. We want users to be able to define filters that are automatically applied to their mailboxes when updating their mail. The goal is to discard some of the messages so that they are not returned when listing the mail as usual. Filtered messages will be stored in a new list of spam in the mailbox. Mailboxes will have a new operation to list all spam. Sorting is not necessary.

The filters are subscribed to a mailbox and will observe all mail updates. If a filter matches a message, it will be removed from the mailbox's list of messages and stored in the spam list of the same mailbox. Note that mail updates should pass both lists to the filters that are subscribed.

- Create a "SpamUserFilter" that filters messages whose sender username contains "spam".
- Create a "TooLongFilter" that filters messages with a body with more than 20 characters.

Create a main program that initializes the system like in the first part. But now uses the new mailboxes instead of the normal ones. And set up some spam filters that will detect some messages. List the message and spam list to show that the filters work.

Using streams, obtain the users that have sent any message that got filtered as spam.

Encoding message bodies

We are interested in security and want our mail store to save the messages encrypted. Use the Decorator pattern to create an abstract *wrapper for the mail store* class that will process the body (and only the body) of the messages sent by applying some transformation from string to string. This process is done when sending a message to the store.

Likewise, when getting the mail, we need to undo the process. Then, the wrapper should also modify the get mail operation to apply the reversed process to all messages retrieved.

The process methods will be resolved with the strategy pattern. We will implement two concrete methods of processing:

- The first one simply reverses the string.
- The second one actually ciphers the message.

Use the following code to cipher strings:

Create cipher object:

```
String key = "IWantToPassTAP12";  // 128 bit key
java.security.Key aesKey =
      new javax.crypto.spec.SecretKeySpec(key.getBytes(), "AES");
Cipher cipher = Cipher.getInstance("AES");
```

Encrypt with:

```
byte[] encrypted = new byte[0];
try {
    cipher.init(Cipher.ENCRYPT_MODE, aesKey);
    encrypted = cipher.doFinal(body.getBytes());
} catch (Exception e) {
    e.printStackTrace();
}
return Base64.getEncoder().encodeToString(encrypted);
```

Decrypt with:

```
byte[] encrypted = Base64.getDecoder().decode(body.getBytes());
String decrypted = null;
try {
    cipher.init(Cipher.DECRYPT_MODE, aesKey);
    decrypted = new String(cipher.doFinal(encrypted));
} catch (Exception e) {
    e.printStackTrace();
}
return decrypted;
```

This is a very simple code for the sake of the exercise, NOT SECURE CODE.

Create a simple main program that creates a mail system with the File mail store. Use the code from the first part that initializes a few users and sends some mail. Also, the part that prints the mails in one of the mailboxes.

Now decorate the File mail store with the reverser processor. If you run it, you should see the messages in the console as usual, but the file should contain the messages reversed.

For instance, a message that initially would be stored to the file like:

```
pedro;daniel;TAP;Student marks;1602251056550
```

Now would appear like:

```
pedro;daniel;TAP;skram tnedutS;1602251056550
```

Now, wrap it using the cipher and check the file. And the same message would now be:

```
pedro;daniel;TAP;7dhHa8mWwkNaxUDKbnqIZw==;1602251056550
```

Finally, chain both decorators to store messages reversed and encrypted. Which stores:

```
pedro;daniel;TAP;9v8ZO/cqSiVH5xCBYsrzqQ==;1602251242334
```

!!! Clear the store file between executions or change the file name. Otherwise, using different wrappers will produce errors.

PART 3 - PATTERNS 2: REDIS

https://redis.io/

Redis is a key-value store where different clients can connect to save and retrieve data. Much like a dictionary, the basic operations are `set(key, value)` and `get(key)`. Since it is language agnostic, both keys and values are only strings.

https://redis.io/commands/set https://redis.io/commands/get

We want to use a Redis store as a new backend for a mail store. That way, different clients can send and receive mail at different JVMs. For that, messages must be converted to string in the same way we used for the file mail store. However, the key-value structure of Redis allows to save the messages like we did in the in-memory option so that we can retrieve all messages to the same user at once. Look at the list capabilities of Redis:

https://redis.io/commands/lpush https://redis.io/commands/lrange

For this exercise, you will need to run a Redis server locally. You can easily download and run a Redis server on a UNIX system as explained in https://redis.io/download. On Windows, you can use WSL or other emulations. If you are familiar with Docker, it is also an option: https://hub.docker.com/_/redis.

To connect to the server from Java, we will use the Jedis client. You can add to the project libraries in IntelliJ from maven with the name:

```
redis.clients:jedis:2.9.0
```

Jedis is very simple to use. To connect to local host, create a Jedis object, which implements all Redis commands:

```
Jedis jedis = new Jedis("localhost");
jedis.set("foo", "bar");
String value = jedis.get("foo");
```

The Redis mail store will adapt the mail store interface to the Jedis client (Adapter). In addition, to avoid multiple connections to the server, we want this class to have a unique instance (Singleton).

Run the same main program from Part 1 but using the new store. You can check the data in the server connecting with the Redis command line client and delete all mails with the FLUSHALL command.

Mail Store Factory

Use the Factory pattern so that the mail system receives a mail store factory in its constructor instead of a mail store directly. That way, the mail system only needs to call `createMailStore()` to any implementation to initialize the store.

Implement three concrete Factories:

- Create the memory store.
- Create the Redis store.
- Create the file store with both wrappers from Part 2.

Create a main program that allows to pass different factories to the mail system.

PART 4 - REFLECTIVE PROGRAMMING

The goal is to introduce dependency injection mechanisms in the MailSystem of Part 1 using reflection. In particular, we must create an annotation (Config) with two fields store (String) and log (boolean). The "store" field will contain the name of the class including its package (part1.MemMailStore) for the selected MailStore we want to use (FileMailStore, MemMailStore, etc.). The log field will activate or deactivate login capabilities over the mail store.

This Config annotation will be included in the mail system class, so that we can remove the mail store parameter from the constructor of this class. Inside this class, we will create a method that will read the annotation, obtain the selected mail store, and create an instance dynamically.

Furthermore, using an Interceptor with a Dynamic Proxy (InvocationHandler) we will create a log class that keeps a log of invocations over the mail store showing them in the screen. When the log is enabled in the annotation, the same method from above will wrap the dynamically created instance with our Interceptor, which will log all calls to mail store methods.