Clinic v2.0

Features:

* Add patient (optional. writes to file)
* Remove patient (optional. removes from file)
* List patients
* Print information for particular patient.
  + By name
  + By PatientId

**Checkpoint 1.**

Create the following:

* Class:
  + *ClinicRunner*
  + HumanClinic
  + HumanPatient
  + PetClinic (optional)
  + PetPatient (optional)
  + ClinicFileReader
  + ClinicKeyboardReader (optional)
* AbstractClass
  + AbstractClinic
  + AbstractPatient
* Interface
  + ClinicReader
* Enum
  + PatientTypes (optional)

Files. These need to be in the src/main/resources folder.

***human\_patients.txt***

1,radu,3

2,mihai,5

3,diana,2

4,liviu,3

5,relu,1

6,vincentiu,4

***human\_problems.txt***

1,raceala

2,durere de cap

3,gripa

4,insomnie

5,sinuzita

***pet\_patients.txt***

1,Fluffy,3

2,Puffy,2

3,Dodo,2

4,Mumbo,3

5,Cola,1

6,Pepsi,1

***pet\_problems.txt***

1,preaJucaus

2,racit

3,carii

**Checkpoint 2. Defining first methods in ClinicReader.**

First of all, we will need to have two methods in ClinicReader.

List<AbstractPatient> readPatients() throws IOException, URISyntaxException;

Map<Integer, String> readProblems() throws IOException, URISyntaxException;

This means that we will, somehow, in our implementations read both patients and problems for them.

Since we can have both human and pet patients, files for both human patients/problems and pet patients/problems are provided.

**Checkpoint 3. Reading data**

We need to read data to add our patients. We are going to create two classes that use the ***ClinicReader*** interface.

Initially, we want to read data only from the file. That means that we’ll make sure that the ***ClinicFileReader*** class will implement the ***ClinicReader*** interface.

At this time, ClinicFileReader will have bodies for the two methods and will return null. We need to write the code that reads from a file called human\_patients.txt in the readPatients() method and we will need to write code that reads from a file called human\_problems.txt in the readProblems() method.

**[Optional]**

Create a constructor so that you can use ClinicFileReader to work with either Human or Pet type patients by receiving the type from an enum. The enum will have the file names stored in it. That enum will have a constructor that receives two parameters. First parameter will be the name of the file for a given patient type. The second parameter is the name of the file of the problems for that specific patient type.

The enum needs to have two private String fields that will be set in the constructor of the enum. There need to be getters for the two fields.

This way we decouple the FileReader from the PatientType filenames.

Now, you will need to write the actual code that will read the patients and the problems from files called human\_patients.txt and human\_problems.txt ;

We will need to use the following code to read from a file called ‘human\_patients.txt’ :

URI patients = ClassLoader.getSystemResource("human\_patients.txt").toURI();

Path pathOfPatients = Paths.get(patients);

List<String> strings = Files.readAllLines(pathOfPatients);

This code will be common between the patients and problems methods so you can re-use it. This means that you would need to create a method that receives a parameter of the file name that you will want to use.

Create a method called *private List<String> readFileByName(String fileName)*. This method needs to now contain the code from above. This method also needs to return the strings that it reads.

Now, that particular method will not be super useful by itself, as you now need to create the list of patients as well. On to Checkpoint 4.

**Checkpoint 4. Creating patients**

The abstract class AbstractPatient needs to have the following:

* An int patientId;
* A String patientName;
* A constructor to assign those two fields.
* Getters for the two fields.

**[Optional]**

* A String called problemName
* An additional constructor to match the new field.This constructor will have three parameters now.
* A getter for the new field.

**Checkpoint 5. Creating human patients.**

The humanPatient class needs to extend AbstractPatient. It will need to have a constructor that calls the two parameter super constructor of AbstractPatient.

**[Optional]**

* Another constructor to match the three parameter constructor created in AbstractPatient

Add code in the constructor of HumanPatient so that when a new HumanPatient object is created, we print to the console (sout) the name of the patient and the fact that has has been added. The message should read like:

‘Human patient *Mirel* has been added. Get well *Mirel*!”. Mirel will be the variable name that is received in the constructor.

**[Optional]**

* In the additional constructor, print a message like:

‘Human patient ***Mirel*** suffering from ***raceala*** has been added. Get well ***Mirel***!”

Mirel will be value of the variable patientName and raceala can be the value of the variable problem. This part can be completed in a following checkpoint, after reading problems from a file and populating a map

Notice that the indented text is the value received as a parameter.

The HumanPatient class must have a toString method. You can generate this from intellij. Use getters to retrieve the name of the patient. Or maybe make sure that the fields in the parent class are protected and use them from there.

**Checkpoint 6. Populating patients from file.**

Now that we’re able to create objects of type humanPatient we can continue to write code in the ClinicFileReader class.

Our method readPatients will now need to use the List<String> that we read from the

readFileByName("human\_patients.txt") method.

We know that each string represents a whole line. But we also need to split those lines into usable ‘columns’ so that we can create the Humanpatient objects.

Since we have access to HumanPatient constructors we just need to iterate through all the strings we received from the readFileByName method, and split them up.

We know, by reading the text files with data, that the first column is the id of the patient, the second column is the name of the patient and the third column is the id of the problem that they’re suffering from.

We will just create simple HumanPatient objects without caring about the problem that they have. We need to use the two parameter constructor that we previously made.

As we’re iterating through all the strings using a **for,** we can create new HumanPatient objects and pass split strings to the constructor.

We will need to have created an object of type List<AbstractPatient> initialized with the concrete type of ArrayList. This variable should be named patients.

It is here where we’ll **add** in the new HumanPatient objects. But before we can do that we need to use the split method on each string from the list of strings that we’ve read.

This will create an array of strings that we can call patientParameters. From here, we can obtain the patientid, patient name and optionally the problemId.

Note that since the patientId is int, you will need to use the parseInt() method of the IntegerClass. This would look like :

int patientId = Integer.valueOf(patientParamters[0]);

Then you need to create, in the for, objects like name and problem id, that can be found on the 1 index and 2 index of the patientParameters.

After which you can create a new temporary human patient object, and then add it to the patients list. We can do that like this :

AbstractPatient temporaryPatient = new HumanPatient(patientId, name);

patients.add(temporaryPatient);

After all this, we can finally return the patients object since it should be fully created.

You should be able to test this in the clinic Runner by having the following code :

ClinicReader clinicReader = new ClinicFileReader();

List<AbstractPatient> patients = clinicReader.readPatients();

System.out.println(patients);

This code should now print out :

Human patient radu added. Get well radu!

Human patient mihai added. Get well mihai!

Human patient diana added. Get well diana!

Human patient liviu added. Get well liviu!

Human patient relu added. Get well relu!

Human patient vincentiu added. Get well vincentiu!

[HumanPatient{patientId=1, patientName='radu', problemName='null'}, HumanPatient{patientId=2, patientName='mihai', problemName='null'}, HumanPatient{patientId=3, patientName='diana', problemName='null'}, HumanPatient{patientId=4, patientName='liviu', problemName='null'}, HumanPatient{patientId=5, patientName='relu', problemName='null'}, HumanPatient{patientId=6, patientName='vincentiu', problemName='null'}]

Note that the line breaks won’t appear in the console, its just how the code appears copied in this document.

At this point in the application, you should have about 100 lines of code written.

Right now you application will have the capability to work with current patients written in the file, or if you add new patients in the file you can have more patients to work with.

**Checkpoint 7. Creating the AbstractClinic.**

Now that we’ve added the mechanism to read patients from a file, we can now add them to a clinic. The mechanism to add patients to the clinic should be similar to the previous mechanism, but now we’re going to be using a Map<AbstractPatient> to store them.

The reason is that if we want to retrieve a patient by id, the map data structure is easier than a list.

Thus, the AbstractClinic abstract class will have the following :

* A field of type Map<Integer, AbstractPatient> -> named currentPatients
* The following abstract methods :
  + public abstract void addPatient(AbstractPatient patient);
  + public abstract void addBulkPatients(List<AbstractPatient> patients);
  + public abstract void removePatientByPatientObject(AbstractPatient patient);
  + public abstract void removePatientByPatientId(Integer patientId);
  + public abstract void listPatients();

**Checkpoint 8. Implementing logic in HumanClinic**

First thing to do is to implement the logic of addPatient(AbstractPatient patient).

This method will recieve a patient. It will need to put in the currentPatient map the patient that was recieved as a parameter using the following logic : key needs to be patient id. This can be gotten from the abstractPatient object. The value needs to be the actual AbstractPatient object.

That’s it.

**[Optional]**

If the abstractpatient is null throw an IllegalArgumentException

Second we need to make sure that we can remove the patient. This can be done in two ways. First way is to remove the patient by patientId. This is the simplest method. The currentPatients object has a method called **remove** which takes in the key value. It will remove the value of the corresponding key.

The second way is to pass in a AbstractPatient object and call the following code:

*currentPatients.values().remove(patient);*

Now, we will need to implement the method addBulkPatients(List<AbstractMethod> patients).

This method will just iterate through the patients that we’ve recieved as parameters, and in turn call the addPatient() method.

Last step is to implement the method listPatients(). This will just iterate through the keyset of the currentPatients map.

Use an enhanced for,for this.

The code should look similar to someting like :

for (Map.Entry<Integer, AbstractPatient> ap : currentPatients.entrySet()){

System.out.println(ap.getKey() + " " + ap.getValue().patientName);

}

Now, in your runner, if you add the following code:

AbstractClinic clinic = new HumanClinic();

clinic.addBulkPatients(patients);

clinic.listPatients();

You should see something like what is below, in addition to what was printed before.:

1 radu

2 mihai

3 diana

4 liviu

5 relu

6 vincentiu