Lab Exercise 7: Queues and Stacks
CS 2334

March 07, 2019

Introduction

In this lab, you will experiment with two common data structures: stacks and queues. You will also be working more with abstract classes and abstract methods.

Learning Objectives

By the end of this laboratory exercise, you should be able to:

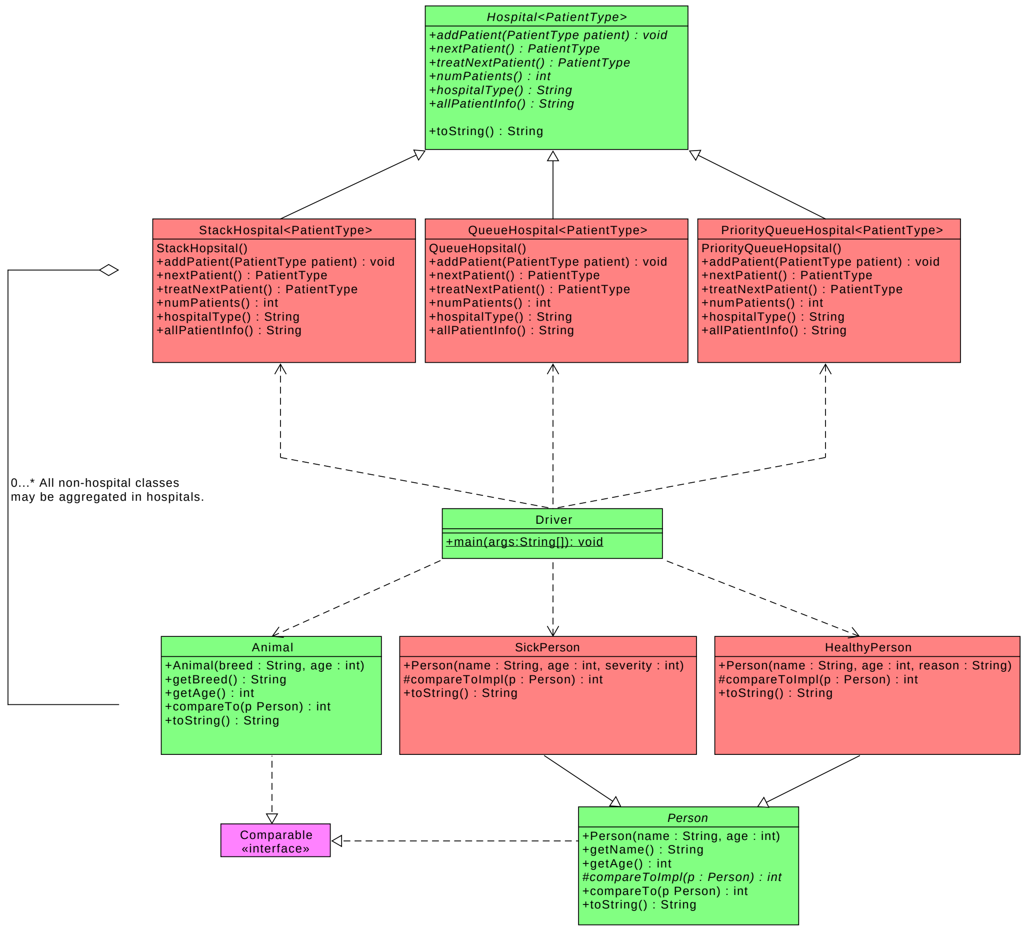
1. Use and create stacks to store and retrieve objects
2. Use the create queues to store and retrieve objects
3. Create and use Abstract Classes and Methods
4. Create and use Generic Classes
5. Read, understand, and create code for javadocs.

Proper Academic Conduct

This lab is to be done individually. Do not look at or discuss solutions with anyone other than the instructor or the TAs. Do not copy or look at specific solutions from the net.

Class Design

Below is the UML representation of the set of classes that make up the implementation for this lab. For this lab, you will be given limited code. You will need to read and precisely follow the UML, this documentation, and the provided javadocs to properly create your lab. **Only public methods are documented. You may create private variables and methods of any type as you see fit. You may not create additional public methods or variables.**


Hospital<PatientType>
+addPatient(PatientType patient) : void
+nextPatient() : PatientType
+treatNextPatient() : PatientType
+numPatients() : int
+hospitalType() : String
+allPatientInfo() : String
+toString() : String
StackHospital<PatientType>
StackHopsital()
+addPatient(PatientType patient) : void
+nextPatient() : PatientType
+treatNextPatient() : PatientType
+numPatients() : int
+hospitalType() : String
+allPatientInfo() : String
QueueHospital<PatientType>
QueueHopsital()
+addPatient(PatientType patient) : void
+nextPatient() : PatientType
+treatNextPatient() : PatientType
+numPatients() : int
+hospitalType() : String
+allPatientInfo() : String
PriorityQueueHospital<PatientType>
PriorityQueueHopsital()
+addPatient(PatientType patient) : void
+nextPatient() : PatientType
+treatNextPatient() : PatientType
+numPatients() : int
+hospitalType() : String
+allPatientInfo() : String
0...* All non-hospital classes may be aggregated in hospitals.
Driver
+main(args:String[]): void
Animal
+Animal(breed : String, age : int)
+getBreed() : String
+getAge() : int
+compareTo(p Person) : int
+toString() : String
SickPerson
+Person(name : String, age : int, severity : int)
#compareToImpl(p : Person) : int
+toString() : String
HealthyPerson
+Person(name : String, age : int, reason : String)
#compareToImpl(p : Person) : int
+toString() : String
Comparable
«interface»
Person
+Person(name : String, age : int)
+getName() : String
+getAge() : int
#compareToImpl(p : Person) : int
+compareTo(p Person) : int
+toString() : String

The coloring of the classes indicates what work you need to do to complete the lab assignment. This is done for your convenience; you can also follow TODOS and the other lab instructions to understand what you need to do. The colors indicate as follows:

1. Green: A class/interface that is completely implemented. You do not need to edit these.

2. Yellow: A class/interface that is partially implemented. You will need to look for TODOs and read the writeup to determine what else is needed to complete the class.

3. Red: A class/interface that is completely unimplemented. You will need to create this class and ensure that it is fully functional.

4. Purple: This is a class/interface provided in standard Java. You can look online for documentation for these classes. You should never attempt to create these classes. Doing so will cause many errors.

Javadoc

This lab has limited code provided to you. This document is mainly an overview of how the classes work and piece together. It does not provide many specific details. Instead, you will read the javadoc provided in your lab to determine how to create your code. This javadoc gives the specification for all your classes and methods. Note that the javadoc is only given for public methods. You may add private methods and variables at your discretion. You will not need to add additional javadoc (you still should document your code, however).

To view the javadoc we have compiled for your lab, once you import the project in eclipse, open the *lab7/doc/index.html* file using your favorite web browser or Eclipse (double clicking in the package explorer will open the web page). This will open up an html page with the documentation for all classes. Use this information alongside the information in your UML to design your code.

The key classes in our project are:

- **Person**: Abstract Class representing a Person who will be admitted to a Hospital. Stores a name and an age.
- **HealthyPerson**: Class representing a specific type of Person admitted to a Hospital - a healthy person. Stores a reason for a visit in addition to a name and an age. HealthyPersons are alphabetically ordered by their names.
- **SickPerson**: Class representing a specific type of Person admitted to a Hospital - a sick person. Stores a severity of their illness in addition to a name and an age. SickPersons are ordered by their illness severity in decreasing order.
- **Animal**: Class representing an Animal to be admitted to a Hospital. Stores a breed and an age. Animals are ordered by decreasing age.
- **Hospital**: An abstract class representing a Hospital. Hospitals store a collection of patients who are waiting to receive medical treatment. Patients are treated one at a time by some defined ordering. How to determine which patient to treat next is the responsibility of the Hospital subclasses. Once a patient is treated, it is removed from the Hospital. Hospitals may have different types of patients. In the case of this lab, the two general patient types used are Person and Animal. This is represented through the generic type "PatientType". The following methods defines how it works:
  - addPatient(): Abstract method. Implementing classes should add the patient passed in into an internal data structure. I.e. the patient should be stored in some way.
  - nextPatient(): Abstract method. Implementing classes should return the patient who will next be treated. Does not treat the patient (i.e. it is not removed from the collection of patients that the hospital still must treat - check what it is without actually affecting it).
  - treatNextPatient(): Abstract method. Implementing classes should treat the next patient and remove them from the Hospital, returning the treated patient. Once someone is healed they no longer need to stay in the Hospital.
  - numPatients(): Abstract method. Implementing classes should return an int indicating the number of patients that are still in the Hospital.

  - hospitalType(): Abstract method. Implementing classes should return their classnames exactly.
  - allPatientInfo(): Abstract method. Implementing classes should return The toString of all patients, concatenated (don't add spaces, newlines, etc...).
  - toString(): Returns some information about the Hospital and its occupancy. Specifically, the string "A %s-type hospital with %d patients.", with replacements of the hospitalType and the number of patients..

- **StackHospital**: An implementation class of Hospital. A StackHospital stores patients in a Stack. Stacks are a "Last In - First Out" or "LIFO" data structure. This means that the last element added to a stack is the first one that is removed from it.

  e.g. we add the elements A,B, and C to a stack in that order ("pushing" the values onto the stack). We then remove 3 elements ("popping") from the stack. They will be removed in the order c, B, A. A Stack is a generic data structure that may be implemented in many ways. E.g. to implement a stack you might use an array, arraylist, linkedlist, hashmap, etc... Java also has some other classes that you may find helpful for creating this class. What is important is that the stack and push/pop elements and that it keeps its LIFO ordering.

  The StackHospital treats patients in a LIFO ordering. The overridden abstract methods should reflect this.

- **QueueHospital**: An implementation class of Hospital. A QueueHospital stores patients in a Queue. Queues are a "First In - First Out" or "FIFO" data structure. This means that the first element added to a queue is the first one that is removed from it.

  e.g. we add the elements A,B, and C to a queue in that order. We then remove 3 elements from the queue. They will be removed in the order that they were inserted in: A, B, C.

  The QueueHospital completes orders in a FIFO ordering. The overridden abstract methods should reflect this.

- **PriorityQueueHospital**: An implementation class of Hospital. A PriorityQueueHospital behaves the same as a QueueHospital unless the patients it is storing have a natural ordering defined. If a natural ordering is defined, then the patients are treated in sorted order (as defined by the natural ordering).

e.g. we add the ordered elements (the natural ordering defines a decreasing order on the second element) (A, 2), (B, 3), and (C, 1) to a queue in that order. We then remove 3 elements from the queue. They will be removed in the order that they were inserted in: (B, 3), (A, 2), (C, 1)

The QueueHospital completes orders in this manner. The overridden abstract methods should reflect this.

Lab 7: Implementation Steps

1. Create and implement all classes from the UML (you should not create comparable, as this is a built-in interface). Remember that you may add private fields and methods.

2. Run the driver class to verify that your code works as a whole.

3. Implement JUnit tests to thoroughly test all classes and methods you created/implemented.

    - You need to convince yourself that everything is working properly
    - Make sure that you cover all of the cases within the methods while creating your tests. Keep in mind that we have our own tests that we will use for grading.

Hints

Look through the java documentation online, particularly for the data structure classes that they have implemented. Be careful with what is a class and what is an interface. There may be some useful classes for implementing your Hospitals.

Submission Instructions

- All required components (TODO list, source code and compiled documentation) are due at 11:59pm on Tuesday, March 12th. Github link is due on canvas.

- Use the same submission process as you used for lab 6. You must submit your implementation to the *Lab 7* area on the Web-Cat server.

Rubric

The project will be graded out of 100 points. The distribution is as follows:

Correctness/Testing: 55 points

The Web-Cat server will grade this automatically upon submission. Your code will be compiled against a set of tests (called *Unit Tests*). These unit tests will not be visible to you, but the Web-Cat server will inform you as to which tests your code passed/failed. This grade component is proportional to the fraction of tests that your code passes (so 27.5 points means that your code passed half of the tests)

Style/Coding: 20 points

The Web-Cat server will grade this automatically upon submission. Every violation of the *Program Formatting* standard described in Lab 1 will result in a subtraction of a small number of points (usually two points). Looking at your submission report on the Web-Cat server, you will be able to see a notation for each violation that describes the nature of the problem and the number of subtracted points.

Design/Readability: 15 points

This element will be assessed by a grader (typically sometime after the lab deadline). Any *errors* in your program will be noted in the code stored on the Web-Cat server, and two points will be deducted for each. Possible errors include:

- Non-descriptive or inappropriate project- or method-level documentation
- Missing or inappropriate inline documentation
- Inappropriate choice of variable or method names
- Inefficient implementation of an algorithm
- Incorrect implementation of an algorithm
- Incomplete coverage of your Unit Tests. We expect that your unit tests will test all lines of your code

If you do not submit compiled javadoc for your lab, 5 points will be deducted from this part of your score.

Note that the grader may also give *warnings* or other feedback. Although no points will be deducted, the issues should be addressed in future submissions(where points may be deducted).

Github: 5 points

You will need to use github when developing your project. The grader will check that you have made several commits (¿= 5) while developing your lab. Commits should have good messages. You will link your github repo on canvas.

Lab Plan: 5 points

Your todo.txt file will be checked to verify its format. A grader will also asses the quality of your lab plan. Your plan should have at least 5 objectives that should be relatively granular (e.g. an objective of "finish lab" is much too large and not a good tasks in a plan). The grader will ensure that your predicted and actual times are reasonable (e.g. not 20000 minutes).