Exam IN1010 spring 2021

Read the whole problem set before starting to answer any questions; useful information may be found later in the text.

The combined answer to problems 1 and 2 shall constitute a complete program with main and all necessary classes. You may deliver your program as one .java file, or you may choose to have each class in a separate file. The .class files are not to be uploaded. Illustrations may either be drawn by hand and photographed using a mobile phone, or generated by a program. Files with illustrations must be named according to which problem they belong to, and they must be in .png, .jpg or .pdf format, like **1a.jpg.** All files that constitute your answer to problems 1 and 2 are to be uploaded in one zip file.

Problems 3 and 4 are uploaded as separate zip files.

Note: The programs are not required to be executable. Insignificant bugs making the program fail to compile or execute, will be ignored by the censors.

Your Java code should be coded according to the object-oriented principles taught in IN1010. Do not use any classes from the Java library, apart from java.lang (which does not need importing), except where explicitly stated that such classes are allowed.

If you need a value not defined in the problem set, you may define a constant with a suitable value.

In this problem set, you will have to make many random decisions using the class Trekk ("draw") which contains a static method **int trekkInt(int min, int max)** which returns an integer x, where min <= x <= max. You may use this class, even if you not written it as your answer to problem 1d.

A forest of paths and walkers

In this problem set you are to write a program simulating how walkers use paths in a forest.

Problem 1. The forest, the paths and the crossings 30%

There are many *stier* ("paths") in a forest. In this problem, we classsify some as *kjerrevei* ("tractor road") and others as *natursti* ("nature trail"). All paths in the forest are either tractor roads or nature trails.
Some paths give a great *utsikt* ("view"), and this applies to both tractor roads and nature trails. You should model this property of having a great view as an interface.

Problem 1 deals mostly with building the forest with paths and crossings, while problem 2 is mostly about simulating (pretending to be) walkers in the forest.

Problem 1a 3%

Draw the class hierarchy for the various paths in the forest as described above.
End problem 1a

Skogen ("the forest") has many paths meeting at *stikryss* ("crossings"). You are to write the classes Skog, Kryss and all the classes and interfaces you described in problem 1a. There are few restrictions on how you program them, as long as you follow these specifications:

The class Kryss shall have a data structure containing all the paths that meet at the crossing. In this class, you may utilise any class from the Java API. When we later want to simulate people walking in the forest, we must decide which path a walker will use when he or she is at a crossing. Consequently, write a method in class Kryss returning a random path from that crossing. It is acceptable for a walker to return by the same path that he or she came from. Furthermore, Kryss needs a Boolean method which returns true if the crossing is isolated, i.e., that no paths lead to or from the crossing. (For example, it is not a good idea to start you trip at such a crossing.)

The class Sti needs a constructor which defines the length of the path (in meters) and the two crossings at either end of the path; these are parameters to the constructor. A path always references two crossings, one at either end. A path has no direction, and it is OK for a path to reference the same crossing at both ends; in this case, a walk along this path will only be a short roundtrip. Later, when we want to simulate people walking in the forest, we must determine where the walker ends up when starting out on a given path. You must write a method finnAndreEnde ("find other end") in the class Sti, which, given one end of the path as a parameter, returns the other end. Furthermore, the class Sti needs a method int beregnGaaTid(int v) ("compute walking time") which determines how long it takes to walk the path with the given speed v, and returns the result. The speed is given as meters per second, and the result should be given as whole minutes. It is your decision to either round the result or drop the digits after the decimal point.

A path with a good view contains a number utsiktsVerdi ("view value") between (and including) 1 and 6 stating the quality of the view. This number should also be a parameter to the class constructor. This constructor must check the parameter value and, it it is not in the range 1-6, an exception (a subclass of RuntimeException) must be thrown. You decide what to name the exception. Any path with a view must have a method to read the view value.

The class Skog must have a constructor which builds the data structure of paths and crossings. There are ANTSTIER paths and ANTKRYSS crossings, and these constants are set by the class constructor. Skog shall contain an array pointing to all crossings. Skog's constructor is to build the structure in the following way: First, create all the crossings (without any paths for the time being) and put these in the array of crossings. Then, create all the paths. For each path, select two random crossings between which the path is to go; also, select the type of path and the length.

Every created path should have an equal chance (25%) of being a tractor road, a nature trail, a tractor road with a view or a nature trail with a view. When creating a path with a view, the program must also select a quality value, and each of the six values should have an equal chance. The length of the path is drawn randomly in the range 220-2500 meters.

Note that this random system of paths is not very realistic, but we don't care about that in problems 1-3. For instance, it is quite all right to have crossings with no paths, or with just one.

The class Skog shall have a method hentTilfeldigKryss() ("get random crossing") which returns a crossing in the forest selected randomly, and another method hentTilfeldigStart() ("get random start") which guarantees that the returned crossing has at least one path and is thus suitable for starting a hike. You may assume that at least one such crossing always exists in the forest.

Problem 1b. Weight 3%

Draw a data structure with three objects of class Kryss and two objects of a subclass of class Sti. Do not include any methods.
End problem 1b

Problem 1c. 23%

Write the classes Skog, Kryss and all the classes and interfaces you described in problem 1a.
End problem 1c

Problem 1d. 1%

Write the class Trekk ("draw") with its static method int trekkInt(int min, int max) ("draw int") as described in the introduction. The method returns a random integer N where 0<=min<=N<=max. One way of drawing a random integer in the range 0 to N-1 (both numbers included) is to write (int)(Math.random()*N).
End problem 1d

Oppgave 2. Simulating walkers in the forest 45%

Based on the program parts you created in the previous problem, you shall now write a complete program simulating walkers in the forest. You shall not use any threads in problems 1 or 2.

In problem 2 you shall make the components of a general simulator and then simulate the walkers using this simulator. You are strongly encouraged to read all of problem 2 thoroughly, including how to simulate the walkers, before starting your implementation.

A simple general simulator
You are to implement a simple general simulator consisting of the classes Simulator, Aktivitet ("activity") and PrioKo ("priority queue"). There will be only one object of the class Simulator, and it shall have an instance variable int globaltid ("global time") starting at 0. The class Aktivitet models a general *activity*. All activities have a *handling* ("action") performed by an instance method void handling(). In addition, all activities have an instance variable int tid ("time") denoting when this action is to be performed next. Consequently, tid >= globaltid is an invariant for all activities.
In every step of the simulation, the action of the activity with the lowest value of tid is executed. To achieve this, the activities are stored in a *prioritetskø* ("priority queue") so that

the activity with lowest time is removed first. This priority queue is implemented as a class PrioKo. The classes Simulator, PrioKo and Aktivitet are designed to work together (they are tightly coupled) and the instance variables in the Aktivitet class should be accessible from other classes in the same package (classes in the same folder). Thus, you should not protect the instance variables of Aktivitet by specifying access as private (or protected).

The class Aktivitet shall have:

- two instance variables referencing, respectively, the objects with lower and higher value of the instance variable tid in the priority queue (and used as successor and predecessor pointers)
- an instance variabel tid (see above)
- an abstract method handling (see above).

Furthermore, the class shall implement the interface Comparable to be able to compare values of the instance variable tid in two Aktivitet objects.

Problem 2a Weight 4%

Write the class Aktivitet
End problem 2a

Priority queue for activities
Without using any classes in the Java library, make a class PrioKo which implements a double linked priority queue of Aktivitet objects. The class is to have two methods: settInn(a) ("enter") and hentUt() ("remove").

hentUt() removes and returns the first object (i.e., the one with lowest tid) in the queue. settInn(a) searches backwards (i.e., starts by comparing the new object with the one with largest tid) and enters the new object in the correct position in the priority queue.

Problem 2b Weight 10%

Write the class PrioKo
End problem 2b

Simulator
Now, you are to write the class Simulator. This simulator shall contain a priority queue. Simulator's constructor has as parameter an array with references to all the activities that are to be simulated; these are put in the priority queue. The activities may all have 0 as the value of tid, and it must be OK to have a priority queue with all activities having the same tid.

The next time anything will happen is determined by the instance variable tid in the activity with lowest tid (i.e., the first Aktivitet in the priority queue). This instance variable is updated in the method handling() in subclasses of Aktivitet.

The simulator shall contain a method void simuler(int t) ("simulate") to simulate the activities; t is how many minutes the simulation shall last. Consequently, the method simuler(int t) should contain a while-loop like this: while (globaltid < t) { . . . }

The body of the while-loop shall do this:

- Remove the activity with the lowest tid from the priority queue.
- Assign globaltid to this lowest tid.
- Call the method handling() in the activity (As this method is abstract in Aktivitet, it must be defined by a subclass like Turgaaer, see below). When terminating the action, we assume that this method increases the local time to the point of time in the future when this activity is to be executed by the the simulator again.
- Enter this activity in the priority queue again (with the new time).

Problem 2c weight 10%

Write the complete Simulator class
End problem 2c

Simulate walkers
You are now to employ this simulator to simulate walkers. The class Turgaaer ("walker") is consequently a subclass of Aktivitet.

A walker has a speed (in meters/minute) and a place (a reference to a Kryss) at which the walker is positioned or headed for. The speed and the crossing where the walker starts are parameters to Turgaaer's constructor. All walkers start at tid equals 0.

When a walker leaves a crossing, the program shall set the activity time to when the walker arrives at the next crossing (see below). Consequently, the method handling() in the Turgaaer class is called by the simulator when a walker reaches a crossing. The method must then choose where the walker should proceed next by drawing a random path from the present crossing (and it may be the path from which he or she came). The walker's next destination is the other end of that path. Then, the walker calls the method beregnGaaTid ("compute walking time") for the selected path and with the walker's speed as parameter. Based on this walking time, the walker determines when he or she will arrive at the other end (the old value of tid (which is 0 when the walker first starts out) + the time it takes to reach the other end of the path). The last thing the method does is to assign the new arrival time at the other end of the path to the instance variable **tid**.

Problem 2d weight 9%

Write the class Turgaaer.
End problem 2d

The main program in class TestSimulator
Write the class TestSimulator with a main() method so that everything works. This method shall create the forest, create all the walkers and create the simulator with all the walkers. Define a constant specifying how many walkers should be created. For each walker, draw a random start crossing at which the walker should start. The walker's speed is also drawn randomly; it should be in the range 20-200. Finally, call the method simuler(t) in the simulator to do the actual simulation; the parameter is an integer drawn randomly from the range 30-480.

Problem 2e weight 10%

Write the class TestSimulator with a main() method as described above.

End problem 2e

Problem 2f weight 2%

In the class Turgaaer, an unlucky programmer may, by mistake, try to specify a new time less than the current time. This will ruin the simulation. How should the Aktivitet class be coded to be more robust and prevent the time from going backwards or standing still?
Deliver your answer as a new Java file named Aktivitet2.java. Please add comments to explain the changes you have made. Aktivitet2 is not supposed to work with the other files in your other classes in problems 1 and 2.
End problem 2f

Delivery of problems 1 and 2: All program files and files with drawings should be collected in a zip file named Oppgave1_2.zip