Data Structures and Algorithms: Final Online Assessment

Due: 31st May 2021 from 18:00 to 1st June 18:00

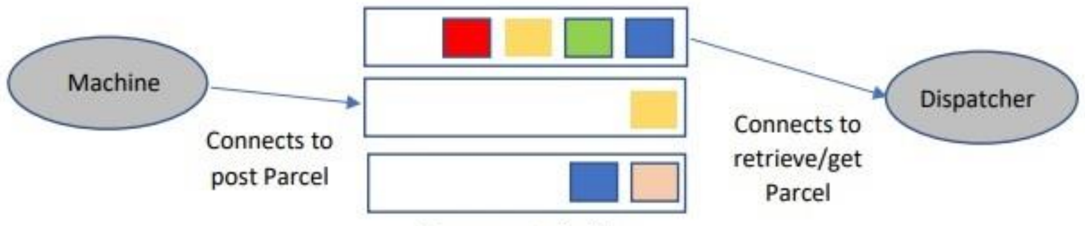
25 Marks Possible

Instructions:

Before attempting this question, you should have sat the multichoice questionnaire worth 75 marks on Blackboard. Create a new project called "Practical Assessment". Download the file from the Practical Assessment folder on Blackboard and build the solution into your Netbeans project (or other IDE alternative). When submitting, make sure you zip up the entire src folder with the following structure; lastname_firstname_studentID.zip (using your own name and ID) and submit to the Practical Assessment folder on Blackboard BEFORE the due date and time (18:00 pm (1st June)). Late submissions will not be accepted. Anti-plagiarism software will be used on all assignment submissions, so make sure you submit YOUR OWN work and DO NOT SHARE your answer with others. Any breaches will be reported and will result in an automatic fail of the course.

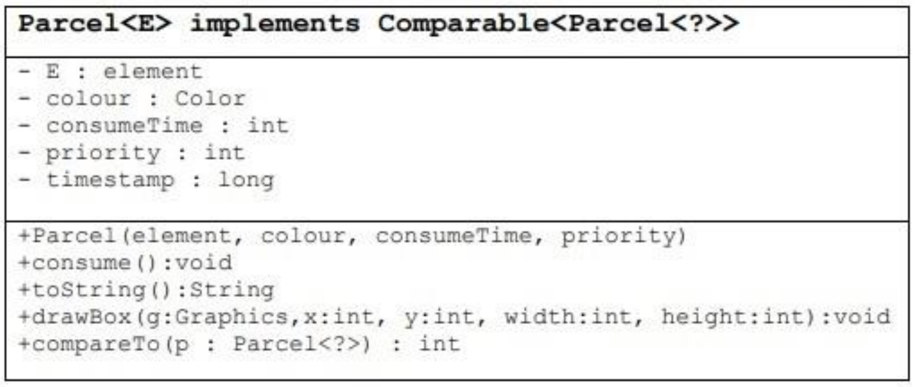
Question – Factory Simulator (25 marks); Recommended Readings –Chapters 1.3, 1.4, 3.2, 4.4

The purpose of this question is to create a factory simulator – a multi-threaded program which uses the producer-consumer design pattern. A *Machine* can run as a thread and connects to a *ConveyorBelt*, once connected it creates *Parcel* obects to post to the belt. The *ConveyorBelt* encapsulates a priority queue which keeps parcels in order of their *Comparable*. A *Dispatcher* also running as a thread connects to an available *ConveyorBelt*, once connected it can retrieve *Parcel* objects. Both the *Machine* and *Dispatcher* can monitor several conveyor belts, but should only connect to one at a time. Each *ConveyorBelt* can only hold a limited amount of *Parcel* objects.



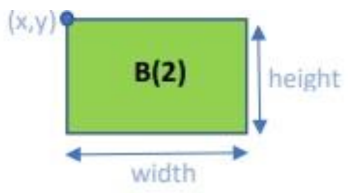
3 ConveyorBelt objects

Part A: Using the UML below create a class that represents a *Parcel* used in this factory simulator.

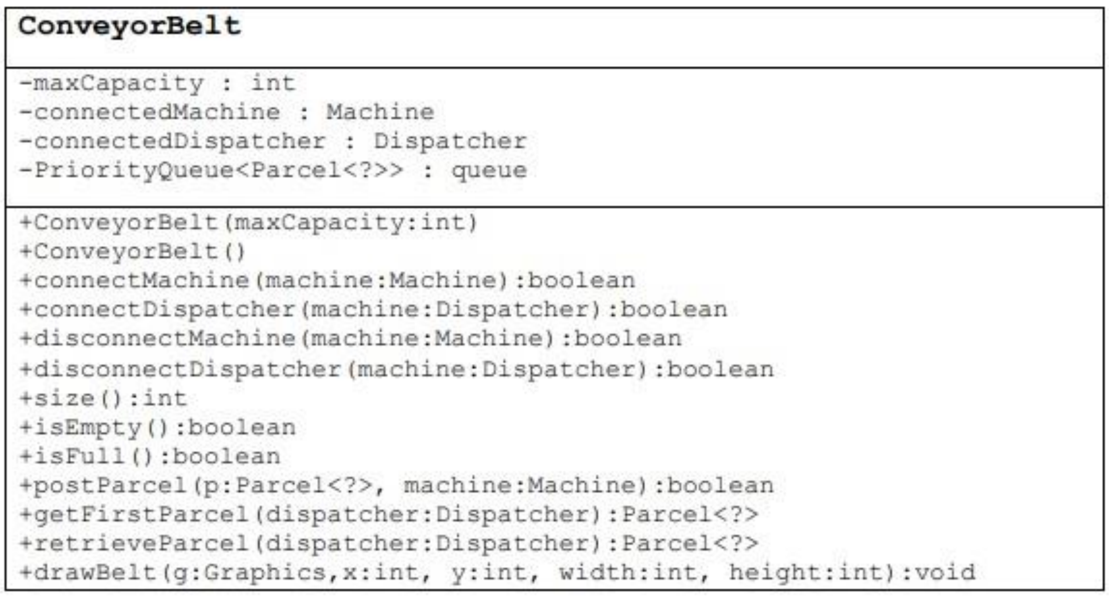


```
Parcel<E> implements Comparable<Parcel<?>>
- E : element
- colour : Color
- consumeTime : int
- priority : int
- timestamp : long
+Parcel(element, colour, consumeTime, priority)
+consume():void
+toString():String
+drawBox(g:Graphics,x:int, y:int, width:int, height:int):void
+compareTo(p : Parcel<?>) : int
```

Each parcel is instantiated with a generic element, a priority, colour and a *consumeTime* – the time it takes to process this parcel. During creation of the *Parcel*, a timestamp is set (in nano second time) to indicate its creation time. A *Parcel* is *Comparable* so can be compared with another *Parcel* even though they might hold a different generic. The ordering is determined firstly by priority and secondly via creation time (timestamp). The class also has a toString which prints out the element and priority in parenthesis, and a *consume* function, which sleeps the calling code for the duration specified by the parcels consumptionTime. A *Parcel* can also be drawn with a *Graphics* object and method parameters as shown below. The specified x,y pixel coordinate represents the top left corner of the drawn box and *width* and *height* parameters determines its dimension.

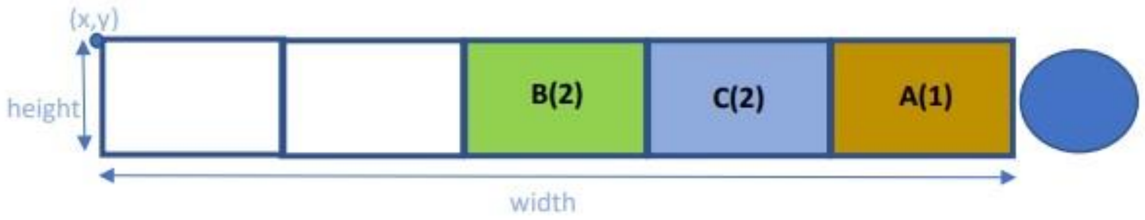


Part B: Using the following UML, create a class called *ConveyorBelt* which encapsulates a *PriorityQueue* specifically designed to hold *Parcel* objects.



```
ConveyorBelt
-----------------------------------------------------------
-maxCapacity : int
-connectedMachine : Machine
-connectedDispatcher : Dispatcher
-PriorityQueue<Parcel<?>> : queue
-----------------------------------------------------------
+ConveyorBelt(maxCapacity:int)
+ConveyorBelt()
+connectMachine(machine:Machine):boolean
+connectDispatcher(machine:Dispatcher):boolean
+disconnectMachine(machine:Machine):boolean
+disconnectDispatcher(machine:Dispatcher):boolean
+size():int
+isEmpty():boolean
+isFull():boolean
+postParcel(p:Parcel<?>, machine:Machine):boolean
+getFirstParcel(dispatcher:Dispatcher):Parcel<?>
+retrieveParcel(dispatcher:Dispatcher):Parcel<?>
+drawBelt(g:Graphics,x:int, y:int, width:int, height:int):void
```

The conveyor belt holds a *maximum capacity* of parcels in which it holds. It cannot exceed this capacity which is given to the constructor. A default capacity of 10 is used if no max capacity is specified during construction. The *ConveryorBelt* only allows one *Machine* and one *Dispatcher* to connect to it at a time (using a null if there is none currently connected). Only the currently connected *Machine* can post parcels to the conveyor belt (if not full) and only the currently connected *Dispatcher* can get and retrieve parcels from the conveyor belt (if any). A connected machine or dispatcher should pass themselves in as reference when posting/retrieving parcels so that the *ConveyorBelt* can check if they are currently connected or not. A *ConveyorBelt* can also be drawn with a *Graphics* object and the given method parameters. The specified x,y pixel coordinate represents the top left corner of the belt and width and height parameters determines its full dimension. The entire belt is drawn with or without parcels. If the belt has a connected dispatcher or machine, it should draw them as circles on each side of the belt. The example belt below has a capacity of 5, but only has 3 parcels in it with a connected dispatcher but no connected machine.



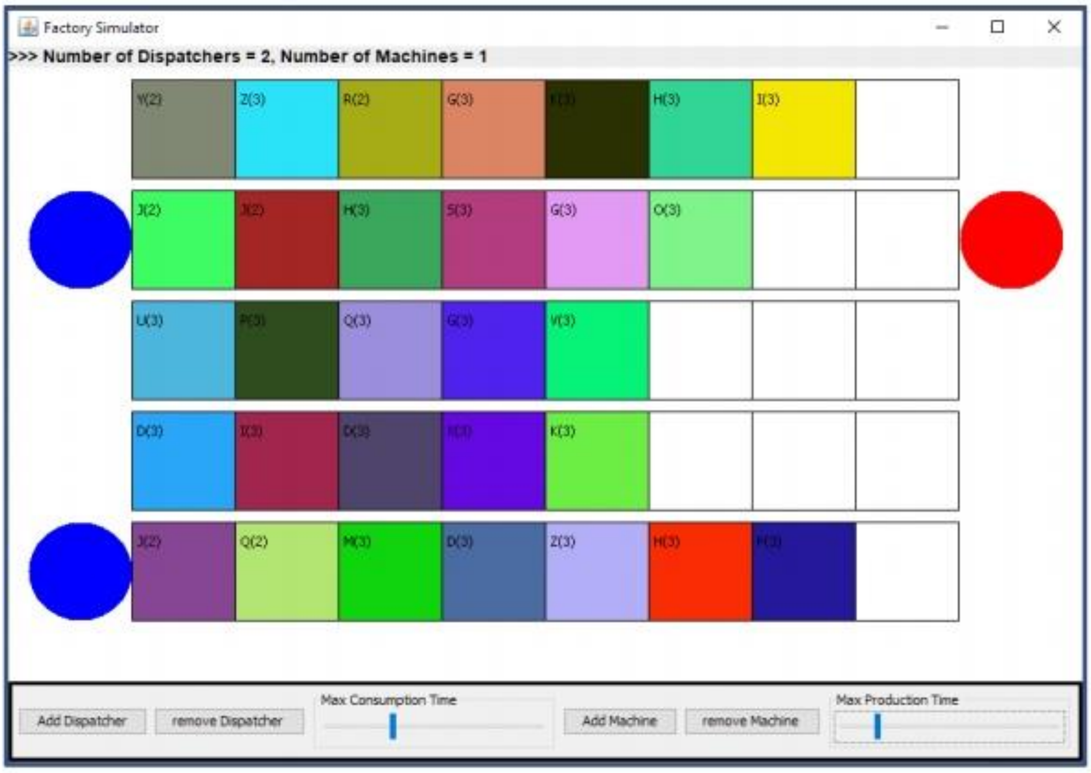
Note: As a *ConveyorBelt* is shared between many threads, it should be made "thread safe" so no race conditions arise in the simulator.

Part C: Create a class called *Machine* that runs continuously as a thread until requested to stop. When a machine is instantiated an array of *ConveyorBelt* objects should be passed into its constructor. It also holds the following public static variables: MIN_CONSUMPTION_TIME, MAX_CONSUMPTION_TIME, MIN_PRODUCTION_TIME and MAX_PRODUCTION_TIME. Set these initially to something suitable.

While a *Machine* is running, it should constantly circulate the *ConveyorBelt* objects that it is monitoring. If it comes across an available belt that is not full, it should try connecting to it. If the belt is full or it is unable to connect (as perhaps another machine is already connected), it simply moves on to the next *ConveyorBelt* in the array and repeats. If the machine successfully connects, it then starts posting *Parcel* objects to it. Before creating each parcel, the *Machine* must sleep for a random duration between min and max production time variables. A *Machine* should create a *Parcel* with a random colour, a random priority (between 1-3), a random letter (between A-Z) for its element and a random *consumeTime* (between min and max consumption time variables) – all passed into the *Parcel* constructor. Once the *Machine* detects that the *ConveyorBelt* becomes full, it should disconnect from the belt and then repeat the entire process by trying to connect to the next *ConveyorBelt* in the array.

Create another class called *Dispatcher* which also runs continuously as a thread until requested to stop. It has a similar behaviour to *Machine*, monitoring an array of *ConveyorBelt* objects passed into the constructor. However, its behaviour is slightly different. Once again, a *Dispatcher* will try to successfully connect to a *ConveyorBelt* if it is not empty and does not already have an existing *Dispatcher* connected to it. If it connects successfully it gets a *Parcel* from the *ConveyorBelt* and calls the consume method on it – effectively sleeping the thread for the parcels specified *consumeTime*. Once consumed it retrieves/removes the parcel from the belt and disconnects – repeating the process by moving to the next available *ConveyorBelt* in the array. Note, the *Dispatcher* only removes a single *Parcel* from the belt per connection, whereas the *Machine* posts multiple parcels until the belt is full per connection.

Part D: Finally, make a GUI called FactorySimulatorGUI which demonstrates an animated visual of the system. It should create FIVE *ConveyorBelt* objects of capacity eight. It should also maintain a List of *Machine* objects and a separate List of *Dispatcher* objects. It should have JButtons to add and remove *Dispatcher* or *Machine* threads. It should also have a JLabel which specifies the amount of current machine and dispatcher threads actively running. It also should have two JSlider's which can adjust the MAX_CONSUMPTION_TIME or MAX_PRODUCTION_TIME static values effectively increasing the range of random duration that either the *Machine* takes to produce a *Parcel* or a *Dispatcher* takes to consume the *Parcel*. An example screenshot of a working FactorySimulatorGUI is shown below with two connected *Dispatcher* objects (blue) and one connected *Machine* object (red) monitoring the same five *ConveyorBelt* objects, each containing randomly coloured parcels with assigned priority numbers.



>>> Number of Dispatchers = 2, Number of Machines = 1