Section A

Problem Description

Your job is to write a program which will allocate seats on an aeroplane.

The aeroplane has 50 rows, numbered 1-50. Each row has six seats, labelled 'A'-'F'. Row 1 is reserved for crew. Rows 2-15 are reserved for business class passengers. The remaining rows are for economy class passengers. Emergency exits are located at rows 1, 10 and 30. We consider the aeroplane seats to be ordered as follows: 1A, 1B, …, 1F, 2A, 2B, …, 2F, *etc*. The *first* and *last* seats are 1A and 50F respectively.

In the instructions below, the specific ranges 1-50 and 'A'-'F' are used for rows and seat letters. However, your program should be designed so that minimal code modifications would be required to change the number of rows and seats per row, which rows are reserved for which types of passengers, and which rows are emergency exit rows. You can assume that each reserved section will always be a contiguous sequence of rows.

Files Provided

You will find a **SectionA** folder in your Lexis home directory with a **src** subdirectory. Inside **src** are three sample data files, **`small.txt`**, **`medium.txt`** and **`large.txt`**, and an **aeroplane** package containing the following Java files:

- **Seat.java**, a skeleton class which you will fill out to represent an aeroplane seat
- **Passenger.java**, a skeleton class which you will fill out and subclass to represent different types of aeroplane passenger
- **SeatAllocator.java**, an incomplete class which you will complete to represent and perform the allocation of passengers to seats
- **Luxury.java**, an enumeration representing various luxuries to which business class passengers are entitled
- **AeroplaneFullException.java**, a checked exception used to handle the aeroplane becoming full
- **MalformedDataException.java**, a checked exception used to handle badly formed input data
- **Main.java**, the entry point for the seat allocation program

During the tasks below, you will be asked to add methods and fields to some of these classes, and to create some further classes. You may feel free to add additional fields, methods and classes as you see fit. To enable auto-testing, you should not change the signatures of any of the provided methods.

Your Task

1) Fill out the class **Seat**, to meet the following requirements:
   - A seat should be represented by a *row* – an integer in the range 1-50, and a *letter* – a character in the range 'A'-'F'.
   - A seat should be constructed from a row and letter.

- A seat should be represented as a string in the obvious way, e.g., a seat with row 5 and letter E should be represented by the string “5E”.
- **Seat** should provide an instance method **`boolean isEmergencyExit()`** which returns **true** if and only if the seat is located in an emergency exit row.
- **Seat** should provide an instance method, **`boolean hasNext()`** which returns **true** for all but the *last* seat in the aeroplane, according to the ordering of seats given above.
- **Seat** should provide an instance method, **`Seat next()`**. When invoked on a **Seat** object, **`next()`** should return a **Seat** object corresponding to the next seat in the ordering of seats given above. If there is no next seat, a **java.util.NoSuchElementException** should be thrown. **Hint:** if you add an **`int`** to a **`char`** the result has type **`int`**; you may find that you need to cast this back to **`char`**.

(10 marks)

2) Write a hierarchy of classes to represent the three types of passenger: crew members, business class passengers and economy class passengers.
   - The abstract class **Passenger** (which you are given) should be at the top of this hierarchy; all other passenger classes should be derived (directly or indirectly) from this class.
   - Every passenger should have two string fields, one for their first name and one for their surname.
   - All non-crew passengers should have a non-negative integer field representing their age.
   - All passengers should have an instance method, **`bool isAdult()`** which returns *true* if and only if the passenger is at least 18 years of age. All crew are required to be adults, so when invoked on a crew member passenger, this method should simply return *true*.
   - Business class passengers should have an associated *luxury*. This can be either champagne, truffles or strawberries, and should be represented using the **Luxury** enumeration with which you are provided.
   - A passenger should be represented by a string consisting of the passenger’s type (crew, business class or economy class), followed by all fields associated with the passenger. The precise format of this string is up to you.
   - Where possible you should avoid duplicating state or functionality that is common to multiple classes.

(10 marks)

3) Returning to the **Seat** class:
   - Two **Seat** objects are regarded as equal if they have identical fields. Implement this notion of equality between seats by overriding appropriate methods of **Object** in **Seat**.

(5 marks)

4) Look at the skeleton class **SeatAllocator**, which represents an allocation of seats to passengers. **SeatAllocator** has a single field, **allocation**, which maps seats to passengers. If a seat is not mapped to any passenger the seat is free. The **`toString()`** method of **SeatAllocator** returns the string associated with this map. Some relevant details of the **java.util.Map** interface are provided at the end of Section A.

SeatAllocator provides an instance method, **`void allocate(String filename)`**, whose job is to read passenger details from the specified input file and allocate seats for these passengers. The functionality for seat allocation is missing: it is your job to complete this functionality.

- A stub instance method:

```
private void allocateInRange(Passenger passenger,
                             Seat first, Seat last)
                    throws AeroplaneFullException
```

  is provided. Implement this method so that it allocates the given passenger to the first suitable seat in the range **`first-last`** (inclusive). A seat is suitable for a passenger if a) the seat is free, and b) if the seat is in an emergency row then the passenger is an adult. If there is no suitable seat, the method should throw an **AeroplaneFullException**.

- Look at the incomplete methods **`allocateCrew`**, **`allocateBusiness`** and **`allocateEconomy`**. These methods read passenger fields from a buffered reader. Complete each of these methods by a) using the fields which have been read to construct an appropriate passenger object, and b) calling **`allocateInRange`** with this passenger object and an appropriate range of seats.

At this point, you should be able to run the **`main`** method in class **Main**. You are provided with three example input files, **`small.txt`**, **`medium.txt`** and **`large.txt`** which you can use to test your program. The **`large.txt`** input file contains lots of dummy passenger entries with name "XXX YYY", and should cause the aeroplane to become full. Inspect these files and check (for a few cases, at least) that the allocation your program produces looks correct. Feel free to write further test inputs if you wish.

(15 marks)

5) Add functionality to **SeatAllocator** for upgrading passengers

- Add to **SeatAllocator** an instance method **void upgrade()**. This method should consider the economy class passengers in increasing order of seats. For each passenger, an attempt should be made to move the passenger to a business class seat, if one is free. The passenger's type should not change.
- Extend the **`main`** method in class **Main** so that, after printing the initial allocation, upgrading is performed and the new allocation is printed.

(6 marks)

6) In class **Main**, write a static method **`countAdults`** which takes a set of passengers and returns the number of adults in the set. It should be possible to call the method with any set of passenger objects, not just with a set of the form **java.util.Set<Passenger>**.

(4 marks)

Total for Section A: 50 marks

Summary of important methods of the **Map<K, V>** interface

An object with type **Map<K, V>** maps keys of type **K** to values of type **V**. A map cannot contain duplicate keys; each key can map to at most one value.

`boolean containsKey(Object key)`	Returns *true* if this map contains a mapping for the specified key.
`boolean containsValue(Object value)`	Returns *true* if this map maps one or more keys to the specified value.
`V get(Object key)`	Returns the value to which the specified key is mapped, or *null* if this map contains no mapping for the key.
`boolean isEmpty()`	Returns *true* if this map contains no key-value mappings.
`Set<K> keyset()`	Returns a **Set** view of the keys contained in this map.
`V put(K key, V value)`	Associates the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced by the specified value. Returns the previous value associated with key, or *null* if there was no mapping for key.
`V remove(Object key)`	Removes the mapping for a key from this map if it is present. Returns the value to which this map previously associated the key, or *null* if the map contained no mapping for the key.
`int size()`	Returns the number of key-value mappings in this map.
`Collection<V> values()`	Returns a **Collection** view of the values contained in this map.

Section B

Problem Description

In many real-life manifacturing and construction projects, a *Critical-Path Analysis* tool is used to schedule the tasks of a project so that the project can be completed in the shortest time. A project is assumed to be broken into *tasks*. Each task has a name and a duration, and tasks may depend on other tasks. Consider, for instance, the project represented diagrammatically in Figure 1. S*tart* indicates the beginning of the project, *Finish* indicates the end of the project, the nodes are the tasks and the directed links are the dependencies between tasks. *TaskA* has a duration of 3 time units, *TaskB* has a duration of 2 time units, etc. *TaskA* and *TaskB* can happen in parallel but they must be both completed before *TaskD* is performed. So the earliest time that *TaskD* can start is 3 time units. (Note that the representation in Figure 1 is only for illustration purposes.)

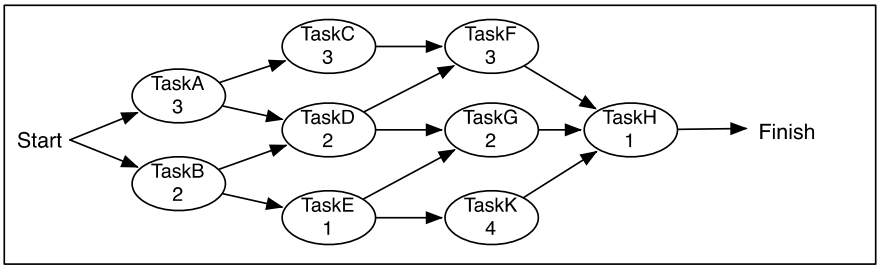


Figure 1: Tasks diagram

In this test we consider the problem of computing the *Earliest Completion Time* (`ec`) for a project, i.e. the minimum amount of time units needed to complete all the tasks of a project. For instance, the earliest completion time for the above project is 10 time units.

We want to write an implementation **Scheduler** that performs the calculation of the earliest completion time for a project using a different type of graph, called ***event-node graph***, instead of the tasks diagram given above.

In an event-node graph (see Figure 2), edges correspond to tasks and nodes correspond to the beginning and ending events of a task. For instance, *e2* and *e4* are the beginning and ending events of *TaskC*. Where a task depends on more than one task (e.g. *TaskD* depends on *TaskA* and *TaskB*), a dummy node (e.g. *e6d* representing the beginning of *TaskD*) is introduced together with dummy edges that join this node with the ending event of the precedent tasks (e.g. *e2* representing the ending event of *TaskA*, and *e3* representing the ending event of *TaskB*). Dummy edges (e.g. the edge from *e2* to *e6d*) have dummy tasks (i.e. no name and zero duration).

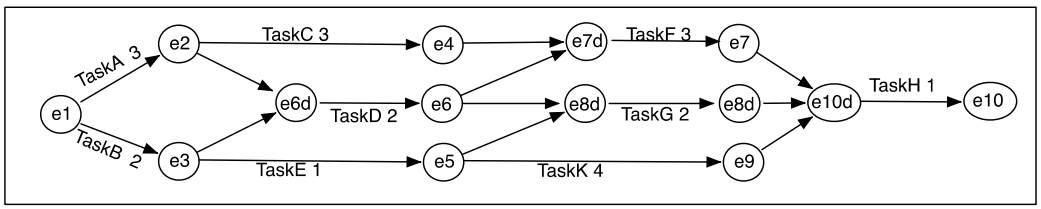


Figure 2: Event-node graph

To compute the earliest completion time for the whole project we need to compute the earliest completion time for each node in the event-node graph. The initial node (e.g. *e1*) of the event-node graph always has completion time equal to zero. The earliest completion time for a node, different from the initial node, is given by the sum of the earliest completion time for its precedent node and the duration of the connecting edge. For instance, the earliest completion time for *e2* is 0+3=3 and the earliest completion time for *e3* is 0+2=2. In case of nodes with more than one precedent node (e.g. dummy

node *e6d*), the biggest of these sums is used as earliest completion time (e.g. for *e6d* the earliest completion time is max(3+0, 2+0) = 3. The earliest completion time for the last node gives the earliest completion time for the whole project.

The **Scheduler** program works as follows.

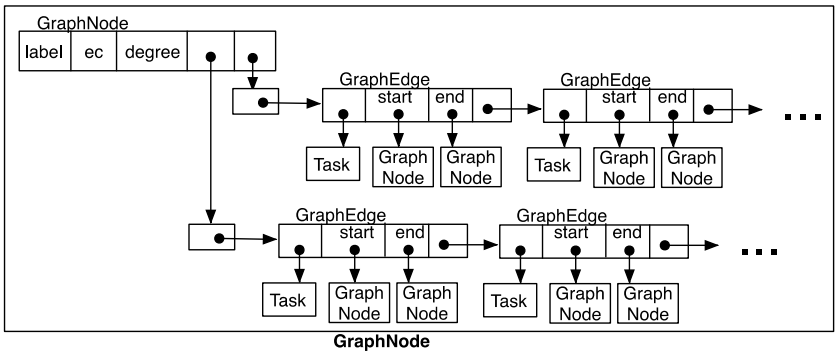
i) It takes as input a text file containing the project information (see Appendix 1 for the format description of the input file) and constructs an event-node graph that stores the project (see the private method **`parseGraph`** of the **Graph** class).

ii) It finds an ordering in which the earliest completion time for each nodes has to be computed. This is given by the output of a ***topological sorting*** algorithm over the graph (i.e. the auxiliary method **`topologicalSort`** of the **Graph** class). This algorithm is described below.

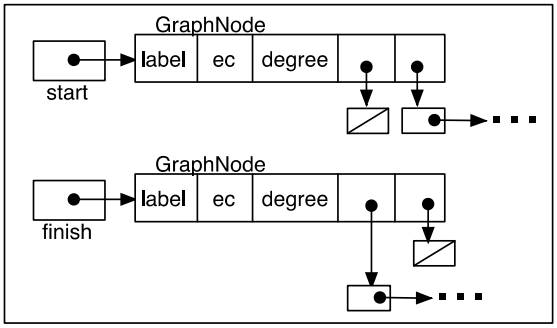
iii) It calculates the earliest completion time for each node according to the description given above and the ordering found in step (ii). This step is given by the auxiliary method **`computeEarliestCompletionTime`** of the **Graph** class.

Data Structure.

Our program implements an *event-node graph* using a **Graph** Abstract Data Type, which in turns makes use of **GraphNode** and **GraphEdge** (see diagrams below). A **GraphNode** represents an event node. It includes a **`label`**, the **`ec`** (earliest completion time) for that event from the start of the project, the **`degree`** of the node (used by the topological sorting algorithm), a list of incoming **GraphEdge** (empty in the case of the start event node) and a list of outgoing **GraphEdge** (empty in the case of the finish event node). A **GraphEdge** stores the **Task** information, and its start and end **GraphNode**.



GraphNode



Abstract Data Type **Graph**

The following methods are included in the **Graph** ADT:

- **`setIncomingDegree`** of each node in the **Graph**. This is an auxiliary method that traverses the graph starting from the start node and recursively sets the **`degree`** of each node to be the number of **`incomingEdges`** of that node. This method is used by the **`topologicalSort`** method described below.

- **`topologicalSort`** of the nodes in the **Graph**. This is an auxiliary method used by the **`computeEarliestCompletionTime`** method described below. It uses two queues, a temporary queue and a result queue. It first initialises the degree attribute of each node using the **`setIncomingDegree`** method. Then, adds the **`start`** node (which always has 0 degree) to the temporary queue, which only stores nodes with zero degree. Then the following steps are repeated until the temporary queue is empty:

- A node is dequeued from the temporary queue and added to the result queue.
- The degree of each of its children nodes (i.e. the nodes connected through its outgoing edges) is decremented by 1.
- Any child node whose degree becomes zero is added to the temporary queue.

Once the temporary queue becomes empty the result queue is returned.

- **`computeEarliestCompletionTime`** for the sorted nodes. This is an auxiliary method that computes the earliest competion time for each node in the queue returned by the **`topologicalSort`** method according to the description given after Figure 2.

Your task is to complete the provided implementation. In addition to this specification, you should carefully read the commented source files.

You are given (all provided files are under the **SectionB** folder in your Lexis home directory):

- For your convenience, a UML class-diagram describing the architecture of the system. Your implementation will build upon the given design.
- The text file *project.txt* that is the input to the **Scheduler**. It includes the description of a given project.
- The class **Scheduler** that creates an event-node graph and returns the earliest completion time of the graph.
- The **GraphInterface** that specifies the main access procedure for the **Graph** ADT.
- The partial implementation of the class **Graph** that realizes the interface **GraphInterface**. This includes the auxiliary methods **`setIncomingDegree`**, **`setIncomingDegreeRecursively, topologicalSort`**, **`computeEarliestCompletionTime`**, described above, and the **`parseGraph`** for constructing an event-node graph from a given input text file.
- The classes **GraphNode** and **GraphEdge** that support the implementation of the **Graph** ADT.
- The class **Task** to store the tasks of the project.
- The implementation of an ADT **GenericList<T>.**
- The partial implementation of the class **Queue<T>** that realizes the given interface **QueueInterface<T>.**
- A class **Node** that supports the implementation of **GenericList<T>** and **Queue<T>.**

You are asked to write the following methods:

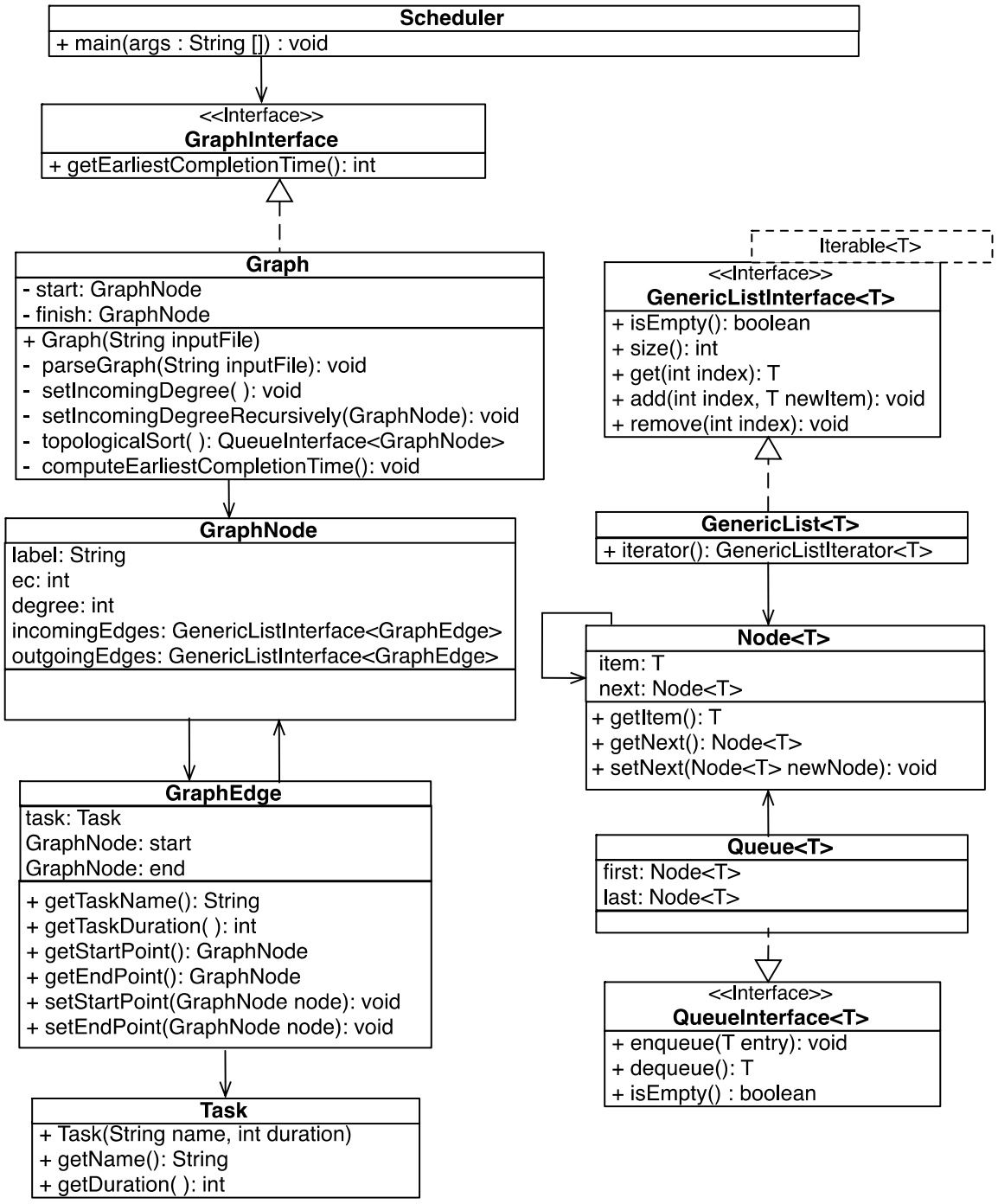
1) The methods **`enqueue(T item)`** and **`dequeue()`** for the class **Queue<T>**. **(10 marks)**.
2) The auxiliary method **`setIncomingDegreeRecursively(GraphNode node)`** in the class **Graph**, that recursively sets the degree of each node, starting from the given node, to be the number of its incoming edges.
   (15 marks)

3) The auxiliary method `topologicalSort()` in the class **Graph**, that returns a queue of the nodes in the graph constructed according to the algorithm described above.
(20 marks)

4) Complete the implementation of the method `computeEarliestCompletionTime()` in the class **Graph**, that computes the earliest completion time for each node in the graph.
(5 marks)

Total for Section B: 50 marks

System Architecture



Appendix 1: format of the input text file describing a given project.

```
9               // total number of tasks in the given project
A 3             // task name and task duration
B 2             //  task name and task duration
C 3             //  task name and task duration
D 2             //  task name and task duration
E 1             //  task name and task duration
F 3             //  task name and task duration
G 2             //  task name and task duration
K 4             //  task name and task duration
H 1             //  task name and task duration
10              //  total number of dependencies between tasks
Finish H        //  task name and precedent dependent task
H F G K         //  task name and precedent dependent tasks
F C D           //  task name and precedent dependent tasks
G D E           //  task name and precedent dependent tasks
K E             //  task name and precedent dependent task
C A             //  task name and precedent dependent task
D A B           //  task name and precedent dependent tasks
E B             //  task name and precedent dependent task
A Start         //  task name and precedent dependent task
B Start         //  task name and precedent dependent task
```