Object Oriented Programming
École Centrale de Lille, G1/G2 – 2022-2023

Sequence 2 – Part 2 – Scheduling

Diego Cattaruzza, Pierre Hosteins, Maxime Ogier,
Frédéric Semet, Pablo Torrealba

Context and objectives

In this subject, we continue our work to design a software to improve the production planning of the company Masks&Protection.

As a reminder, the French company Masks&Protection, based in the north of France, is currently facing a strong demand for the production of single-use masks. This year, the demand is really exceptional and the production managers are not sure that there is enough time to produce all the masks. In addition, the contracts stipulated with the customers require that the company meets a given deadline for the delivery of the masks. If the deliveries are performed after the deadlines, the company ends up paying heavy penalties (proportional to the delay). It is therefore essential to respect as closely as possible the deadlines imposed by customers.

Masks&Protection decided to hire on the best students of the Hauts-de-France region to improve its production planning. The task is to design softwares able to provide a production plan capable to manage large quantities of demands.

In the previous subject, we have developed an object model for the production workshop. The objective of this subject is to design algorithms to determine good quality solutions for the scheduling of the production tasks.

This subject will illustrate the following concepts:

- the notion of interface;
- the notion of static method;
- the management of collections of objects in Java;
- the interface **List** and its implementations (**ArrayList** and **LinkedList**);
- the use of sorting methods.

Good practices to adopt

- Test your code as you go along.
- Always respect the Java naming conventions.
- Make indentations in the right way.
- Give understandable and relevant names to your variables, attributes, classes, methods.
- Respect as much as possible the encapsulation principle.
- Use appropriate data structures.
- Make the methods short with few indentation levels.
- Do not duplicate pieces of code, but favor reusable code.
- Use the *refactoring* tools of your IDE (right-click then *Refactor* on Netbeans).
- Use Javadoc to understand how the methods you call work.
- When necessary, comment your own code according to the Javadoc format (comment starting with "`/**`" above the attribute and method definitions).

1 Optimisation of the production scheduling: minimising total completion time

This is the time to surprise Mister Safety and show him the abilities of the students of Centrale Lille!

At the end of the previous subject, you must have noticed that Mister Safety does not take into account the characteristics of the tasks to determine the production schedule. However, it seems quite logical to consider the duration of the tasks and their deadline to determine the schedule. This would hopefully reduce the total execution time and penalties to be paid.

To do this, you will proceed in two steps. In this section, we are interested in minimising the total completion time (which allows Mister Safety to complete tasks more quickly, and therefore pay less overtime). In the second part, as a bonus, you can show Mister Safety that you can also minimise penalty costs!

1.1 Implementation of the interface and scheduling classes

Question 1. As a first step, in order to compare the performance of the different scheduling methods, create a new package **scheduling** where you include the objects required to produce the schedules in the workshop. In this new package, create an interface **Scheduling**. This interface defines only method **schedule(int numberMachines, List<Tasks> tasks)** which takes as parameters the number of machines in the workshop and a list of tasks. This method returns an object of type **Workshop** in which the tasks have been scheduled.

The interface **Scheduling** specifies the way a method schedules the given tasks on the given number of machines. Then, we define a few classes, each implementing a method to schedule tasks. We notice that each method schedules the same list of tasks. Hence, we need to create a copy of this list in each scheduling method and schedule the copies of the tasks.

Question 2. Add a static method **public static List<Task> copyTasks(List<Task> tasks)** to the interface **Scheduling**. This method makes a copy of the list of tasks in parameter. To ensure that your code works correctly, the copy must be a deep copy (each task is also copied). Thus, a copy constructor in the **Task** class can be very useful. You can test your method with the following code:

```
List<Task> tasks = new ArrayList<>();
tasks.add(new Task(100));
tasks.add(new Task(150));
List<Task> copyTasks = Scheduling.copyTasks(tasks);
copyTasks.add(new Task(400));
System.out.println("Initial number of tasks: " + tasks.size());
// Has to be 2
System.out.println("Final number of tasks: " + copyTasks.size());
// Has to be 3
```

Question 3. Add a class **SchedulingMisterSafety** which implements the interface **Scheduling**. This class creates the workshop, and then schedules the tasks in it according to the modus operandi proposed by Mister Safety.

Question 4. In the class **Workshop**, add a method **public void display(boolean verbose)** which displays the criteria (completion time, penalty cost) of the workshop. If the parameter **verbose** is set to **true**, then the workshop is also displayed (this can be useful for debugging). Schedule some tasks and test that everything works well.

In general, if you want to test the performance of an algorithm, you need to build a test set with different instances (in this case, they are characterised by a number of machines and a task list). On Moodle, file *SchedulingTester.java* contains a class **SchedulingTester** with 6 test sets. These test sets are characterised by the number of machines and the number of tasks given in the Table 1.

Question 5. File *SchedulingTester.java* contains class **SchedulingTester** which has to be added in the **scheduling** package. In this class, there is a static method **compareSchedulingAlgorithms** which is not implemented yet. Implement this method. In the method **main** at the end of the file, there is only the scheduling method of Mister Safety (for the moment). Do a test with the method **main** which is provided in class **SchedulingTester** to check that everything works correctly.

Test set	1	2	3	4	5	6
nb. of machines	2	3	4	4	4	3
nb. of tasks	6	40	40	60	20	1000

Table 1: Number of machines to be considered for the test sets.

1.2 Ordering of objects

Now, we want to develop new scheduling algorithms to minimise the total completion time. To do so, a first simple idea is to sort the tasks by increasing processing time before applying again the modus operandi explained by Mister Safety.

To define the order in which you wish to sort the tasks in the list, class **Task** must implement interface **Comparable<Task>**. Thus, the header of your **Task** class must be:

```
public class Task implements Comparable<Task> {
    ...
}
```

You will notice that the following error appears:

> Task is not abstract and does not override abstract method compareTo(Task) in Comparable

You have two options.

1. Declare class **Task** as abstract (with the keyword **abstract**). In this case, you would no longer be able to instantiate (= create objects of) the class **Task**. This class could be used as a parent class and one (or more) child classes would inherit from it (using the keyword **extends**, remember the Driver, Technician, Truck, Formula1 classes) and implement the abstract methods.

2. Implement the abstract methods defined in the interface implemented by **Task**.

In the particular case of this subject, it seems reasonable to choose the second option. Interface **Comparable<E>** declares a single method: **public int compareTo(E o)**. This method compares the current object (**this**) with the object (**o**) given as parameter and returns an integer whose exact value is of no importance, but which must be:

- negative if "**this** < **o**";
- zero if "**this** = **o**";
- positive if "**this** > **o**".

An order is then defined for the manipulated objects. Below, you can find an example of an order (by decreasing deadline):

```
1  @Override
2  public int compareTo(Tache t) {
3     if(t.deadline > this.deadline) { // 'this' is after 't'
4        return 1;
5     } else if(t.deadline < this.deadline) { // 'this' is before 't'
6        return -1;
7     } else { // 'this' is equal to 't'
8        return 0;
9     }
10 }
```

As the returned value does not matter and only its sign counts, this method can be written in a simpler manner as follows:

```
1  @Override
2  public int compareTo(Tache t) {
3     return (t.deadline - this.deadline);
4  }
```

Moreover, each object has a method **public boolean equals(Object o)** which returns **true** if object **o** is equal to object **this**. This method is implemented in the **Object** class from which all objects are derived, and defines by default an equivalence relation as discriminating as possible: two objects are considered equal if they have the same reference. Fortunately, it is possible to redefine method **equals** in objects we create, so that the notion of equality is consistent with the notion of "physical" equality between two objects.

Similarly, the class **Object** has a method **public int hashCode()** which returns an integer called *hash code* associated with the object. This method is used when creating hash tables, such as those used to implement the **HashSet** and **HashMap** classes.

It is strongly recommended that the natural ordering defined by the **compareTo** method is compatible with equality, namely *(x.compareTo(y)==0) == (x.equals(y))*. It is also strongly recommended that equality is consistent with the hash codes, i.e. if *x.equals(y)*, then *x.hashCode() == y.hashCode()*.

Question 6. Modify class **Task** so that it implements interface **Comparable<Task>** and its method **compareTo(Task o)** in order to have the tasks ordered by increasing processing time. Redefine methods **equals** and **hashCode** to ensure consistency on the equality of two objects. To do so, you can use the Netbeans wizards (right-click then *Insert Code ...* then *equals and hashCode...*). Be careful that method **compareTo** returns **0** if and only if method **equals** returns **true**. Test in the main method of class **Task** that method **compareTo(Task o)** works correctly.

1.3 Sorting the elements of a collection

The **Collections** class has a static method **sort(List<T> list)** which sorts the elements of **list** (which must implement the interface **List**). The elements of **list** are reorganised in order to respect the natural ordering of the elements of the list (the elements must therefore implement the interface **Comparable**). The efficiency of the sorting is $\mathcal{O}(nlogn)$ where n is the number of elements in the list (feel free to have a look at the Javadoc for more details).

To sort a list according to the reverse order of the natural ordering, first, we can sort the list according to the natural ordering, then, we call static method **reverse (List<T> list)** of class **Collections**. This method reverses the order of the elements in the list (in linear time). We should not tinker: changing the definition of method **compareTo(Task t)** of class **Task** is not a good idea at all! We are then obliged to modify this method each time we want to sort according to a new criterion.

Furthermore, static method **shuffle(List<T> list)** of class **Collections** shuffles the elements of the list. This can be useful during your tests, for example.

Question 7. Define two new classes that implement the **Scheduling** interface:

- **SchedulingIncreasingTime**, which sorts the tasks by increasing processing time before making the assignment to the machines according to the modus operandi of Mister Safety;
- **SchedulingDecreasingTime** which sorts the tasks by decreasing procesing time before making the assignment to the machines according to the modus operandi of Mister Safety.

Modify the tests of the **main** method of class **SchedulingTester**, so that we can compare the three scheduling algorithms in terms of total completion time. Run the tests on the 6 test sets. What about the results you found? Have a

look at the results of test set 1: do you think it is possible to improve them? If yes, propose a better solution, if no, give an argument.

1.4 A bit of randomness

Question 8. Now it's your turn! Suggest other scheduling algorithms that may improve the previous results. In particular, we could think of introducing a bit of randomness...

2 Bonus - Minimisation of the total penalty cost

Sorting the tasks to minimise the total completion time of the schedule allowed you to obtain a schedule that is efficient enough to surprise Mister Safety. Now that you know how to construct such schedules, Mister Safety points out that for the same schedule it is possible to minimise the penalty costs without increasing the total completion time. To do so, it is sufficient to swap the order of execution of the tasks on a machine.

Question 9. Modify method **compareSchedulingAlgorithms()** of class **SchedulingTester** in the following manner. First, display the values of the total completion time and the total penalty cost, after having scheduled the tasks. Then, try to improve the schedule in terms of total penalty cost without modifying the total completion time. Of course, you are strongly encouraged to add new methods to the classes in the **workshop** package, if necessary. If your results are amazing, run to Mister Safety, maybe he will hire you!

References

[1] Delannoy, C., *Programmer en Java, Eyrolles, 2014*