Programmation Orientée Objet
IG2I, LE2 – 2018-2019

CTP – 31 Janvier 2019 – Partie 2

Diego Cattaruzza, Philippe Kubiak, Maxime Ogier

Conditions du déroulement de la partie 2

- **durée** : 3h20 ;
- ordinateurs personnels autorisés ;
- documents et pages internet autorisés ;
- accès à Moodle autorisé ;
- les moyens de communication (mails, messageries instantanées, Facebook, Tweeter, téléphones portables, Google Drive, etc.) sont INTERDITS ; le fait qu'un moyen de communication ne soit pas listé ci-dessus ne vous donne pas le droit de l'utiliser ;
- si vous utilisez du code que vous n'avez pas développé par vous-même lors du CTP, vous devez citer vos sources (dans les commentaires) ; ne pas citer ses sources est une fraude.

Rendu

Vous déposez sur Moodle, dans la zone de dépôt **Dépôt du CTP 2018/2019** une archive compressée de votre répertoire de projet. Cette archive doit être nommée **CTP_Nom_Prenom.zip** avec votre nom et votre prénom.

Un nommage différent donnera lieu à deux points de pénalisation. Cette règle est également valable pour toute proposition de nommage de ce CTP : projet, paquetages, classes, méthodes, attributs, ...

Vérifiez bien que les fichiers sources que vous avez réalisés lors du CTP sont présents dans le dossier que vous déposez sur Moodle (oui, vérifiez bien cela : toutes les années il y a des répertoires vides ou contenant les mauvais fichiers !) Il ne sera pas possible de renvoyer une nouvelle version de votre projet après la date limite de dépôt.

Notation

Cette seconde partie de l'épreuve de CTP est évaluée sur 16 points. Les deux dernières sections du sujet sont en bonus.

Pour chaque question, la moitié des points est attribuée si le code fonctionne correctement. **L'autre moitié des points est attribuée en fonction de la qualité du code** (respect des principes de la programmation orientée objet, efficacité des algorithmes) et de la présentation du code (indentation correcte, respects des conventions Java dans les noms de classe, attributs, méthodes et variables, commentaires au format Javadoc quand c'est pertinent). **Il ne faut donc pas négliger la qualité et la présentation de votre code**.

Si le travail rendu lors du CTP n'est pas d'assez bonne qualité (il ne permet pas de valider que vous avez acquis les bases de la programmation orientée objet), alors la note attribuée sera automatiquement de 0.

Consignes

Le sujet est composé de 5 grandes parties liées entre elles. À la fin, deux parties supplémentaires sont en bonus. Vous êtes très fortement encouragés à traiter les parties dans l'ordre.

Vous êtes bien sûr autorisés à rajouter les attributs ou méthodes que vous jugez nécessaires même si ce n'est pas dit explicitement dans le sujet.

Attestation sur l'honneur

Je soussigné(e) ..... certifie que le projet déposé sur Moodle est issu d'un travail purement personnel. Je certifie également avoir cité toutes les sources de code externe dans les commentaires de mon code.

Fait à .....

Le .....

Signature .....

Sujet

Description du contexte

La compagnie Air2I est une importante compagnie aérienne, qui effectue une centaine de vols par jour. Elle fait face à de nombreux problèmes de planification, et en particulier, ce qui nous intéresse ici est la planification du personnel navigant commercial (hôtesses et stewards) sur les vols. Dans la suite de ce sujet, le personnel navigant commercial sera désigné par l'acronyme PNC.

Un vol représente la liaison d'un aéroport de départ vers un aéroport d'arrivée, avec des dates de départ et d'arrivée. En fonction du temps de trajet, le vol nécessite qu'un certain nombre de PNC soient à son bord.

Chaque PNC peut effectuer plusieurs vols successifs, mais il faut tout de même respecter un temps minimum entre deux vols pour faire le changement au sein de l'aéroport. Ce temps peut être plus long s'il faut changer d'aéroport.

Étant donné un ensemble de vols à effectuer sur les prochains jour et un ensemble de PNC, le problème auquel nous nous intéressons est d'affecter les PNC sur les vols de façon à maximiser le nombre de vols qui ont leur équipage en PNC complet. En effet, si des vols n'ont pas leur équipage en PNC complet, alors Air2I doit faire appel à un prestataire extérieur, ce qui lui coûté très cher.

Face à la complexité du problème à traiter, Air2I compte sur l'élite des élèves informaticiens afin de proposer une application capable de réaliser cette planification. Notez bien que la direction du service informatique de Air2I souhaite que le code produit soit dans le langage Java, et que le code soit de très bonne qualité afin de pouvoir être intégré au sein de son système d'informations.

1 Mise en place du modèle objet (3 points)

Dans un premier temps, nous nous occupons simplement de mettre en place le modèle objet de base avec les aéroports, les vols et les PNC.

1.1 Aéroport

Un aéroport est caractérisé par un code qui contient en général 3 caractères, par un nom et par un entier qui représente sa zone. Par exemple, l'aéroport de Lille se nomme "Lille Lesquin", son code est "LIL", et sa zone est 1. Par ailleurs, notez bien que le code d'un aéroport est unique : deux aéroports différents ne peuvent pas avoir le même code.

Question 1. Créez un nouveau projet nommé **CTP_Nom_Prenom** avec votre nom et votre prénom. (Rappel : un nommage non conforme aux règles ⇒ 2 points de pénalisation). Ajoutez un paquetage **modele**. Dans ce paquetage, ajoutez une classe **Aeroport**, avec ses attributs, un constructeur par données, et les accesseurs et mutateurs nécessaires. Redéfinissez les méthodes **equals**, **hashCode** et **toString**. Faites un test pour vérifier que tout fonctionne correctement.

1.2 PNC

Pour le moment, nous allons considérer qu'un PNC est simplement défini par son nom et son prénom. Deux PNC différents ne peuvent pas avoir le même nom et le même prénom.

Question 2. Dans le paquetage **modele**, ajoutez une classe **PNC**, avec ses attributs, un constructeur par données, et les accesseurs et mutateurs nécessaires. Redéfinissez les méthodes **equals**, **hashCode** et **toString**. Faites un test pour vérifier que tout fonctionne correctement.

1.3 Vol

Un vol est caractérisé par son aéroport de départ, son aéroport d'arrivée, sa date de départ et sa date d'arrivée. Si deux vols ont le même départ, la même arrivée, et les mêmes dates, alors ils sont identiques. Par ailleurs, notez que, dans un vol, les aéroports de départ et d'arrivée ne peuvent pas être identiques ; et la date d'arrivée ne peut pas être antérieure à la date de départ.

Notez, qu'en java, une date se modélise avec la classe **Date** du paquetage **java.util**. Quelques méthodes utiles de la classe **Date** sont : **getTime**, **after**, **before**, **compareTo**. N'hésitez pas à aller voir dans la documentation ce que font ces méthodes.

Question 3. Dans le paquetage **modele**, ajoutez une classe **Vol**, avec ses attributs, un constructeur par données, et les accesseurs et mutateurs nécessaires. Si les données en paramètre du constructeur ne sont pas cohérentes, alors un message texte devra être affiché sur la console. Néanmoins l'objet vol sera tout de même créé. Il faut juste qu'un message d'avertissement soir affiché. Redéfinissez les méthodes **equals**, **hashCode** et **toString**. Faites un test pour vérifier que tout fonctionne correctement.

Dans votre test, pour créer des objets de type **Date** facilement, vous pouvez vous inspirer du code suivant. Notez que la méthode **parse** peut lever une exception de type **ParseException**.

```
1  DateFormat format = new SimpleDateFormat("dd/MM/yyyy HH:mm",
       Locale.FRANCE);
2  Date d1 = format.parse("31/01/2019 11:45");
3  Date d2 = format.parse("31/01/2019 12:30");
```

2 Différents types de PNC (3,5 points)

Chez Air2I, tous les PNC ne sont pas logés à la même enseigne. Les PNC recrutés avant 2010 fonctionnent sur un système de grade qui dépend de leur

nombre d'heures de vol. Il faut donc leur comptabiliser toutes leurs heures de vol. Normalement, un PNC peut être amené à effectuer n'importe quels vols. Mais récemment les syndicats ont obtenus que certains PNC soient affectés à un sous ensemble d'aéroports et ne puissent donc être affectés que sur des vols qui relient ces aéroports.

2.1 PNC gradés

Les PNC gradés possèdent un grade, et un nombre d'heures de vol (nombre entier). Voici la correspondance entre les grades et le nombre d'heures de vol :

- `stagiaire`, si moins de 2 000 heures ;
- `échelon 1`, entre 2 000 et 10 000 heures ;
- `échelon 2`, entre 10 000 et 30 000 heures ;
- `exceptionnel`, entre 30 000 et 50 000 heures ;
- `hors classe`, au delà de 50 000 heures.

Question 4. Ajoutez dans le paquetage **modele** une énumération **Grade** qui contient les différents grades possibles. Pour chaque grade, on associera le nombre d'heures de vol maximum pour ce grade (2 000 pour `stagiaire`, 10 000 pour `échelon 1`, etc.).

Question 5. Dans le paquetage **modele**, ajoutez une classe **PNCGrade** qui représente un PNC gradé (donc un PNC avec un grade et un nombre d'heures de vol). Ajoutez un constructeur par données avec en paramètres le nom, le prénom et le nombre d'heures de vol (sans le grade). Le grade doit être déterminé en fonction du nombre d'heures de vol. Ajoutez les accesseurs et mutateurs nécessaires. Redéfinissez la méthode **toString**. Faites un test pour vérifier que tout fonctionne correctement.

Pour déterminer le grade en fonction du nombre d'heures de vol, une solution simple (même si peu élégante) est d'utiliser une structure conditionnelle (**if**, **else**) sur le nombre d'heures de vol.

2.2 PNC affectés à des aéroports

Pour les PNC qui sont affectés à certains aéroports, il faut mémoriser tous les aéroports auxquels ils sont affectés.

Voici les spécifications demandées pour cette classe :

- un constructeur **public PNCAffectAeroports(String prenom, String nom)** ;
- une méthode **public void ajouterAeroport(Aeroport a)** qui ajoute l'aéroport **a** aux aéroports autorisés ;
- une méthode **public boolean aeroportAutorise(String code)** qui renvoie **true** si l'aéroport avec le code **code** est autorisé, et **false** sinon.

Question 6. Dans le paquetage **modele**, ajoutez une classe **PNCAffectAeroports** qui respecte les spécifications présentées ci-dessus.

Faites bien attention au choix de votre structure de données pour les aéroports autorisés. Pour chaque aéroport, le code est unique, et la méthode **aeroportAutorise** sera souvent appelée ...

Vous pouvez ensuite tester cette classe avec un code qui peut ressembler au code suivant.

```
1  PNCAffectAeroports hot1 = new PNCAffectAeroports("bb", "bb");
2  PNCAffectAeroports hot2 = new PNCAffectAeroports("cc", "cc");
3  PNCAffectAeroports hot3 = new PNCAffectAeroports("dd", "dd");
4  Aeroport paris = new Aeroport("CDG", "Paris Charles de Gaulle", 1);
5  Aeroport lille = new Aeroport("LIL", "Lille Lesquin", 1);
6  hot1.ajouterAeroport(paris);
7  hot2.ajouterAeroport(lille);
8  hot3.ajouterAeroport(paris);
9  hot3.ajouterAeroport(lille);
10 hot3.ajouterAeroport(null);
11 System.out.println(hot1.aeroportAutorise("CDG")); // true
12 System.out.println(hot2.aeroportAutorise("CDG")); // false
13 System.out.println(hot3.aeroportAutorise("???")); // false
```

2.3 Compatibilité entre un vol et un PNC

Ètant donné un vol v, un PNC est toujours compatible avec ce vol. Il en est de même pour un PNC gradé. En revanche, pour un PNC affecté à des aéroports, il n'est compatible avec le vol v que si l'aéroport de départ et l'aéroport d'arrivée font tous les deux partie de ses aéroports autorisés.

Question 7. Modifier votre code de telle sorte que l'on puisse savoir si un PNC est compatible ou non avec un vol. Les modifications peuvent concerner les classes **PNC**, **PNCGrade** et **PNCAffectAeroports**. Testez le bon fonctionnement de votre code.

3 Équipage d'un vol (2,5 points)

L'application que nous développons devra pouvoir donner l'équipage présent sur chaque vol : l'ensemble des PNC affectés à ce vol. Afin de toujours offrir la meilleure expérience à bord de l'appareil pour ses passagers, Air2I définit un

nombre minimum de PNC sur chaque vol, en fonction de la durée du vol. Si on note d la durée en heures d'un vol, voici les règles utilisées :

- si le vol est court ($d < 2$), il faut au moins 2 PNC ;
- si le vol est moyen ($2 \leq d < 4$), il faut au moins 4 PNC ;
- si le vol est long ($d \geq 4$), il faut au moins 7 PNC.

En conséquence, on souhaite modifier la classe **Vol** de telle sorte que :

- il y ait un attribut pour mémoriser l'équipage affecté au vol ;
- il y ait une méthode qui permette d'ajouter un PNC à l'équipage du vol ;
- il y ait une méthode qui renvoie un booléen indiquant si l'équipage affecté au vol est complet ou non.

Question 8. Modifiez la classe **Vol** afin de prendre en compte les modifications présentées ci-dessus concernant l'équipage du vol. Testez que tout fonctionne correctement.

Si on veut comparer la durée entre deux dates, le plus simple est de convertir ces dates en millisecondes (plus exactement le nombre de millisecondes écoulées depuis le 1er janvier 1970). Ensuite, on peut ajouter ou retirer des millisecondes à cette date. Voici un petit exemple ci-dessous.

```
Date d = new Date(); // maintenant
long time = d.getTime(); // temps ecoulé en millisecondes
long heureEnMs = 60*60*1000; // nombre de millisecondes dans une
    heure
long timeDansUneHeure = time + heureEnMs;
```

4 Ensemble de vols (5 points)

À présent, nous souhaitons pouvoir savoir si un PNC peut effectuer un vol ou non. Dans un premier temps, il faudra donc que chaque PNC soit associé avec l'ensemble de ses vols. Dans un second temps, il faudra pouvoir déterminer si un PNC peut effectuer ou non un vol supplémentaire (qui ne doit donc pas être en conflit avec les vols auxquels il est déjà affecté).

4.1 Ensemble trié

Afin d'être en conformité avec le système d'informations actuel de chez Air2I, le directeur du service informatique impose que les vols réalisés par chaque PNC soient triés :

- par date de départ croissante ;
- en cas d'égalité sur la date de départ, par date d'arrivée croissante ;
- en cas d'égalité sur les deux critères précédents, par code d'aéroport de départ croissant ;
- en cas d'égalité sur les trois critères précédents, par code d'aéroport d'arrivée croissant.

En Java, la classe **TreeSet** qui implémente l'interface **Set** permet de représenter des ensembles triés. Cependant, si l'on souhaite faire un **TreeSet** avec des objets de type **E**, il faut définir le critère de tri des objets de type **E**. Une manière simple de réaliser cela est que la classe **E** implémente l'interface **Comparable<E>**, et que l'on spécifie dans la méthode **compareTo(E obj)** quel est le critère de tri.

Les questions suivantes vont détailler comment réaliser un **TreeSet** de vols dans la classe **PNC**. Si vous n'y arrivez pas, ou si vous y passez trop de temps, vous pouvez faire une liste de vols à la place, et passer à la suite du sujet.

Question 9. Faites en sorte que la classe **Aeroport** implémente l'interface **Comparable<Aeroport>**, et implémentez la méthode **compareTo** de telle sorte que les aéroports soient triés par ordre croissant de leur code. Vous pouvez vous inspirer du code suivant pour tester :

```
Aeroport paris = new Aeroport("CDG", "Paris Charles de Gaulle", 1);
Aeroport lille = new Aeroport("LIL", "Lille Lesquin", 1);
Aeroport marseille = new Aeroport("MRS", "Marseille Provence", 2);
Aeroport lille2 = new Aeroport("LIL", "Lille Lesquin 2", 1);
System.out.println(lille.compareTo(paris)); // doit etre positif
System.out.println(lille.compareTo(marseille)); // doit etre negatif
System.out.println(lille.compareTo(lille2)); // doit etre nul
```

Question 10. Faites en sorte que la classe **Vol** implémente l'interface **Comparable<Vol>**, et implémentez la méthode **compareTo** de telle sorte que les vols soient triés selon l'ordre décrit au début de la Section 4.1. Testez pour vous assurer que tout fonctionne correctement.

N'oubliez pas que la plupart des objets disposent d'une méthode **compareTo**.

Question 11. Ajoutez dans la classe **PNC** un attribut de type **TreeSet<Vol>** que vous initialiserez dans le constructeur. Nous nous occuperons d'ajouter les vols plus tard.

4.2 Compatibilité entre vols

Pour que deux vols soient compatibles, il faut que les PNC aient le temps de faire le changement entre ces deux vols. Le temps pour faire le changement n'est pas le même selon que l'on soit dans le même aéroport ou non. Ainsi, si un vol v_1 arrive à la date t_1^A dans un aéroport, et qu'un vol v_2 part à la date t_2^D du même aéroport, alors, il faut une durée minimum de 2 heures entre t_2^D et t_1^A. En revanche, si un vol v_1 arrive à la date t_1^A dans un aéroport, et qu'un vol v_2 part à la date t_2^D d'un autre aéroport, alors, il faut une durée minimum de 12 heures entre t_2^D et t_1^A. On suppose que pendant ces 12 heures, un PNC peut utiliser un autre moyen de transport pour changer d'aéroport.

Question 12. Dans la classe **Vol**, ajoutez une méthode **public boolean estCompatible(Vol v)** qui renvoie **true** si le vol **v** est compatible, et **false** sinon. Vous pouvez vous inspirer du code suivant pour réaliser vos tests.

```
1  DateFormat format = new SimpleDateFormat("dd/MM/yyyy HH:mm",
       Locale.FRANCE);
2  Aeroport paris = new Aeroport("CDG", "Paris Charles de Gaulle", 1);
3  Aeroport lille = new Aeroport("LIL", "Lille Lesquin", 1);
4  Vol v1 = new Vol(paris, lille,
5        format.parse("31/01/2019 11:45"),
6        format.parse("31/01/2019 12:30"));
7  Vol v2 = new Vol(paris, lille,
8        format.parse("31/01/2019 17:45"),
9        format.parse("31/01/2019 18:30"));
10 Vol v3 = new Vol(lille, paris,
11       format.parse("31/01/2019 12:45"),
12       format.parse("31/01/2019 13:30"));
13 Vol v4 = new Vol(paris, lille,
14       format.parse("02/02/2019 11:45"),
15       format.parse("02/02/2019 12:30"));
16 Vol v5 = new Vol(lille, paris,
17       format.parse("31/01/2019 15:45"),
18       format.parse("31/01/2019 16:30"));
19 System.out.println("Compatibilité");
20 System.out.println(v1.estCompatible(v2)); // false
21 System.out.println(v1.estCompatible(v3)); // false
22 System.out.println(v1.estCompatible(v4)); // true
23 System.out.println(v1.estCompatible(v5)); // true
24 System.out.println(v2.estCompatible(v1)); // false
25 System.out.println(v3.estCompatible(v1)); // false
26 System.out.println(v4.estCompatible(v1)); // true
27 System.out.println(v5.estCompatible(v1)); // true
```

4.3 Possibilité d'effectuer un vol

Étant donné un vol v, un PNC peut effectuer ce vol :

- s'il est compatible avec ce vol (voir la Section 2.3) ;
- s'il est compatible avec tous les vols qu'il effectue déjà (ceux qui sont dans son ensemble de vols).

Question 13. Dans la classe **PNC**, ajoutez une méthode qui permet de savoir si le PNC peut effectuer un vol passé en paramètre de la méthode.

5 Affecter des vols à un PNC (2 points)

Une grande partie du code est à présent en place. Il nous reste cependant à pouvoir affecter des vols aux PNC. Pour que l'affectation soit réalisée, il faut que le PNC puisse effectuer le vol (voir Section 4.3). Par ailleurs, il faudra que le PNC fasse partie de l'équipage du vol.

Question 14. Dans la classe **PNC**, ajouter une méthode pour affecter un vol à un PNC. Cette méthode doit renvoyer un booléen pour indiquer si l'affectation a bien eu lieu ou non. Faites bien attention que l'ajout ne se fasse que si le PNC peut effectuer ce vol, et que si l'ajout est réalisé, alors le PNC est bien dans l'équipage du vol. Testez pour vérifier que tout fonctionne correctement.

6 Bonus : planification des vols (2 points bonus)

À présent nous souhaitons planifier l'affectation des PNC sur les prochains vols opérés par Air2I. Ceci doit être réalisé dans une classe **Planning**, qui doit permettre de réaliser cette affectation. Plus précisément, le responsable du service informatique souhaite que cette classe possède :

- un constructeur par défaut ;
- une méthode **public void ajouterVol(Vol v)** qui permet d'ajouter un vol opéré par Air2I ;
- une méthode **public void ajouterPersonnel(PNC pers)** qui permet d'ajouter un PNC pour effectuer les vols ;

- une méthode **public void affecterPersonnel()** qui fait l'affectation de tous les PNC aux vols, en essayant de maximiser le nombre de vols avec un équipage complet ;
- une méthode **public String infosVolsComplets()** qui renvoie une chaîne de caractères contenant le nombre de vols complets et le pourcentage de vols complets.

Question 15. Ajoutez un paquetage **planning**. Dans ce paquetage, ajoutez une classe **Planning**, avec les constructeurs et méthodes listés précédemment. Déterminez les attributs nécessaires. Vous pouvez tester que votre code fonctionne correctement en vous inspirant du code suivant.

```
1  DateFormat format = new SimpleDateFormat("dd/MM/yyyy HH:mm",
       Locale.FRANCE);
2  Aeroport paris = new Aeroport("CDG", "Paris Charles de Gaulle", 1);
3  Aeroport lille = new Aeroport("LIL", "Lille Lesquin", 1);
4  Vol v1 = new Vol(paris, lille, format.parse("31/01/2019 11:45"),
       format.parse("31/01/2019 12:30"));
5  Vol v2 = new Vol(paris, lille, format.parse("31/01/2019 17:45"),
       format.parse("31/01/2019 18:30"));
6  Vol v3 = new Vol(lille, paris, format.parse("31/01/2019 12:45"),
       format.parse("31/01/2019 13:30"));
7  Vol v4 = new Vol(paris, lille, format.parse("02/02/2019 11:45"),
       format.parse("02/02/2019 12:30"));
8  Vol v5 = new Vol(lille, paris, format.parse("31/01/2019 15:45"),
       format.parse("31/01/2019 16:30"));
9  PNCAffectAeroports hot1 = new PNCAffectAeroports("bb", "bb");
10 PNCAffectAeroports hot2 = new PNCAffectAeroports("cc", "cc");
11 PNCAffectAeroports hot3 = new PNCAffectAeroports("dd", "dd");
12 hot1.ajouterAeroport(paris);
13 hot1.ajouterAeroport(lille);
14 hot2.ajouterAeroport(paris);
15 hot2.ajouterAeroport(lille);
16 hot3.ajouterAeroport(paris);
17 PNCGrade pil1 = new PNCGrade("bb", "bb", 8270);
18 PNCGrade pil2 = new PNCGrade("cc", "cc", 15646);
19 PNCGrade pil3 = new PNCGrade("dd", "dd", 36543);
20 Planning plan = new Planning();
21 plan.ajouterVol(v1);
22 plan.ajouterVol(v2);
23 plan.ajouterVol(v3);
24 plan.ajouterVol(v4);
25 plan.ajouterVol(v5);
26 plan.ajouterPersonnel(pil1);
27 plan.ajouterPersonnel(pil2);
28 plan.ajouterPersonnel(pil3);
29 plan.ajouterPersonnel(hot1);
30 plan.ajouterPersonnel(hot2);
31 plan.ajouterPersonnel(hot3);
32 plan.affecterPersonnel();
33 System.out.println(plan.infosVolsComplets());
```

Pour la méthode **affecterPersonnel**, commencez par faire des choses simples. S'il vous reste du temps, vous pouvez tester des idées plus complexes à mettre en œuvre.

Question 16. Pour faire un test de plus grande ampleur, utilisez le fichier `TestPlanning.java` présent sur Moodle. Cette classe doit être mise dans le paquetage **planning**. Si vous avez bien fait votre affectation, vous devriez atteindre un taux de vols complets de l'ordre de 94%.

7 Bonus : montée en grade (2 points bonus)

Dans cette section bonus, nous revenons sur la montée en grade des personnels gradés, afin de proposer une manière plus élégante de déterminer leur grade en fonction de leur nombre d'heures de vol.

Question 17. Dans l'énumération **Grade**, ajoutez une méthode **public Grade getSuivant()** qui renvoie le grade suivant le grade actuel de l'énumération (évidemment, pour le dernier élément de l'énumération, on renvoie ce même élément).

Question 18. Dans la classe **PNCGrade**, ajoutez une méthode **miseAJourGrade** qui doit mettre à jour le grade du PNC en fonction de son nombre d'heures de vol. On fera en sorte que cette méthode ne teste pas tous les cas possibles, mais fasse appel à la méthode **getSuivant** de l'énumération **Grade**.

Question 19. Dans la classe **PNCGrade**, ajoutez une méthode **effectuerVol(Vol v)** qui, en fonction du vol **v** en paramètre, doit mettre à jour le nombre d'heures de vol du PNC, ainsi que son grade si nécessaire. Pour le nombre d'heures d'un vol, nous ferons un arrondi à l'entier supérieur.