UNIVERSITÉ CAEN NORMANDIE

UFR des Sciences

Université de Caen Normandie
Département mathématiques-informatique
2018–2019, 2e session
26 juin 2019
Bruno Zanuttini

L2 informatique, L2 mathématiques

**Examen**

Unité M.MIM3A1 : Introduction à la programmation orientée objet

2 h — Tous documents autorisés

---

*Chaque candidat doit, au début de l'épreuve, porter son nom dans le coin de la copie qu'il cachera par collage après avoir été pointé. Il devra en outre porter son numéro de place sur chacune des copies, intercalaires, ou pièces annexées.*

# 1 Notions fondamentales (10 points)

**Classes et interfaces** Pour la question 1, on considère les définitions d'interfaces et de classes suivantes (l'élision `[...]` signale qu'il manque une partie non pertinente du code) :

```
public interface Verifiable {
    public boolean verify();
}

public interface CryptographicSystem {
    public String getSystem();
}

public abstract class IntegritySystem implements Verifiable {
    protected String message;
    protected String key;
    public IntegritySystem(String message, String key) {[...]}
    [...]
}

public class RSASignedMessage
extends IntegritySystem implements CryptographicSystem {
    protected String author;
    [...]
    @Override
    public String getSystem() {
      return "RSA";
    }
}

public class BankAccountNumber extends IntegritySystem {
    [...]
}

public class SolvedEquation implements Verifiable {
    protected double [] coefficients;
    protected double value;
    [...]
}
```

Il n'est pas important de comprendre ce que représentent ces classes pour répondre aux questions, mais celles-ci seront d'autant plus concrètes avec les informations suivantes :

- RSA est un système cryptographique, permettant notamment d'ajouter à un message une signature électronique, qui peut être vérifiée si l'on connaît l'auteur du message [1],
- les numéros de comptes bancaires comportent une « clef de contrôle » (les deux derniers chiffres) qui permettent de vérifier qu'il n'y a pas eu d'erreur de saisie dans le numéro lui-même (les autres chiffres, qui jouent ici le rôle de « message »),
- l'« intégrité » d'un message est le fait qu'il ne comporte pas d'erreur ni de modification, ce que permettent de vérifier, chacun pour ce qui les concerne, une signature électronique ou une clef de contrôle,
- la classe `SolvedEquation` représente une équation polynomiale à une inconnue $x$, par la liste de ses coefficients (constante, coefficient de $x$, coefficient de $x^2$, etc.), et une solution potentielle (valeur pour $x$) pour cette équation.

**Question 1 (5 points)** *Pour chacune des affirmations suivantes, indiquer si elle est correcte ou non, et justifier brièvement :*

1. *la classe* `IntegritySystem` *implémente l'interface* `Verifiable`,
2. *la classe* `IntegritySystem` *implémente l'interface* `CryptographicSystem`,
3. *la classe* `BankAccountNumber` *implémente l'interface* `Verifiable`,
4. *on peut créer des instances de l'interface* `Verifiable`,
5. *on peut créer des instances de l'interface* `CryptographicSystem`,
6. *on peut créer des instances de la classe* `IntegritySystem`,
7. *on peut créer des instances de la classe* `RSASignedMessage`,
8. *la classe* `IntegritySystem` *doit définir une méthode de signature* `public boolean verify()`,
9. *la classe* `RSASignedMessage` *doit définir une méthode de signature* `public boolean verify()`,
10. *la classe* `BankAccountNumber` *doit définir une méthode de signature* `public boolean verify()`,
11. *la classe* `SolvedEquation` *doit définir une méthode de signature* `public boolean verify()`,
12. *la classe* `BankAccountNumber` *doit définir une méthode de signature* `public String getSystem()`,
13. *la classe* `SolvedEquation` *doit définir une méthode de signature* `public String getSystem()`,
14. *la classe* `RSASignedMessage` *doit définir un constructeur pour être compilée sans erreur,*
15. *la classe* `SolvedEquation` *doit définir un constructeur pour être compilée sans erreur.*

**Collections** On rappelle que la librairie standard Java définit notamment les classes et interfaces suivantes :

```
public interface Collection<E> {[...]}
public interface List<E> extends Collection<E> {[...]}
public interface Set<E> extends Collection<E> {[...]}
public class ArrayList<E> implements List<E> {[...]}
public class LinkedList<E> implements List<E> {[...]}
public class HashSet<E> implements Set<E> {[...]}
```

On rappelle également que l'interface `Collection` permet de représenter des ensembles finis d'objets, avec des redondances possibles (plusieurs occurrences d'un même élément) et sans ordre particulier, que l'interface `List` permet de représenter des ensembles d'objets *ordonnés* (par leur position dans la liste), avec des redondances possibles, et que l'interface `Set` permet de représenter des ensembles d'objets non nécessairement ordonnés, mais *sans redondance*. Les classes `ArrayList`, `LinkedList` et `HashSet` sont des classes concrètes implémentant l'interface suggérée par leur nom. Enfin, on rappelle que la notation `Classe<E>` signifie que `E` peut être remplacée par n'importe quelle interface ou classe. Par exemple, la classe `ArrayList<String>` représente des listes, de type `ArrayList`, contenant des éléments de type `String`.

Pour les questions suivantes, on s'intéresse à une classe permettant de représenter des films, par les noms de leurs acteurs (chacun de type `String`), leur bande-annonce (de type `Video`) et leurs chapitres (parties successives du film, de type `Video` également). On envisage

- des attributs `actors`, `teaser` et `chapters`,

1. Précisément, si l'on connaît une information de cet auteur, appelée sa « clef publique ».

— un constructeur prenant en argument une valeur pour chacun de ces attributs,
— des méthodes `getActors`, `getTeaser` et `getChapters`.

**Question 2 (2 points)** *Parmi les 6 types de la librairie standard Java listés ci-dessous, indiquer lequel utiliser pour déclarer l'attribut* `actors` *(ligne* `protected ... actors` *dans le code Java) ? Quel type utiliser comme type de retour de la méthode* `getActors` *? Justifier.*

**Question 3 (2 points)** *Même question pour l'attribut* `chapters` *et la méthode* `getChapters`.

**Question 4 (1 point)** *Donner la définition complète, en Java ou en pseudo-code, d'un constructeur de la classe qui soit cohérent avec les réponses aux questions 2 et 3.*

# 2 Conception (10 points)

L'objectif de l'exercice est de concevoir des classes et interfaces pour une application permettant de gérer un jeu, dans lequel des personnages contrôlés par le joueur (PJ, pour « personnage joueur ») se déplacent de zone en zone et rencontrent des personnages contrôlés par le jeu (PNJ, pour « personnage non joueur »). On souhaite en particulier pouvoir représenter le jeu à différents niveaux de détail :
— différentes zones sont reliées (ou non) par des routes (chaque route relie deux zones entre elles),
— à l'intérieur d'une zone, on peut trouver d'autres zones (des « sous-zones »),
— à l'intérieur d'une sous-zone, on peut trouver des sous-sous-zones, et ainsi de suite,
— les routes peuvent relier n'importe quelle zone à n'importe quelle autre (y compris une zone à l'une de ses sous-zones ou sous-sous-zones, par exemple),
— les plus petites zones (qui ne contiennent pas de sous-zones) sont constituées d'un type uniforme de terrain (forêt, mer, etc.) dont dépendent les PNJ qui peuvent être rencontrés (par exemple, des lutins en forêt, des pirates en mer, etc.).

Pour toutes les questions, on pourra répondre soit en donnant un diagramme UML des classes, soit en donnant du code Java ou Python, soit en donnant du pseudo-code, au choix.

On suppose qu'une interface `PNJ`, des classes `Lutin` et `Pirate` implémentant cette interface, et une classe `PJ` sont déjà définies.

**Question 5 (1 point)** *Proposer une interface générique* `Zone` *pour représenter à la fois les zones, les sous-zones, les sous-sous-zones, etc., et contenant seulement une méthode permettant de récupérer la surface de la zone (en mètres carrés).*

**Question 6 (3 points)** *Proposer une interface générique* `Terrain`*, contenant seulement une méthode retournant soit un PNJ, soit* `null`*, à chaque fois qu'elle est appelée (cette méthode servant à déterminer les rencontres). Proposer une classe* `Forêt` *implémentant l'interface* `Terrain` *de sorte que l'on rencontre un* `new Lutin()` *1 fois sur 3 lorsque la méthode correspondante est appelée ; donner le code complet de la classe.*

**Question 7 (1 points)** *Proposer une classe* `Route` *permettant de représenter les routes ; donner seulement les attributs de la classe.*

Pour la représentation précise des zones, on propose de créer deux classes concrètes, l'une permettant de représenter les zones ne comprenant pas de sous-zone, et l'autre permettant de représenter des zones contenant un ensemble (par exemple, de type `Set<...>`) de sous-zones, elles-mêmes pouvant être de tout type implémentant l'interface `Zone`.

**Question 8 (1 points)** *Proposer un type, le plus générique possible, pour les sous-zones d'une zone.*

**Question 9 (3 points)** *Proposer une définition complète des deux classes concrètes pour la représentation précise des zones, en donnant tous les attributs et la liste des méthodes à définir pour chacune (sans donner la définition elle-même).*

**Question 10 (1 point)** *Résumer les réponses aux questions 5 à 9 en un unique diagramme UML ou en un petit texte, en vous assurant que toutes les informations demandées au début de l'exercice peuvent bien être représentées.*