Introspection et annotations

Exercice 1 *L'ArrayList mystère*

Vous disposez d'une classe dans laquelle l'un des attributs est une ArrayList (brute, non paramétrée) qui est créée puis remplie, mais vous ne savez pas de quoi !

Question 1. Écrivez une méthode retournant pour un indice i les méthodes dont vous disposez sur l'objet de rang i.

Question 2. Écrivez une méthode retournant la classe la plus spécifique dont tous les objets de l'ArrayList sont instances. Notez que la méthode de la classe Class `public Class getSuperclass()` retourne la superclasse de la Class appelante.

Exercice 2 *Fabrique de personnages*

Question 3. Ecrivez 3 classes représentant des personnages d'un jeu fictif : une classe abstraite personnage introduisant entre autres un nom, un nombre de points et un nombre de vies ; une sous-classe personnageInvisible (les personnages invisibles peuvent se rendre invisible un certain temps, puis ils redeviennent visibles pendant un temps minimum) introduisant un booléen spécifiant s'ils sont visibles ou pas, le temps pendant lequel ils peuvent rester invisibles et le temps d'attente avant de pouvoir redevenir invisible ainsi qu'une méthode permettant de se rendre invisible ; et les personnages bonus (ces personnages gagnent des vies supplémentaires quand ils franchissent certains seuils de points) qui introduisent tous les combien de points le nombre de vies est augmenté (avec redéfinition de l'accesseur en écriture sur les points qui met à jour le cas échéant le nombre de vies). On introduira les accesseurs et les constructeurs paramétrés et par défaut.

Question 4. Créez une classe fabrique de personnages, dont une méthode permet de prendre en paramètre un nom de classe de personnage et qui retourne une instance de cette classe.

Question 5. Créez une autre méthode qui prend comme paramètre une instance de personnage, puis demande interactivement à un utilisateur de la fabrique les valeurs des attributs de cette instance. Proposez une version avec introspection puis en utilisant la liaison dynamique (en définissant une méthode d'initialisation dans les classes personnage). Notez que la méthode de la class Class `public Field getFields()` retourne tous les attributs publics de la Class appelante. Dans la classe Field les méthodes `public String getName()` et `public void set(Object objet, Object valeur)` permettent respectivement d'obtenir le nom de l'attribut et d'en changer la valeur pour un objet donné.)

Exercice 3 *Migration de personnages*

Question 6. Dans la fabrique de personnages, ajoutez une méthode qui prenne comme paramètre une instance de l'une des classes de personnages (classe origine), le nom d'une autre classe dans laquelle on veut transformer le personnage (classe cible) et retourne un objet de la classe cible dont tous les attributs ont été initialisés :

- pour les attributs communs (de même nom) entre les deux classes, la valeur est recopiée
- pour les attributs de la classe cible que n'a pas la classe origine, la valeur est demandée lors d'une interaction avec l'utilisateur de la fabrique.

Exercice 4 *Annotation Todo*

Question 7. Reprenez l'annotation Todo portant sur les méthodes, utilisable à l'exécution, dont les éléments précisent :

- le type de la tâche à effectuer sur la méthode (écrire, améliorer la complexité, refactoriser, tester)

- le numéro de la version pour laquelle ce doit être effectué
- la durée estimée de la tâche

Question 8. Annotez vos classes personnage avec cette annotation.

Question 9. Écrivez une classe permettant de manipuler ces annotations avec des méthodes permettant, pour un ensemble de classes incluses dans un tableau :

- de lister toutes les annotations Todo,
- de récupérer toutes les annotations correspondant à un certain type de tâche passé en paramètre
- de retourner la durée cumulée des tâches à réaliser lors d'une certaine version.

Exercice 5 *Une méthode toString() générique (F. Mallet)*

Question 10. Utilisez l'introspection pour créer une méthode toString() générique. Cette méthode toString prend en paramètre un objet de type Object et affiche la valeur de chacun de ses champs. Attention, si les champs sont des références sur d'autres objets on descendra en profondeur pour afficher "récursivement" leurs valeurs également.

Question 11. Proposez une variante de la méthode toString admettant un deuxième paramètre qui est la profondeur à laquelle on souhaite descendre dans la structure. Notez la méthode de la classe Class `public boolean isPrimitive()` permet de vérifier si le type de l'élément étudié est un type primitif).

Exercice 6 *Implémentation partielle du lookup de méthodes*

Question 12. Écrivez un programme qui reçoit en entrée le nom d'une méthode et l'objet sur lequel elle doit être invoquée (objet receveur), et qui vérifie si dans la classe de l'objet cette méthode existe (vous vérifierez que l'objet n'est pas null). On utilisera la méthode getDeclaredMethods() de la classe Class. Si le programme ne la trouve pas dans la classe, il va la chercher dans la superclasse, et ainsi de suite. Si celle-ci n'existe pas, le programme devra lever une exception `MethodeInexistanteException`. Sinon, il invoquera la méthode sur l'objet.

Pour commencer, prenez en compte des méthodes sans paramètres. Il faudra vérifier ici que la méthode est accessible avant de l'invoquer (méthode publique pour simplifier), sinon il faudra lever une exception de type MethodeNonAccessibleException.

Exercice 7 *Injecteur de dépendances*

Soient les classes suivantes :

```
class A { B b;
          public void m() {System.out.println("Je suis m dans A."); b.n();}
}
class B { public void n() {System.out.println("Je suis n dans B");} }
```

Question 13. Si l'on écrit dans un main le code suivant, que se passe-t-il ? A a = new A(); a.m();

Afin de résoudre le problème souligné ci-dessus, et au lieu d'initialiser l'attribut b avec une instance de B dans le code, nous allons faire ça depuis un programme externe, que nous devons écrire et qu'on va appeler injecteur de dépendances. Ce genre d'outils existent réellement, comme dans l'édition Java EE (Java Enterprise Edition).

Question 14. Définissez une annotation InjectMe portant sur les attributs et utilisable à l'exécution. Celle-ci sera utilisée pour annoter les attributs qui doivent être initialisés depuis l'extérieur (par l'injecteur).

Question 15. Écrivez un programme permettant de prendre en entrée l'instance d'une classe ayant des attributs non initialisés et annotés InjectMe. Il va ensuite récupérer les types de ces attributs annotés, qui doivent être des

classes, instancier ces classes et affecter les références des instances créése aux attributs. Nous allons simplifier le problème, en considérant que les types des attributs annotés sont des classes qui sont référencées dans le CLASSPATH lors de l'exécution de votre programme (elles sont donc accessibles depuis votre programme, il suffit de les instancier). Nous allons considérer également que ces classes comportent des constructeurs sans paramètres.

Exercice 8 *Objets composites (M. Blay-Fornarino et A Ocello)*

Un objet composite est un objet complexe défini comme un ensemble d'objets, appelés composants, qui décrivent les différentes parties de l'objet composite. Chaque composant peut être lui-même un objet composite. Lorsqu'on appelle une méthode sur un objet composite, cette méthode peut être traitée localement (par le composite lui-même s'il sait traiter cette méthode) ou bien être déléguée à l'un des composants du composite qui saurait la traiter. Du point de vue du code client qui utilise le composite, tout se passe comme si le composite était capable de traiter lui-même toutes les méthodes appelées. Les langages aux capacités d'intercession, sont capables de modifier la façon de traiter l'envoi et la réception de messages pour implémenter ce genre de comportement, par exemple. En Java nous ne disposons malheureusement pas des capacités d'intercession. Toutefois nous allons voir comment il est possible de simuler en partie ce comportement à l'aide de l'introspection et de l'héritage. Pour cela, nous allons créer une classe Composite qui définit une méthode appelle prenant en paramètre une chaîne de caractères et un tableau de paramètres effectifs. Cette méthode permet d'appeler une autre méthode (dont le nom correspond au premier paramètre de appelle et dont les paramètres effectifs correspondent au second paramètre de appelle) sur l'objet lui-même ou sur un de ses attributs non primitifs.

Par exemple, la classe Voiture définit, entre autres, les attributs prix, carrosserie, moteur et la méthode setPrix. Si Voiture hérite de Composite, on aura le comportement suivant pour l'instance maFord :

- maFord.appelle("setPrix", [20000]) appelle setPrix sur l'objet récepteur maFord.
- maFord.appelle("setCouleur", ["rouge"]) appelle setCouleur sur l'attribut car de maFord.
- maFord.appelle("changeRegime", []) appelle changeRegime sur l'attribut mot de maFord

```
import java.lang.reflect.*;

class Composite {
    public void appelle(String s, Object[] params) {
        // TO DO
    }
}

class Voiture extends Composite {
        private Carrosserie car;
        protected Moteur mot;
        protected Double prix;
        protected String options;
        public Voiture(String str) {
                car = new Carrosserie();
                mot = new Moteur();
                prix = new Double(0);
                options = str;
        }
        public String getOptions() {
                System.out.println("Voiture.getOptions " + options);
                return options;
        }
        public void setPrix(Double p) {
                prix = p;
                System.out.println("Voiture.setPrix " + prix);
        }
}


class Scenic extends Voiture {
```

```
        protected Radio radio;
        public Scenic(String str) {
                super(str);
                radio = new Radio();
        }
        public void setPrix(Double p) {
                prix = p; System.out.println("Scenic.setPrix " + prix);
        }
}
class Carrosserie {
    private String couleur;
    public void setCouleur(String couleur) {System.out.println("Carrosserie.setCouleur "-
}
class Moteur {
    private boolean bool;
    public void changeRegime() {
        bool = ! bool;
        System.out.println("Moteur.setRegime "+bool);
    }
}
class Radio {
        private boolean isOn;
        public void on() {
                isOn = true;
                System.out.println("Radio.on");
        }
        public void off() {
                isOn = false;
                System.out.println("Radio.off");
        }
}

class TestComposite {
    public static void main( String[] args ) {
                Voiture v = new Voiture("toit ouvrant");
                Object[] params1 = { 20000.00 };
                Object[] params2 = { "rouge" };
                Object[] params3 = new Object[0];
                v.appelle("getOptions", params3);
                v.appelle("setPrix", params1);
                v.appelle("setCouleur", params2);
                v.appelle("changeRegime", params3);
                System.out.println("*****");
                Senic s = new Scenic("climatisation");
                s.appelle("getOptions", params3);
                s.appelle("setPrix", params1);
                s.appelle("setCouleur", params2);
                s.appelle("changeRegime", params3);
                s.appelle("on", params3);
    }
}
```