TP N°3 – R2.01

Ce TP a pour objectif de jouer avec l'héritage pour comprendre son fonctionnement sur le volet réutilisation de code et sous-typage. Les classes développées ici n'ont donc aucun intérêt autre que pédagogique ! Il n'y a pas de classes Java à rendre mais uniquement un document au format impérativement `pdf` détaillant vos réponses aux questions du TP.

Question 1 : mise en place des classes

a) Écrivez dans votre répertoire de TP3 de R2.01 en Java les 5 classes A, B, C, D, E dont le code est fourni en annexe (faites des copier/coller depuis ce document).

b) Compilez ce code qui ne comporte, a priori, aucune erreur.

c) Vérifiez la structure de votre sous-répertoire `class`. Qu'observez-vous ? Comment `javac` gère la notion de package ?

d) Exécutez ce code. Faites état de l'affichage obtenu.

Question 2 : héritage et chaînage obligatoire des constructeurs

Le chainage des constructeurs est <u>obligatoire</u> entre chaque constructeur d'une sous-classe et l'un des constructeurs de son unique super-classe. Il peut être réalisé soit par un *chaînage explicite* du programmeur qui insère le mot clé `super(…)` en première instruction d'un constructeur soit par un *chaînage automatique* réalisé par le compilateur qui ajoute en première instruction d'un constructeur le mot clé `super()`. Le chainage automatique n'est possible qu'à la réserve que la super classe dispose d'un constructeur sans paramètre (sinon la compilation de la sous classe échoue).

a) Détaillez avec vos connaissances et en vous aidant de l'affichage produit à l'exécution comment sont chaînés les constructeurs :

- de B et A
- de C et B
- de D et A
- de E avec Object.
- De A avec Object

b) Que se passe-t-il si l'on met en commentaire l'instruction `super(-10);` dans le constructeur de la classe C puis que l'on recompile la classe C ? Expliquez le problème.

c) Que se passe-t-il si cette fois ci, on place cette instruction `super(-10) ;` du constructeur de la classe C après l'affichage (donc en deuxième instruction). Expliquez le problème.

Question 3 : héritage et réutilisation de code avec et sans usufruit

Les 4 niveaux de visibilité Java (`public, protected, (rien), private`) ont une double sémantique : i) pour *l'accès inter-objets* (un objet accède au membre d'un autre objet) et ii) pour *l'accès d'héritage* (un objet accède à un membre dont il a hérité). Ces accès diffèrent selon que les deux classes en présence sont situées ou non dans le même package. Dans le cas de l'accès d'héritage, une classe hérite toujours de tous les membres de ses classes ancêtres. Mais elle n'a pas nécessairement l'accès direct dans son code à ces membres hérités (on parle d'usufruit). Lorsqu'un accès inter-objets se fait sur un membre hérité (donc impliquant pour l'objet receveur un accès d'héritage), c'est la visibilité inter-objets de la classe cliente par rapport à la super-classe propriétaire du membre dont on réalise l'accès qui s'applique.

a) Rajoutez dans la classe C une méthode de signature `public void m5()` dont l'objectif est de :
   - Vérifiez quels sont les *accès d'héritage* qui sont autorisés dans `C` (donc avec usufruit) sur tous les membres hérités de `A` (donc selon les 4 niveaux de visibilité). Mettez ensuite en commentaire les accès refusés par le compilateur. On étudie ici l'accès d'héritage dans le cadre d'une sous-classe (C) située dans un package différent de la super-classe dont elle hérite (A via B)
   - Mettez en évidence que même l'attribut `a1` est bien présent dans une instance de la classe C et que l'on peut même obtenir sa valeur et l'afficher.

b) Rajoutez dans la classe B une méthode de signature `public void m6()` dont l'objectif est de :
   - Vérifiez quels sont les *accès d'héritage* qui sont autorisés dans B (donc avec usufruit) sur tous les membres hérités de A. Mettez ensuite en commentaire les accès refusés par le compilateur. On étudie ici l'accès d'héritage dans le cadre d'une sous-classe (B) située dans le même package que la super-classe dont elle hérite (A).

Question 4 : Accès inter-objets avec ou sans accès d'héritage

a) Rajoutez deux méthodes `protected void m7()` et `void m8()` dans la classe `C`. Ces deux méthodes ne font qu'un affichage de leur nom. Complétez le code du `main()` pour vérifier quels sont les membres (héritées de A ou définies dans la classe C elle-même, pour étudier les 4 niveaux de visibilité) d'une instance de la classe C qui sont appelables depuis le `main()`. Ce faisant vous étudiez les *accès inter-objets* qui sont licites entre deux objets instances de deux classes du même package (ici les classes C et E du package `client`) en distinguant les accès inter-objets qui sont directs (membres définis par la classe C) de ceux qui impliquent un accès d'héritage (membres de C hérités de A).

b) Réalisez le même travail depuis le `main()` sur une instance de la classe A. Vous étudiez ce faisant l'accès inter-objets entre deux classes situées dans des packages différents (sans accès d'héritage nécessaire).

c) Sur la base des expérimentations menées vérifiez si les 3 tableaux ci-dessous sont corrects.

Accès d'héritage avec usufruit (oui/non) dans le code de B qui est sous classe directe ou indirecte de A.

Visibilité Java du membre de A	A et B sont dans le même package	A et B dans des packages ≠
`private`	non	non
`(rien)`	oui	non
`protected`	oui	oui
`public`	oui	oui

Accès inter-objets <u>direct</u> autorisé (oui/non) si un objet B est client d'un objet A d'un membre défini par A

Visibilité Java du membre de A	A et B sont dans le même package	A et B sont dans des packages ≠
`private`	non (oui si deux instances de la même classe A=B!)	non
`(rien)`	oui	non
`protected`	oui	non
`public`	oui	oui

Accès inter-objets (indirect) autorisé (oui/non) si un objet B est client d'un objet A d'un membre dont A a hérité d'une classe C (c'est la visibilité de C qui sert de référence !)

Visibilité Java du membre de C	A et C sont dans le même package	A et C sont dans des packages ≠
`private`	non	non
`(rien)`	oui	non
`protected`	oui	non
`public`	oui	oui

Question 5 : premier pas dans l'héritage et la redéfinition

La redéfinition est le moyen pour une sous-classe de proposer un nouveau code à une méthode héritée avec usufruit. La signature de la méthode redéfinissante doit être identique à celle de la méthode redéfinie (la visibilité peut augmenter cependant, et le type de retour être « covariant »). Toute demande d'exécution de la méthode redéfinie exécutera la nouvelle version de la méthode. Il est cependant possible dans le code de la sous-classe de demander explicitement dans son code le lancement de la version redéfinie avec le mot clé `super.m(…)`.

En analysant l'affichage résultant de l'envoi d'un message `m2()` à une instance de la classe A, de la classe B, de la classe C et de la classe D depuis le `main()`. Expliquez ce que font les redéfinitions présentent dans ce code.

Question 6 : premier pas dans l'héritage et le sous-typage

En Java, l'héritage entraine le sous-typage. Donc une sous classe B d'une classe A vérifie que `B<:A` (lire B est sous type de A). Partout ou l'on attend un type A on acceptera à la compilation un type B. Le système de contrôle de type de Java est sûr mais ne travaille que sur le type statique (de déclaration des variables). En effet déterminer le type (dynamique) réel de l'objet point par une variable est un problème indécidable. Le compilateur Java ne cherche donc jamais à faire ce calcul. Une instruction est donc correctement typée à la compilation et à l'exécution si : i) le type statique de la variable receveuse est un un super-type du type statique de l'expression qu'on lui affecte, ii) le type statique de la variable dispose bien d'une méthode de la signature réclamée. Il est possible de forcer la main par transtypage (cast en anglais) au compilateur pour qu'il laisse passer (sous certaines conditions) une instruction ne respectant les deux règles ci-dessus. Il génère cependant du code pour vérifier qu'à l'exécution vous ne l'avez pas « trompé ». Si c'est le cas, la JVM lance une exception.

a) Complétez le `main()` fourni au départ avec le code ci-dessous. Indiquez quelles sont les instructions qui compilent et celles qui ne compilent pas. Expliquez pourquoi en vous appuyant sur la règle de sous-typage par héritage.

```
unA=unB;

unA=unD;

unB=unC;

unA=unC;

unA.m5();

unC=unA;
```

b) Que révèle le résultat de la compilation de l'instruction `unA.m5() ;` sur le fonctionnement du compilateur ? S'il avait laissé passer cette instruction l'exécution se serait-elle finalement bien passée ? Demandez à `chatGPT` de vous expliquer simplement ce phénomène (précisez lui bien dans votre prompt que C est une sous-classe Java de A).

c) On enlève le code précédent et on met le code ci-dessous à sa place. Identifiez les instructions qui compilent ou non. Lancez l'exécution. Que constatez-vous ? Demandez à `chatGPT` de vous expliquez simplement ce qui se passe. Que conclure sur l'usage du transtypage (cast) en Java ?

```
unA=unC;

unC=(C)unA;

unC.m5();

unA=unD;

unC=(C)unA;
```

Travail à rendre :

Un unique document `pdf` qui porte votre nom et prénom et le numéro de TP contenant les réponses à toutes les questions du TP3.

Annexe

```
package jeux;
public class A {
      private int a1;
      int a2;
      protected int a3;
      public int a4;

      public A(){
            System.out.println("Constructeur de A");
            this.a1=1;
            this.a2=2;
            this.a3=3;
            this.a4=4;
      }

    public int getA1(){
      System.out.println("getA1 de A");
      return this.a1;
    }
    protected void m2(){
      System.out.println("m2 de A");
    }
    void m3(){
      System.out.println("m3 de A");
    }
    private void m4(){
      System.out.println("m4 de A");
    }
}
```

```
package jeux;
public class B extends A{
      private int b1;
      public B(int i){
            System.out.println("Constructeur de B");
            this.b1=i;
      }
      protected void m2(){
            System.out.println("m2 de B");
```

```
            super.m2();
    }
}
```

```
package jeux;
public class D extends A{
      private int d1;
      public void m2(){
            System.out.println("m2 de D");
    }
}
```

```
package client;
import jeux.B;
public class C extends B{
      private int c1;

      public C(){
            super(-10);
            System.out.println("Constructeur de C");
      }
    public void m1(){
      System.out.println("m1 de C");
    }
}
```

```
package client;
import jeux.*;
public class E{
      public static void main(String args[]){
            A unA=new A();
            B unB=new B(3);
            C unC =new C();
            D unD =new D();
            E unE=new E();
      }
}
```