JAVA POO - TP3

Table des matières

Toutes les lignes en commentaires dans les captures d'écran sont des lignes de code qui ne compilent pas !

I. Mise en place des classes

C/

```
mgouelo@mgouelo-HP-Pavilion-Laptop-15-eh1xxx:~/D
IE1/POO/TP3/class$ tree
.
├── client
│   ├── C.class
│   └── E.class
└── jeux
    ├── A.class
    ├── B.class
    └── D.class

3 directories, 5 files
```

D/

```
mgouelo@mgouelo-HP-Pavilion-Laptop-15-eh1xxx:~/D
IE1/POO/TP3/ws$ javac -d ../class ../src/*.java
mgouelo@mgouelo-HP-Pavilion-Laptop-15-eh1xxx:~/D
IE1/POO/TP3/ws$ java client.E
Constructeur de A
Constructeur de A
Constructeur de B
Constructeur de A
Constructeur de B
Constructeur de C
Constructeur de A
```

II. Héritage et chaînage des constructeurs

A/

B -> A : chaînage automatique par le compilateur car le constructeur de la super-classe ne nécessite pas de paramètre.

C -> B : chaînage explicite (*super(-10)*) car la super-classe B requiert un paramètre de type int.

D -> A : chaînage automatique réalisé par le compilateur JAVA car la super-classe A n'a pas de paramètre.

E -> Object : c'est un chaînage automatique car la super-classe Object n'a pas de paramètre

A -> Object : c'est un chaînage automatique car la super-classe Object n'a pas de paramètre

B/

Si on compile la classe C en mettant l'instruction *super(-10)* en commentaire on obtient l'erreur suivante :

> *error : constructor B in class B cannot be applied to given types;*

En effet, car pour compiler la classe C, il est nécessaire de fournir un paramètre de type *int* demandé par la super-classe B. Donc on ne peut pas compiler C avec un chaînage automatique. Il est donc nécessaire de conserver cette instruction dans le code.

C/

En compilant C en mettant l'instruction *super(-10)* après le *System.out.println()* on obtient l'erreur suivante :

> *error : call to super must be first statement in constructor*

L'erreur nous rappelle que l'instruction *super()* doit être impérativement la première instruction du constructeur de la sous-classe

III. Héritage et réutilisation de code avec et sans usufruit

A/

Dans la classe C, les instructions ne compilant pas sont :

> *this.a1 = 2;*

> *this.a1 = 3;*

> *this.m3();*

> *this.m4();*

Dans la classe B, les instructions ne compilant pas sont :

> *this.a1 = 2;*

> *this.m4();*

IV. Premier pas dans l'héritage et la redéfinition

```
// unC.c1 = 3;
unC.m7();
unC.m8();
unC.m1();

// unC.a1 = 3;
// unC.a2 = 4;
// unC.a3 = 5;
unC.a4 = 6;

// unA.a1 = 3;
// unA.a2 = 4;
// unA.a3 = 5;
unA.a4 = 6;
```

V. Premier pas dans l'héritage et le sous-typage

A/

```
// unA.m2();
// unB.m2();
// unC.m2();
unD.m2();
```

Analyse :

> *unA.m2();*

Ne compile pas car la méthode *m2()* provient de la classe A dans le package Jeux et cette méthode *protected* est appelée dans la classe E qui n'est pas une sous-classe de A et qui est dans le package Client.

> *unB.m2();*

Ne compile pas car la méthode est redéfinie mais toujours en *protected*. B étant dans un package différend que E cela ne compile donc pas.

> *unC.m2();*

C est dans le même package que E. C hérite de B qui hérite de A. Donc la méthode *m2()* utilisée pour les instances de C est protected. Or E n'est pas dans le même package que A et B donc on ne peut pas appeler une méthode héritée de A depuis E.

> *unD.m2();*

La classe D est une classe fille de la classe mère A dans laquelle on retrouve la méthode *m2()* redéfinis de *protected* à *public* ce qui fait que cette méthode peut être appelée n'importe où même depuis un autre package. C'est la raison pour laquelle cette ligne de code compile et s'exécute.

VI. Premier pas dans l'héritage et le sous-typage

A/

```
unA=unB;
unA=unD;
unB=unC;
unA=unC;
// unA.m5();
// unC=unA;
```

L'avant dernière ligne de compile pas car on applique sur un objet de la classe mère A une méthode de la classe fille C. C hérite de A mais cela n'est pas réciproque donc A ne connait pas les méthodes de la sous classe C et ne peut donc les manipuler.

Dans la dernière instruction, on souhaite stocker dans une variable de type C une variable de type A. Or A ne possède pas nécessairement les mêmes caractéristiques de C donc cela ne compile pas par prévention.

B/

Ce que l'erreur révèle sur le comportement du compilateur c'est qu'il agit en premier lieu sur les types uniquement. Dans notre code de *main()* on crée un objet de type A puis utilise cet objet avec la méthode *m5()*. Le compilateur ne voyant pas de méthode *m5()* dans le code de la classe de A, celui-ci détermine que ce type et cette méthode sont incompatibles et retourne une erreur.

Si le compilateur n'avait pas refusé cette instruction :

Dans la méthode *m5()* on utilise exclusivement des attributs et des méthodes de la classe A donc l'exécution se serait réalisée sans encombre d'où le caractère pessimiste du comportement du compilateur JAVA

C/

Tout le bloc de code fournis sur le sujet compile. A l'exécution, seule la dernière instruction provoque une erreur. Ce que l'on peut conclure c'est qu'on ne peut pas "caster", échanger les types entre 2 sous-classes sœurs. L'intérêt du *cast* est qu'il peut remédier au problème rencontré dans la question précédente et d'utiliser les méthodes d'une sous-classe.