JAVA 10 : L'HÉRITAGE

Quelques principes fondamentaux de programmation :

Il y a 2 grandes règles (formulée ici par Kent Beck) que tout développeur devrait connaître :

1. Le système (code et tests pris dans leur ensemble) doit communiquer tout ce que vous avez l'intention de communiquer.
2. Le système ne doit contenir aucune duplication de code.

=> Ces deux contraintes constituent la règle : Une Fois et Une Seule.

Du code en trop

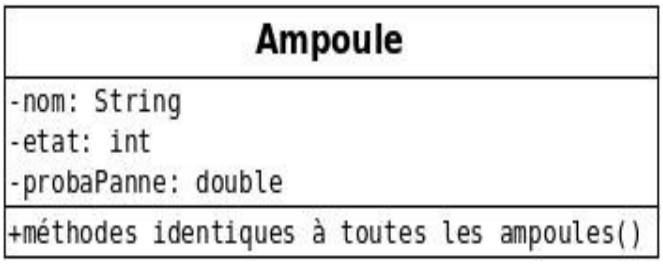
Lors de l'écriture des différentes classes d'ampoules, vous avez reproduit le même code à plusieurs reprises, souvent par copier/coller. Cette méthode de programmation, si elle permet d'aller vite dans un premier temps, rend les applications extrêmement difficiles à maintenir puisqu'en cas d'erreur dans le code dupliqué il faut penser à corriger l'erreur dans toutes les copies. Le principe DRY (don't repeat yourself) permet d'éviter ce type de problème.

Pour éviter cette duplication de code, nous disposons en Programmation Orientée Objet de l'héritage. L'héritage est une forme de réutilisation du code qui permet aux programmeurs de développer de nouvelles classes à partir de classes existantes.
La classe existante est appelée classe de base ou classe mère, la nouvelle classe est appelée classe dérivée ou classe fille. L'un des principaux avantages de l'héritage est qu'il permet de réutiliser le code d'une classe de base.

On va donc créer une classe de base Ampoule qui va contenir, toutes les méthodes communes et les attributs communs à chaque type d'ampoule.

Chaque ampoule est caractérisée par :

- un nom (Led, incandescente, tube)
- un état
- une probabilité de panne



Travail à faire :

1– Importer dans votre workspace le projet luminairebis . Télécharger le fichier luminairebis.zip (extraire le fichier dans votre workspace, puis choisir : file → import → Existing Projets into workspace → choisir le projet luminairebis)
2 - Identifiez dans les différentes classes d'ampoules les méthodes qui sont strictement identiques. Quel est leur nombre ? Y a-t-il des attributs communs ?
3 - Que devient la méthode probaPanne(), si l'on considère que la probabilité de panne est un attribut de chaque classe d'ampoule ?
4 – Créer une classe Ampoule, conforme à la représentation UML. Créer dans la classe Ampoule un constructeur qui recevra en argument tous les attributs de la classe.
(Source → Generate Constructor using Fields)

Désormais nous disposons d'une classe de base ou classe mère, à partir de cette classe nous allons pouvoir créer d'autre classe, les classes filles.

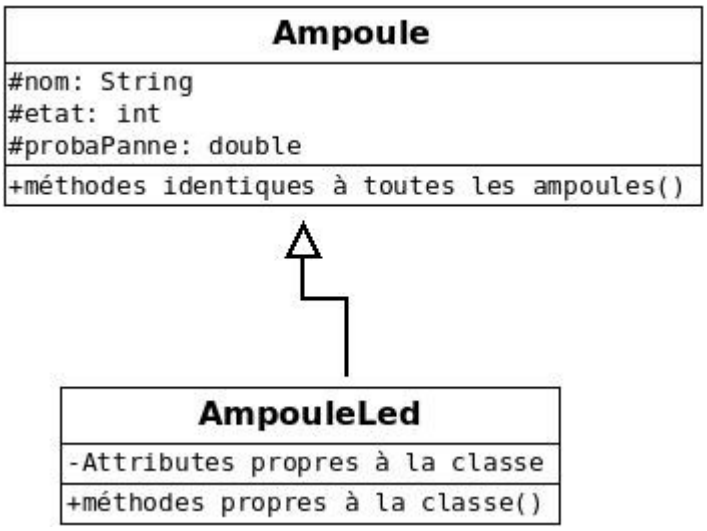
Pour créer une classe fille à partir d'une classe mère, il faut ajouter à la déclaration de la classe le mot clef **extends** :

```
public class AmpouleLed extends Ampoule implements Eclairage
```

Pour que les attributs de la classe mère soient accessibles dans les classes filles, il faut les déclarer **protected** et non **private**. Un attribut **protected** est un attribut qui ne peut être utilisé qu'à l'intérieur de la classe ou dans les classes dérivées.

Ainsi la classe fille, va pouvoir accéder à toutes les méthodes déclarer dans la classe mère et notamment au constructeur. Pour faire appel au constructeur de la classe mère, on utilise le mot clef : **super.** : Source → Generate Constructors from Superclass

Notre diagramme UML devient :



Travail à faire :

1 – Dans la classe Ampoule modifier la visibilité des attributs (remplacer private par protected).
2 – Faites dériver (hériter) les différentes classes d'ampoules de la classe mère : Ampoule.
3 – Dans les classes filles, supprimer tous les attributs et méthodes déclarées dans la classe mère.
4 – Tester vos différentes classes dans App.java

```
AmpouleLed eclairage = new  AmpouleLed("ampoule Led",0,1) ;
```

5 – Tester des luminaires avec différents types d'ampoules.
6 - Réutilisez vos travaux , pour créer une nouvelle classe d'ampoule, AmpouleTriState, qui a une probabilité de panne de 0,5% et qui n'accepte que 3 états en dehors de la panne : [0] éteinte, [5] mi-puissance, [10] pleine puissance.
7 – Quelles solutions adopter pour éviter la duplication du code des méthodes intensifier et diminuer ? Proposez au moins 2 solutions en expliquant leurs avantages et les inconvénients respectifs.

TP : Calcul des impôts locaux

Dans le cadre de l'informatisation d'une mairie, on veut automatiser le calcul des impôts locaux. On distingue deux catégories d'habitation : les habitations à usage professionnel et les maisons individuelles, l'impôt se calculant différemment selon le type d'habitation. Pour cela, un développeur du service informatique a défini les classes HabitationPro et HabitationIndividuelle : (Annexe B)

L'impôt d'une maison individuelle se calcule en fonction de la surface habitable à raison de 2€ le m2, du nombre de pièces et de la présence ou non d'une piscine. On compte 50€/pièce et 100€ supplémentaire en cas de présence d'une piscine .

L'impôt d'une habitation professionnelle se calcule en fonction de la surface occupée par le bâtiment, à raison de 2€ le m2. et du nombre d'employés travaillant dans l'entreprise. On compte 100€ supplémentaire de 10 à 49 employés, 200€ de 50 à 99 employés et 300€ pour un nombre d'employés supérieur ou égal à 100.

Travail à faire :

1 – Créer un nouveau projet Habitation. Copier dans ce projet les classes HabitationIndividuelle et HabitationPro.

2 - Proposez une solution basée sur l'héritage, qui permettent d'éviter d'avoir à écrire les méthodes communes aux 2 types d'habitation.

3 – Proposez une interface Android de ce type :