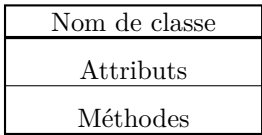
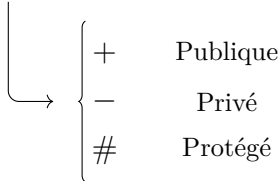
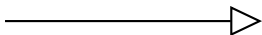
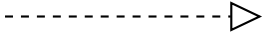


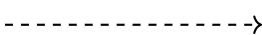


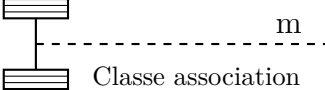



LIF-13

## TD2. Diagramme de classes (Fil rouge Épisode I). (correction)

### 1 Rappel

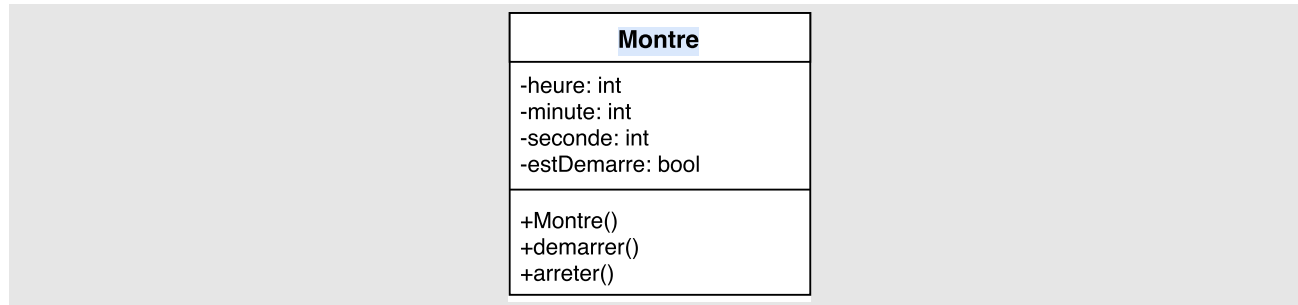
Symboles	Description
	<p>Le premier encart contient le nom de la classe.</p> <p>Le second encart contient ses attributs, c'est-à-dire les caractéristiques de la classe.</p> <p>Le troisième encart contient les méthodes de la classe, c'est-à-dire les fonctions qui peuvent être appelées.</p> <p>Les attributs comme les méthodes peuvent être publiques, privées ou protégées.</p>
	<p>Les attributs/ méthodes publiques sont accessibles par tous, ceux qui sont privés ne le sont que par l'objet qui les contient et ceux qui sont protégés sont accessibles par l'objet et ses descendants.</p>
 <p>Généralisation</p>	<p>La classe à l'origine de la flèche hérite de toutes les propriétés (prototypage, attributs, méthodes) de la classe au bout de la flèche.</p>
 <p>Réalisation</p>	<p>Une classe ne peut hériter que d'une autre classe.</p> <p>Une implementation doit pointer sur une interface, c'est-à-dire une classe particulière qui n'a aucun attribut.</p> <p>L'implémentation est la garantie que la classe source contient toutes les méthodes de l'interface. Une classe peut avoir plusieurs implémentations</p>
 <p>Association</p>	<p>Une association indique qu'il y a des attributs des classes concernées dans les instances des classes associées.</p> <p>La cardinalité des extrémités indique à combien d'instances de la seconde classe, une instance de la première classe doit être liée.</p>
 <p>Navigabilité</p>	<p>La navigabilité indique que l'association ne peut se lire que dans un seul sens alors que par défaut elle peut se lire dans les deux sens.</p>
 <p>Dépendance</p>	<p>La classe source a besoin de la classe cible.</p>
 <p>Agrégation</p>	<p>L'agrégation représente une relation d'inclusion structurelle de type tout/partie.</p>
 <p>Composition</p>	<p>La composition représente une relation de contenance structurelle. Si l'objet composite est détruit, les composantes ne peuvent plus exister.</p>
 <p>Classe association</p>	<p>La classe association indique que l'association entre deux classes se fait par l'intermédiaire d'une troisième classe qui n'existe que dans le cadre de cette association.</p>
 <p>Association n-aire</p>	<p>L'association n-aire permet de faire des associations pour plus de deux classes.</p>

## 2 Fil rouge - Montre digitale

Nous reprenons ici l'exercice "fil rouge" de montre digitale (par exemple une smartwatch) en y apportant un diagramme de classe. Nous allons pour ce faire adopter une démarche de développement dite *incrémentale* ou *itérative* pour améliorer progressivement le modèle proposé et identifier, puis dégager, les aspects génériques qui sont intéressants de réutiliser. Le client de la montre désire en outre deux nouvelles fonctionnalités : le fait que cette dernière puisse être arrêtée et démarrée.

La démarche de modélisation de vos programmes est primordiale pour deux aspects (entre autre). De part son côté universel, cela permet d'avoir un langage commun pour communiquer avec les autres programmeurs de votre équipe. De part ses caractéristiques intrinsèques, il permet de concevoir des solutions flexibles, permettant de s'adapter aux changements que les clients de vos applications pourraient demander lors du développement de votre programme, ainsi que de pouvoir réutiliser des morceaux de vos anciens codes dans d'autres projets et prévenir divers problèmes de logique interne.

**Question 1** *Donner le diagramme de classes d'une application de montre à affichage digital (heures, minutes et secondes), en considérant que l'heure est réellement mémorisée par les 3 attributs correspondants (pas par un seul entier de type "timestamp"). Une telle montre peut être arrêtée, et, le cas échéant, doit pouvoir être redémarrée. Vous ferez apparaître le constructeur de la classe. Vous porterez une attention particulière à la visibilité des différents éléments. Vous ne ferez pas (encore) apparaître les fonctions nécessaires au passage du temps (incrémenterSeconde, incrémenterHeure, incrémenterMinute, etc.) et à l'affichage.*



### Question 2

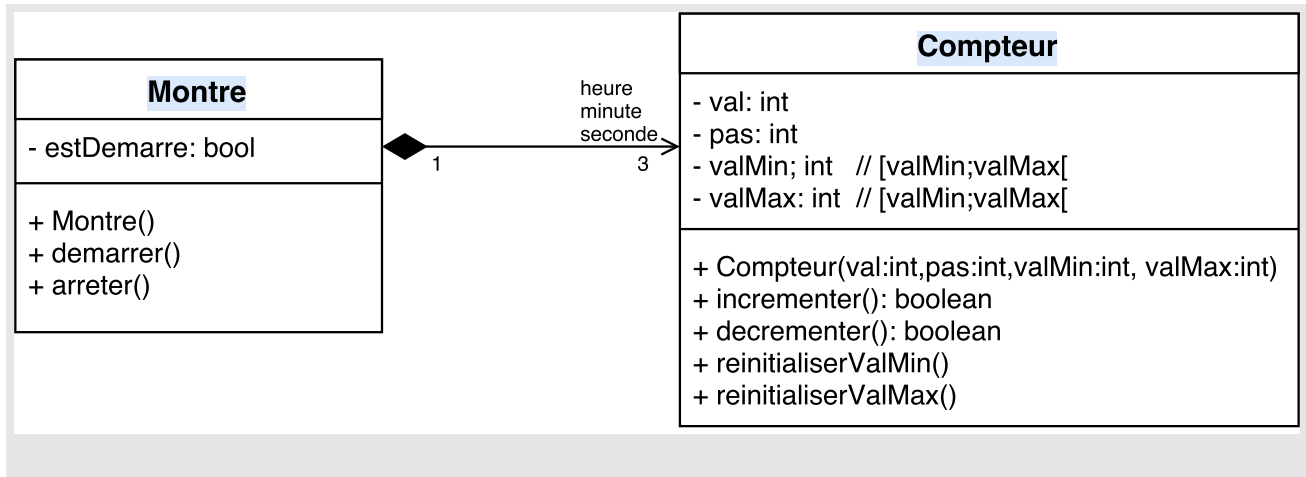
Nous voulons maintenant gérer l'avancement du temps de la montre. Pour ce faire, plutôt que d'avoir un fonctionnement spécifique pour les secondes, les minutes et les heures, nous allons utiliser le **concept générique de COMPTEUR**. Un compteur est une classe générique et réutilisable permettant d'avancer d'une valeur de départ à une valeur de fin, avec un pas donné, qui peut être réinitialisé.

Notre montre se montrerait maintenant versatile aux différentes situations de comptage, telles que :

- format 12/24 heures,
- 364 jours et 364 jours 1/4,
- changer les extrema (par exemple [1;61[ plutôt que [0;60[),
- modifier le pas de temps (on peut vouloir des compteurs à la demi seconde près),
- ...

**Modifier le diagramme de classes de la montre** afin d'introduire le concept générique

de compteur.



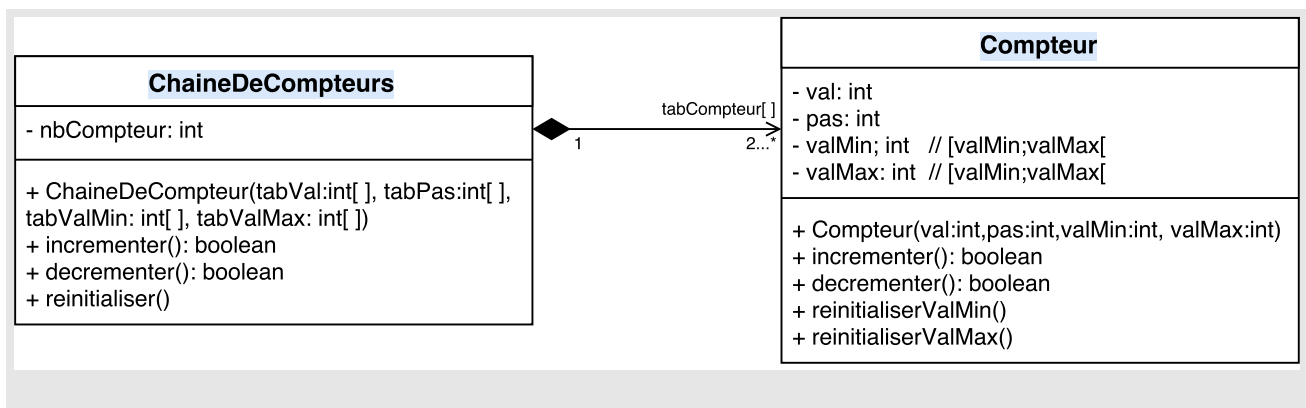
### Question 3

Nous souhaitons maintenant **abstraire le concept de montre**. Une montre n'est qu'un cas particulier d'une chaîne de compteurs qui s'incrémente selon une condition donnée. Cela permettrait de pouvoir :

- compter des jours et des mois,
- compter à rebours,
- compter à la manière des mayas (base 20),
- compter par groupe de 10,
- ...

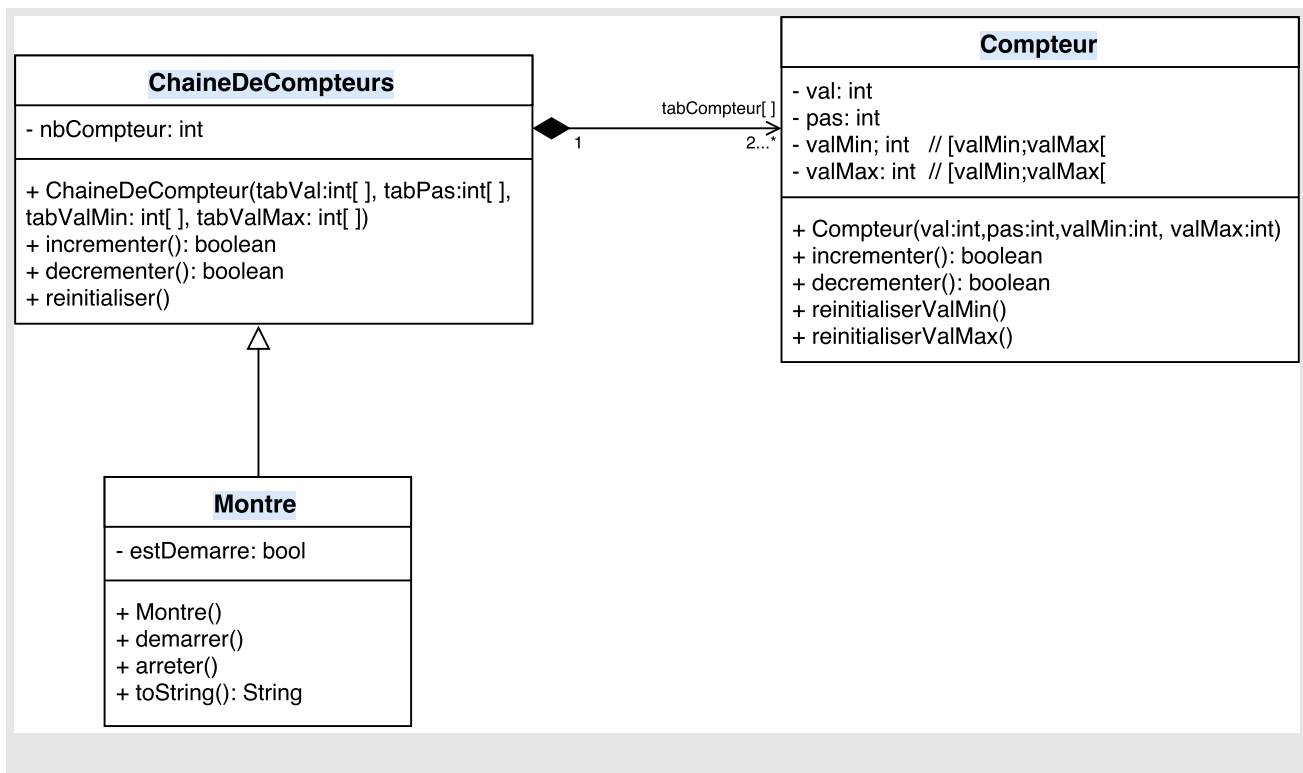
L'idée est donc d'extraire un concept qui soit assez générique pour qu'il serve d'interface à la manipulation de compteurs liés les uns aux autres. Par exemple, lorsqu'on incrémente, décrémenter ou réinitialise la valeur courante, les mécanismes internes n'ont pas à être connus des utilisateurs.

**Proposer un nouveau diagramme de classes**, le plus générique possible, qui modélise cette notion de chaîne de compteur qui remplace le concept de montre.



### Question 4

**Compléter le diagramme de classes précédent** pour modéliser la classe *Montre*.



## Question 5

Une fonctionnalité intéressante serait de pouvoir **comparer deux horaires entre eux** (par exemple, pour savoir si une montre est en avance par rapport à une autre).

Pour tirer parti de la puissance du paradigme-objet, il faut correctement décomposer ce que l'on cherche à obtenir comme comportement. Si l'on compare directement des montres entre elles, on va peut être vouloir comparer des propriétés supplémentaires qu'uniquement l'horaire. Il faut donc pouvoir **générer ce concept d'horaire à partir des montres**, puis les comparer. Enfin, le comportement de comparaison peut être vu comme une action universelle : on peut comparer différents objets entre eux. Ici, on cherche à spécialiser ce comportement pour le concept d'horaire, de telle sorte qu'un horaire puisse se comparer à un autre.

**Compléter le diagramme de classes précédent** pour générer un concept d'horaire à partir des montres et comparer deux horaires entre eux.

