Programmation orientée objet et temps réelle avec Java

Héritage de classes (partie 2)

Dominique Blouin
Ingénieur de recherche
Télécom Paris, Institut Polytechnique de Paris
dominique.blouin@telecom-paris.fr

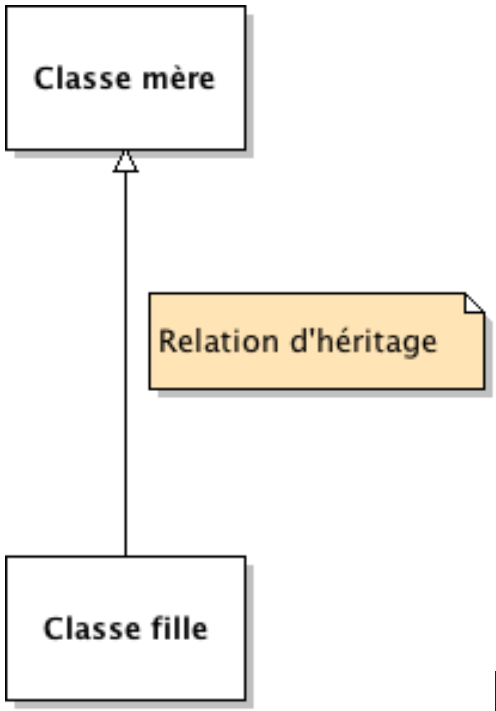
Objectifs d'apprentissage

- **Classes abstraites**
- **Méthodes abstraites**
- **Bonnes pratiques de modélisation**

Rappel: l'héritage de classe

- Une classe **A** peut déclarer qu'elle **hérite** d'une autre classe **B**.
  - Cette classe A est dite classe **fille** ou **sous-classe** de la classe **B**.
  - La classe **B** est dite classe **mère** ou **super-classe** de la classe **A**.

- Signification: La classe fille **hérite** des déclarations faites dans la classe mère.



Redéfinition de méthodes: exemple

```
public class Item {  // Un article dans un magasin

    private double netPrice;

    public double getNetPrice() {
        return netPrice;
    }

    public double getVAT() {      // VAT = Value Added Tax
        return 0.185 * netPrice ; // 18,5%
    }

    public double getATIPrice() { // ATI = All Taxes Included
        return netPrice + getVAT() ;
    }

    ...
}
```

Héritage: exemple

- Le mot clé **`extends`** permet de déclarer la relation d'héritage :

```
public class LuxuryItem extends Item {

    @Override
    public double getVAT() {
        return 0.33 * getNetPrice() ; // 33% tax rate
    }

    ...
}
```

- **`@Override`** est une annotation. Elle est une indication destinée au compilateur pour lui signifier qu'il s'agit d'une **redéfinition de méthode**. Le compilateur vérifiera que c'est bien le cas.
  - L'annotation **`@Override`** n'est pas obligatoire mais **fortement recommandée**.

Fonctions de l’héritage

- **Modélisation** :
  - Etant donnée une classe d'objets, on la **partitionne** en sous-classes. Ainsi une classe **`Shape`** peut être partitionnée en des sous-classes spécialisées : **`Circle`**, **`Square`**, etc.
  - Etant donnée une classe d'objets, on peut la **raffiner** en créant une sous-classe. Par exemple, une classe **`Student`** peut être spécialisée en une classe **`TelecomParisStudent`** décrivant les spécificités des étudiants de Télécom Paris.

- **Architecture logicielle** :
  - Les sous-classes d'une classe **partagent** le code des méthodes et les attributs de la classe mère.

Significations de l'héritage

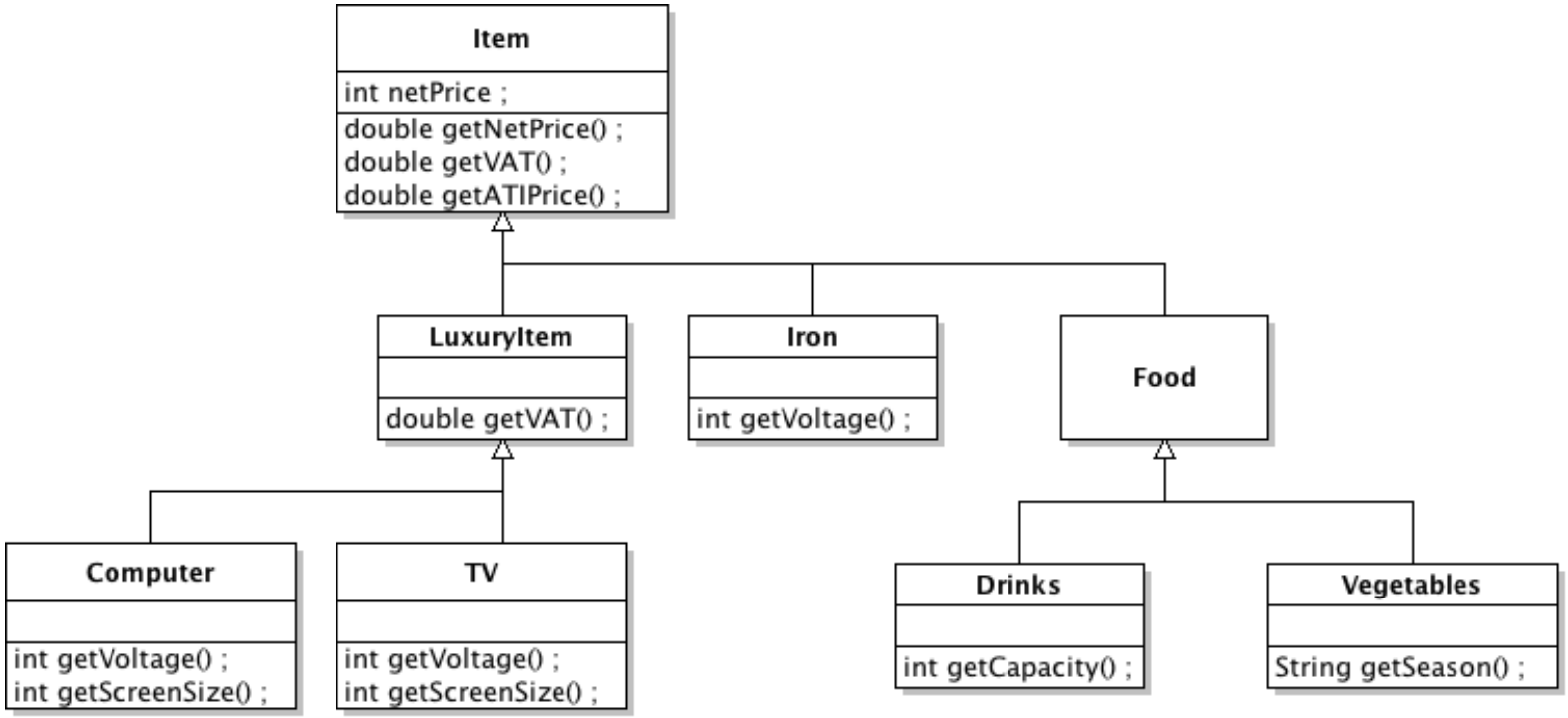
- Est une sorte de ....
- Est un genre de ....
- Est une catégorie de ....
  - Ainsi, une TV est une **sorte** d'Item.

- Est une extension de ...
  - Ainsi, un **`ColouredPoint`** est une extension de **`Point`**.

- Est une spécialisation de ...
- Est un cas particulier de ...
  - Ainsi, un **`LuxuryItem`** est une spécialisation d'**`Item`**.

- Sans oublier le simple partage de code.

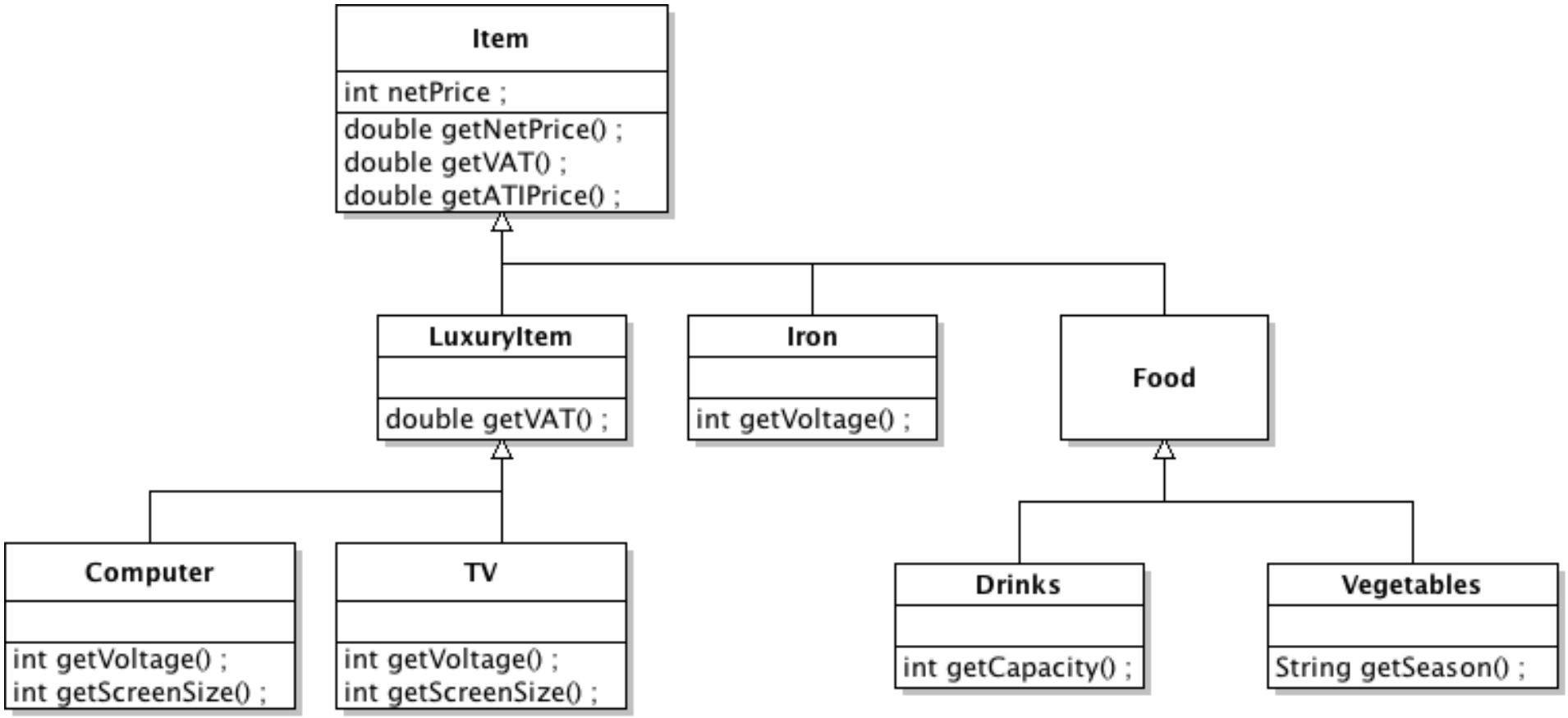
Classe abstraite:
Exemple des articles d'un magasin



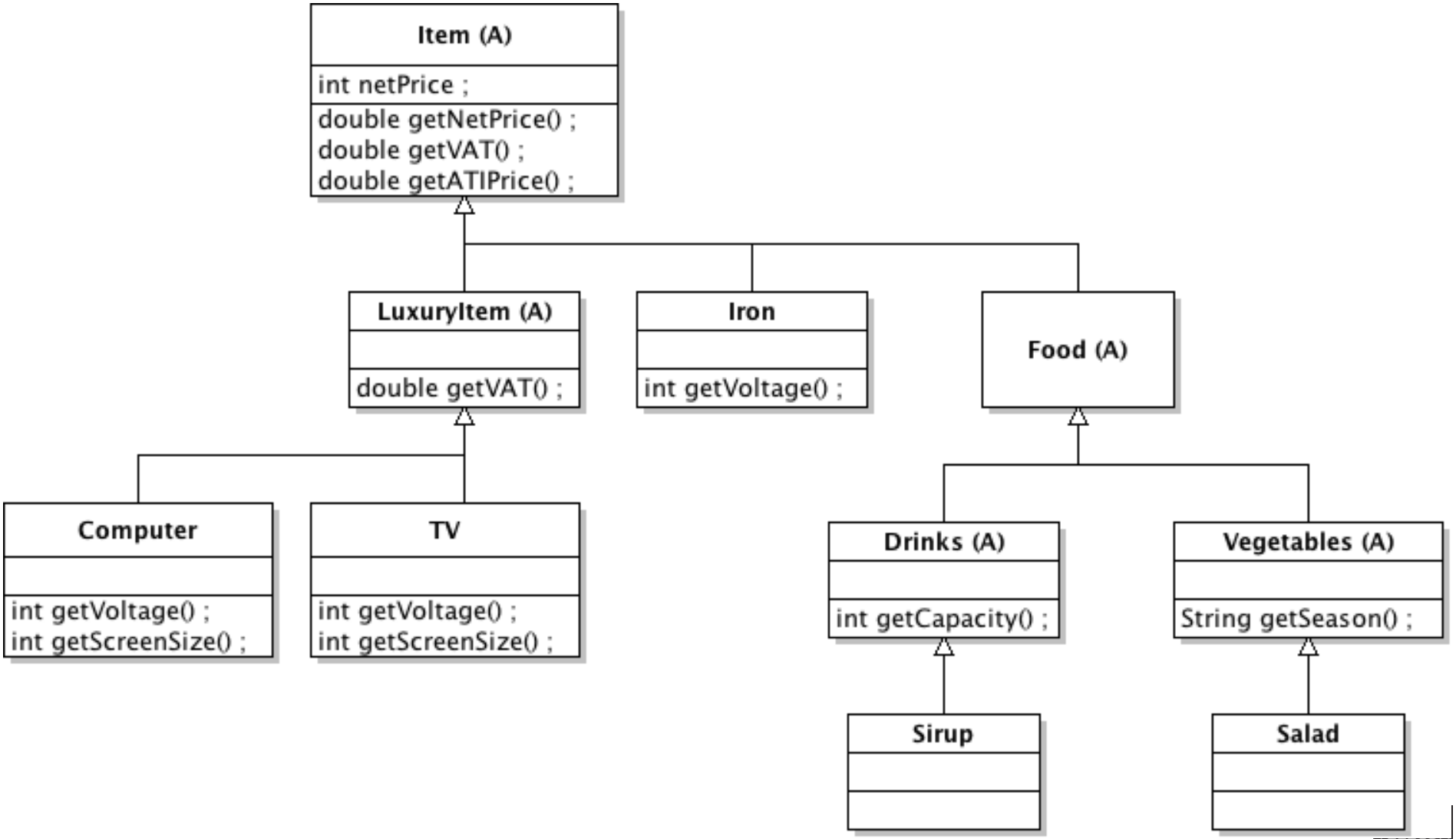
- Remarquons qu'un objet de la classe **`LuxuryItem`** sera obligatoirement une instance de la classe **`TV`** ou de la classe **`Computer`**.
  - Créer directement une instance de la classe **`LuxuryItem`** n'a pas de sens.

- Ce type de classe est appelée une classe **abstraite** par opposition aux classes concrètes que sont les classes **`TV`** et **`Computer`**.

Quelles sont les classes abstraites?



Réponse: classes marquées par (A)



Exemple avec la modélisation des articles

- Pour déclarer qu'une classe est **abstraite**, on utilise le mot clé **`abstract`**:

```
public abstract class Item {
    ...
}

public abstract class LuxuryItem extends Item {
    ...
}

public class TV extends LuxuryItem {
    ...
}
```

Exemple avec la modélisation des articles

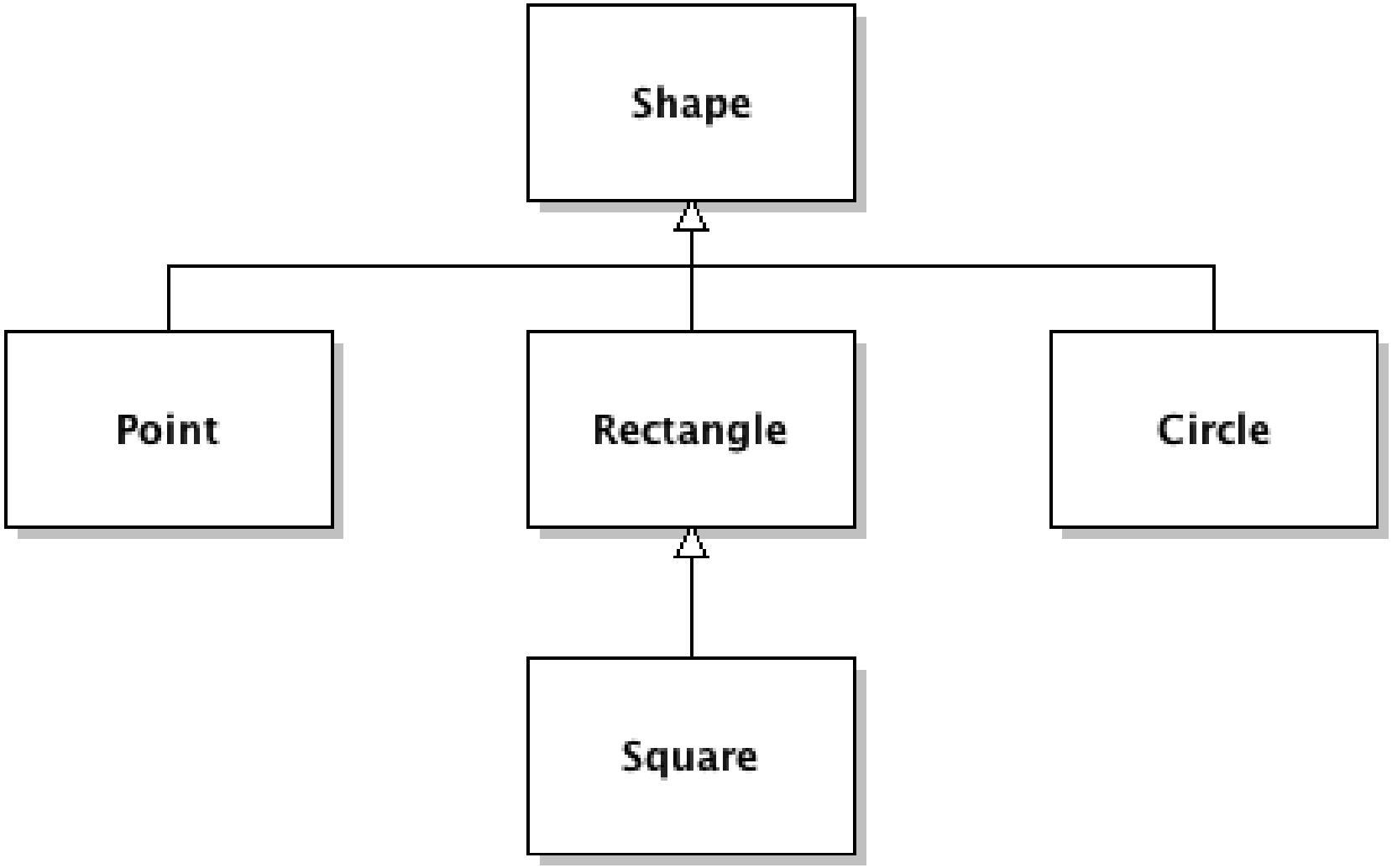
- En déclarant qu'une classe est **abstraite**, on s'interdit de créer des **instances directes** de la classe.

- Une instruction **`new Item()`** provoquera une erreur à la compilation.

- Il est quand même possible de déclarer des variables dont le type est une **classe abstraite**:

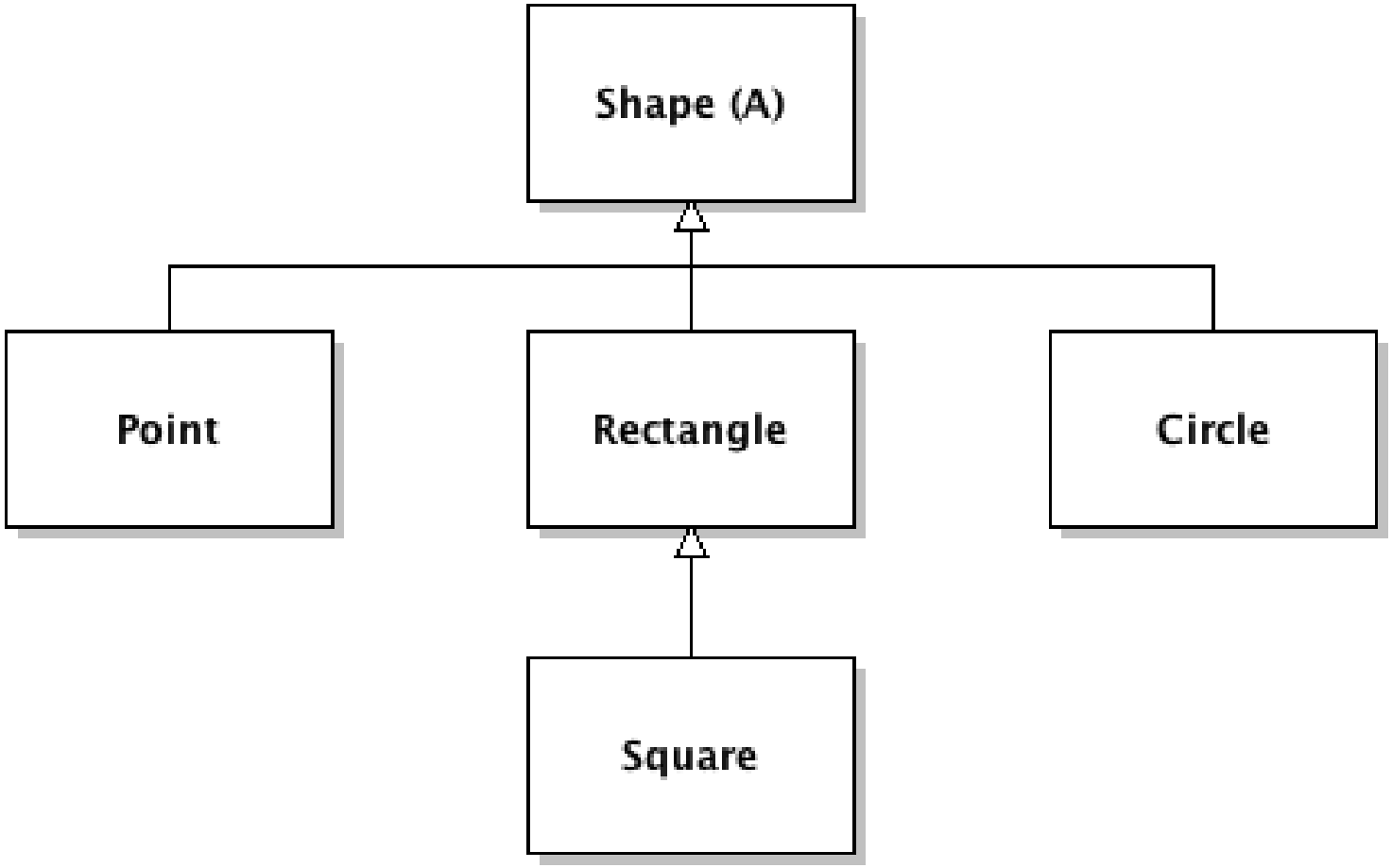
```
LuxuryItem luxuryItem = new Computer();
```

Exemple avec les figures



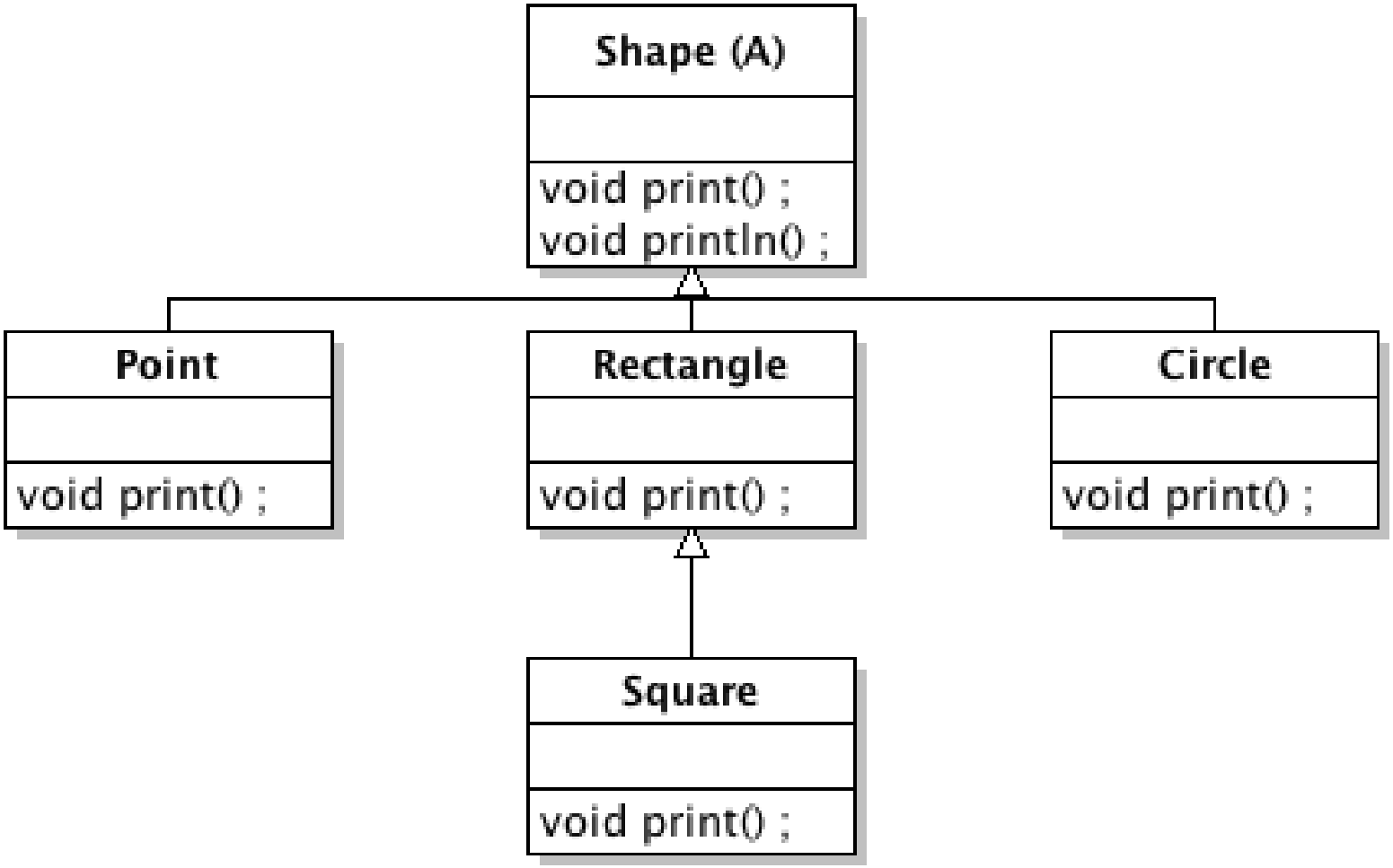
Exemple avec la modélisation des articles

- La classe **Shape** ne doit pas être instanciée. Seules les sous-classes doivent l'être. Elle doit donc être déclarée abstraite.



Méthodes abstraites

- Nous avions une méthode **`print()`** dans la classe **`Shape`** :



Méthodes abstraites

- Cette méthode **`print()`** dans la classe **`Shape`** était nécessaire pour que le compilateur sache que toutes les figures concrètes ont une méthode **`print()`**.

- Cependant, la méthode **`print()`** dans la classe **`Shape`** ne sera jamais exécutée. En effet, tous les objets ayant le type **`Shape`** seront des instances des sous-classes concrètes de **`Shape`**.

- Les instructions de la méthode **`print()`** dans la classe **`Shape`** sont donc inutiles.

- Java propose de qualifier la méthode **`print()`** dans la classe **`Shape`** d'**abstraite** avec le mot clé **`abstract`**.

Exemple avec la modélisation des articles

- Cela donne :

```
public abstract class Shape {

    public abstract void print();

    public final void println() {
        print();
        System.out.println();
    }

    ...
}
```

- Une méthode déclarée **`abstract`** est définie par son **en-tête** et n'a **pas de corps**.

Classes et méthodes abstraites

- Puisque la classe **`Shape`** est déclarée abstraite, le compilateur n'acceptera pas la création d'une instance directe de la classe **`Shape`** avec une instruction **`new Shape(...)`**.

- Comme la méthode **`print()`** est déclarée abstraite, une sous-classe de la classe Shape qui **ne donne pas une implémentation de cette méthode** devra aussi être déclarée **abstraite**.

Modélisation des articles

- Maintenant que nous connaissons les notions de classes et de méthodes abstraites, nous allons proposer une modélisation améliorée des articles d'un magasin.

- Nous commençons par la racine de l'arbre d'héritage. Ce sera la classe **`Item`** qui contiendra ce qui est commun à tous les articles.

- Cette classe sera sous-classée trois fois :
  - La classe abstraite **`EssentialItem`** modélisant des articles de première nécessité avec une TVA à 5%.
  - La classe abstraite **`OrdinaryItem`** modélisant des articles ordinaires avec une TVA à 18,5%.
  - La classe abstraite **`LuxuryItem`** modélisant des articles de luxe avec une TVA à 33%.

Revisitons la classe `Item`

```
public abstract class Item {

    private final String denomination;

    public Item(String denomination) {
        this.denomination = denomination;
    }

    public final String getDenomination() {
        return denomination;
    }

   public abstract double getNetPrice() ;

   public abstract double getVAT() ;

   public final double getATIPrice() {
      return getNetPrice() + getVAT() ;
   }
}
```

Revisitons la classe `EssentialItem`

- La classe est abstraite pour des raisons logiques : il n'y a aucun sens à créer une instance directe de cette classe.

- La classe est également abstraite car elle n'implémente pas la méthode **`double getNetPrice();`**

```
public abstract class EssentialItem extends Item {

    public EssentialItem(String denomination) {
        super(denomination); // Appel obligatoire au
                             // constructeur de Item
    }

    public final double getVAT() {
        return 0.05 * getNetPrice();
    }
}
```

Revisitons la classe `OrdinaryItem`

- Même remarques que pour la classe **`EssentialItem`**.

```
public abstract class OrdinaryItem extends Item {

    public OrdinaryItem(String denomination) {
        super(denomination);
    }

    public final double getVAT() {
        return 0.185 * getNetPrice();
    }
}
```

Revisitons la classe LuxuryItem

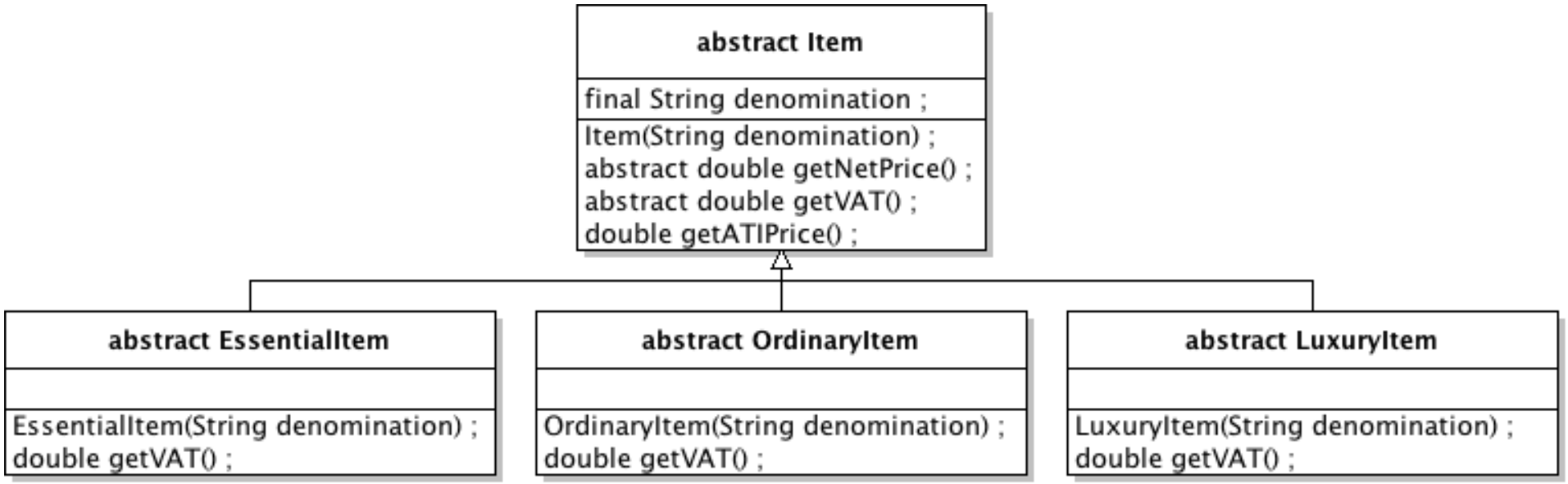
- Même remarques que pour la classe **LuxuryItem**.

```
public abstract class LuxuryItem extends Item {

    public LuxuryItem(String denomination) {
        super(denomination);
    }

    public final double getVAT() {
        return 0.33 * getNetPrice();
    }
}
```

Diagramme de classes



Modélisation des articles

- Nous venons de proposer un modèle possible pour les classes **`Item`**, **`EssentialItem`**, **`OrdinaryItem`** et **`LuxuryItem`**.

- Cette modélisation est tout à fait correcte. Mais un bon informaticien s'inquiète toujours lorsqu'il voit les mêmes instructions à différents endroits de son programme.
  - Cela peut traduire une **faiblesse** dans son modèle.

- Les méthodes **`double getVAT()`** des trois sous-classes de la classe **`Item`** sont très similaires, à l'exception du taux de TVA qu'elles utilisent.

- Pourquoi ne pas stocker ce taux de TVA dans un attribut de la classe **`Item`** ?

Un attribut pour le taux de TVA

```
public abstract class Item {

    private final String denomination;
    private final double vatRate;

    public Item(String denomination, double vatRate) {
        this.denomination = denomination;
        this.vatRate = vatRate;
    }

    public final String getDenomination() {
        return denomination;
    }

    public abstract double getNetPrice() ;

    public final double getVAT() {
        return vatRate * getNetPrice();
    }

    public final double getATIPrice() {
        return getNetPrice() + getVAT();
    }
}
```

Un attribut pour le taux de TVA

```
public abstract class EssentialItem extends Item {

    public EssentialItem(String denomination) {
        super(denomination, 0.05);
    }
}

public abstract class OrdinaryItem extends Item {

    public OrdinaryItem(String denomination) {
        super(denomination, 0.185);
    }
}

public abstract class LuxuryItem extends Item {

   public LuxuryItem(String denomination) {
      super(denomination, 0.33);
   }
}
```

Comparaison des deux modèles

- Ce nouveau modèle des classes **`Item`**, **`EssentialItem`**, **`OrdinaryItem`** et **`LuxuryItem`** n'est ni meilleur ni moins bon que le précédent modèle.
- Il évite la **duplication de code** observée dans les trois sous-classes.
- Mais il introduit une dose de complexité dans la classe racine Item.
- Chacun est libre de préférer l'un ou l'autre des deux modèles.

Incrémentalité et modularité de la modélisation

- La modélisation des articles d'un magasin peut être réalisée de manière **incrémentale** et **modulaire** :
  - Incrémentale: On n'est pas obligé de développer la totalité de l'application pour qu'elle fonctionne.
    - On peut développer et tester des parties isolées comme les télévisions puis passer à une autre catégorie d'articles.
  - Modulaire: Chaque catégorie d'article est modélisée par un ensemble de classes logiquement indépendantes des autres.
    - Cela pourrait se traduire par un package dédié.

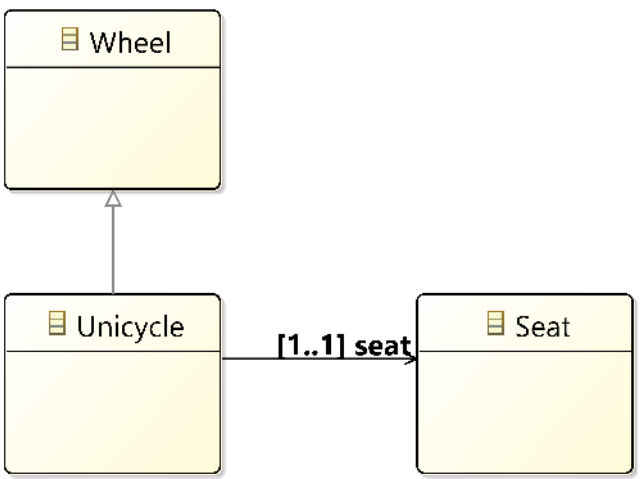
- Suppose que les classes de base et les interfaces ont été **bien conçues** :
  - Une modification **a posteriori** des classes de base peut demander de revoir **une grande partie** du code.
  - Si on renonce à ces modifications, alors la modélisation d'un nouveau type d'article sera mal programmée.

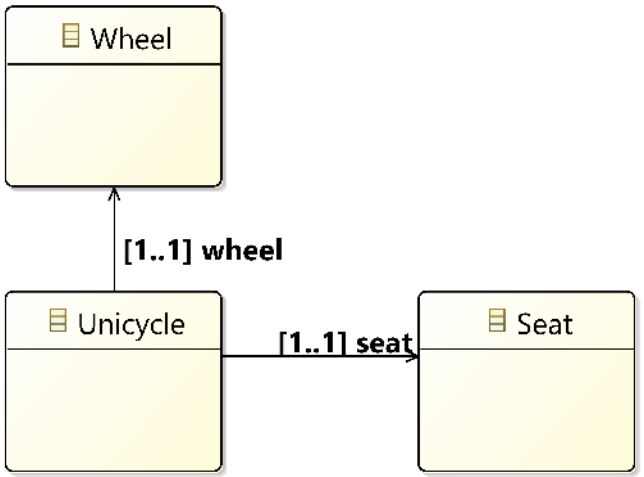
Bonne utilisation de l'héritage

- L'héritage est une notion particulièrement **complexe**.

- Il faut se rappeler de la sémantique de l'héritage:
  - Une sous-classe représente un sous-ensemble des objets de la classe mère.

- Distinguer l'héritage de la composition:





Conclusion

- L'héritage est une notion complexe ayant de nombreuses significations et posant parfois des problèmes conceptuels (héritage multiple par exemple).

- Bien modéliser les éléments d'un problème, c'est-à-dire déterminer un arbre d'héritage à la fois compréhensible, logique et efficace, requiert de la méthode et de l'expérience.

- Comme souvent en informatique, il n'y a pas de méthode absolue. Il n'y a que des approches, des bonnes pratiques et de l’expérience.