Programmation orientée objet en Java

Héritage de classes (partie 2)

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

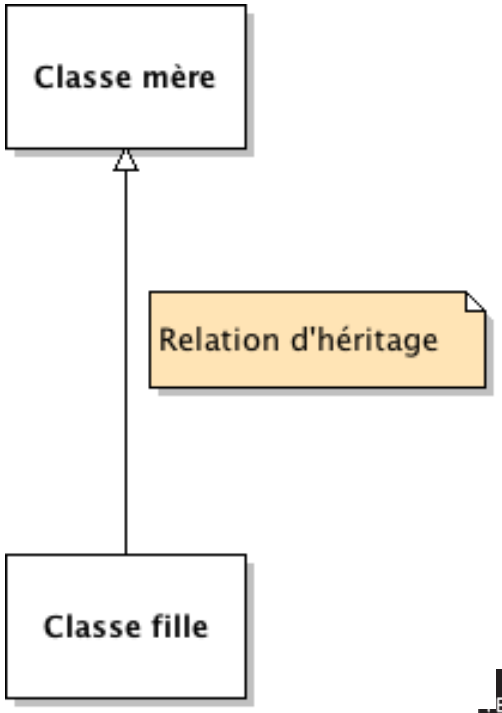
Objectifs d'apprentissage

- **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 * getNetPrice(); // 18,5%
    }

    public double getATIPrice() { // ATI = All Taxes Included
        return getNetPrice() + 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 raffinée 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

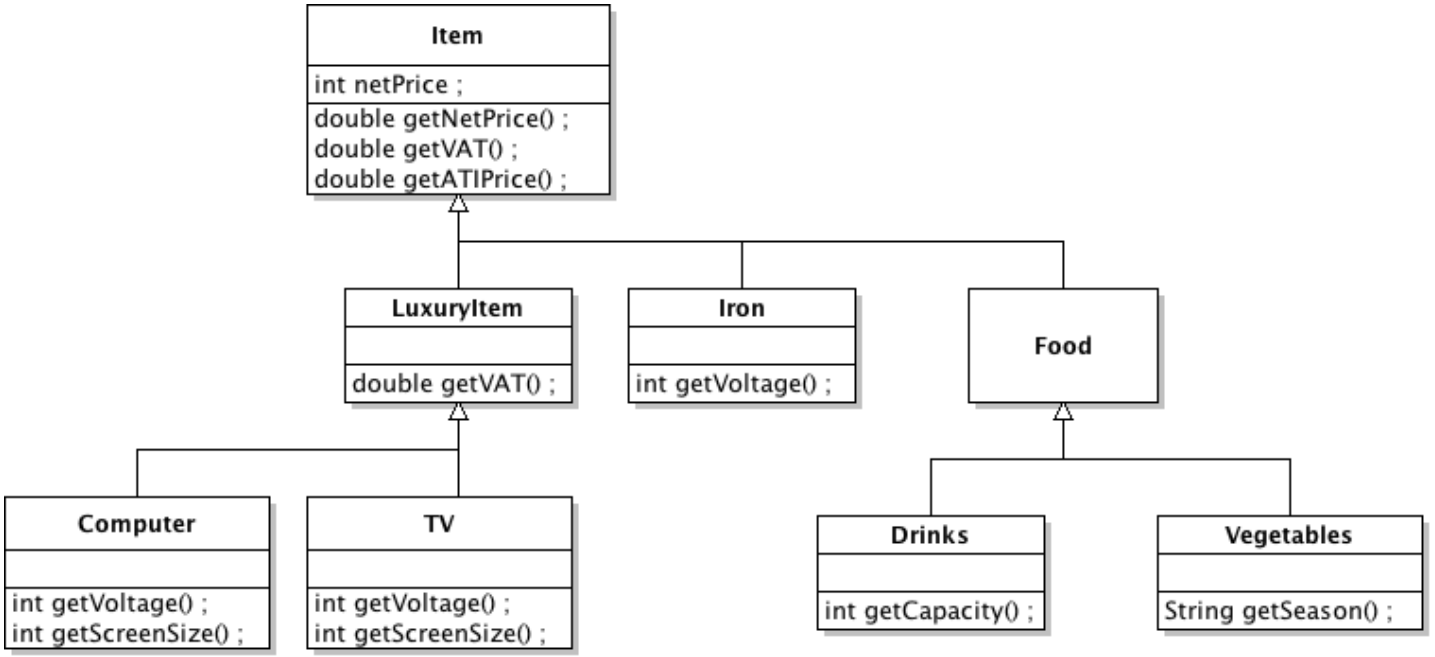
- Est une sorte de ....
- Est un genre de ....
- Est une catégorie de ....
  - Ainsi, une TV est une **sorte** d'Item.

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

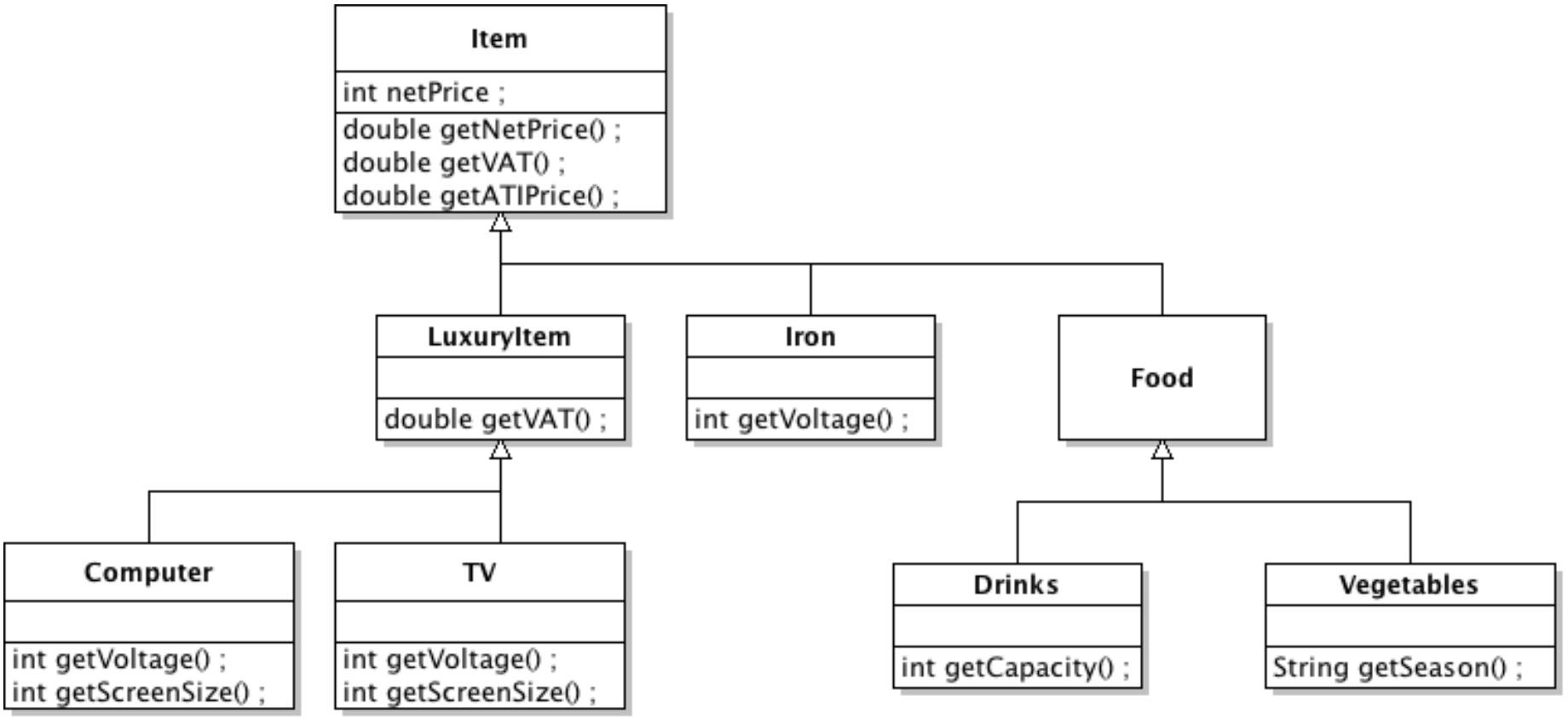
Classe abstraite : exemple des articles d'un magasin



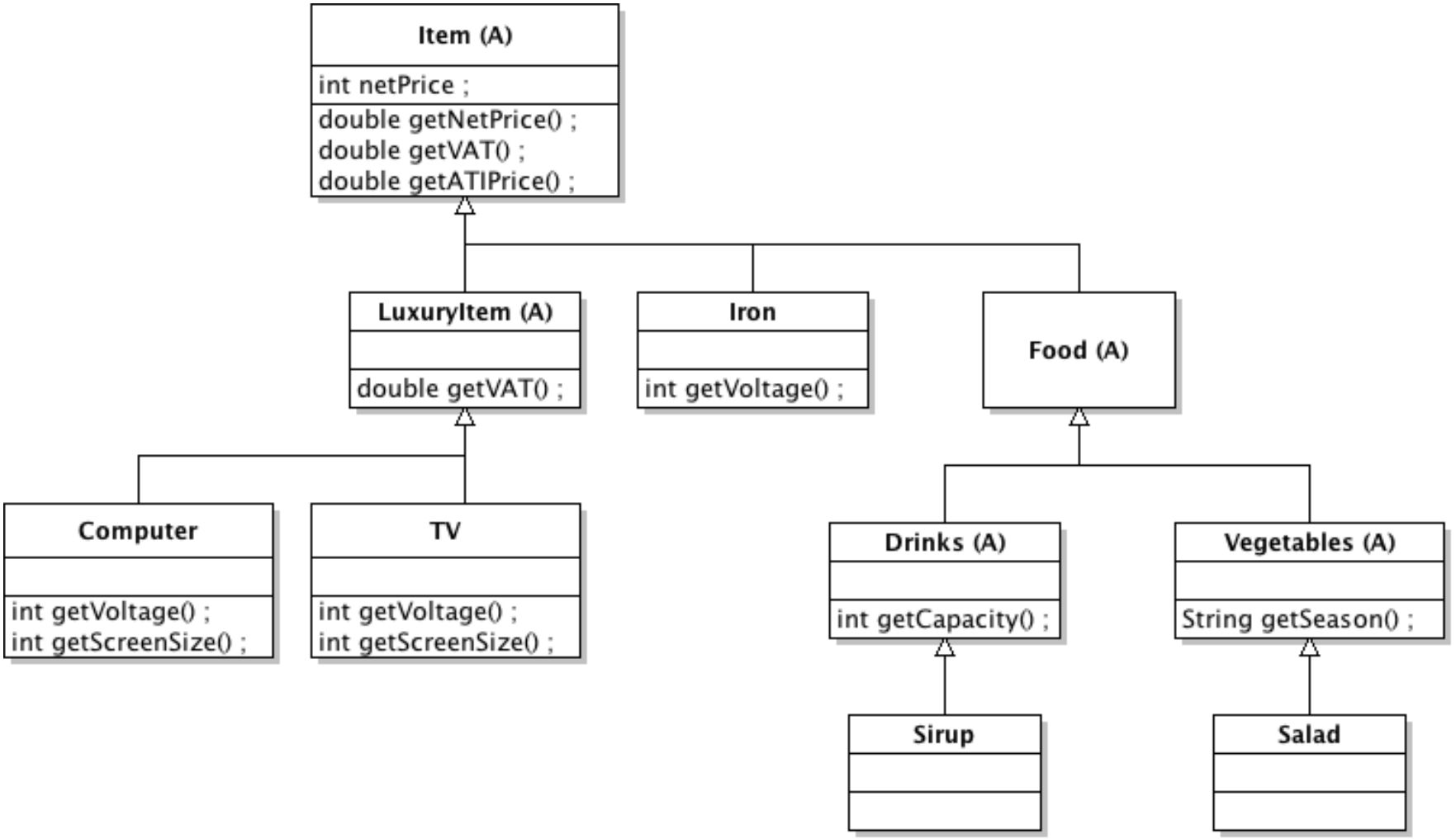
- Remarquons qu'un objet de la classe **`LuxuryItem`** sera obligatoirement une instance de la classe **`TV`** ou de la classe **`Computer`**.
  - Créer directement une instance de la classe **`LuxuryItem`** n'a pas de sens.

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

Exercice : quelles sont les classes abstraites?



Réponse : classes marquées par `(A)`



Exemple avec la modélisation des articles de magasin

- 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 {
    ...
}
```

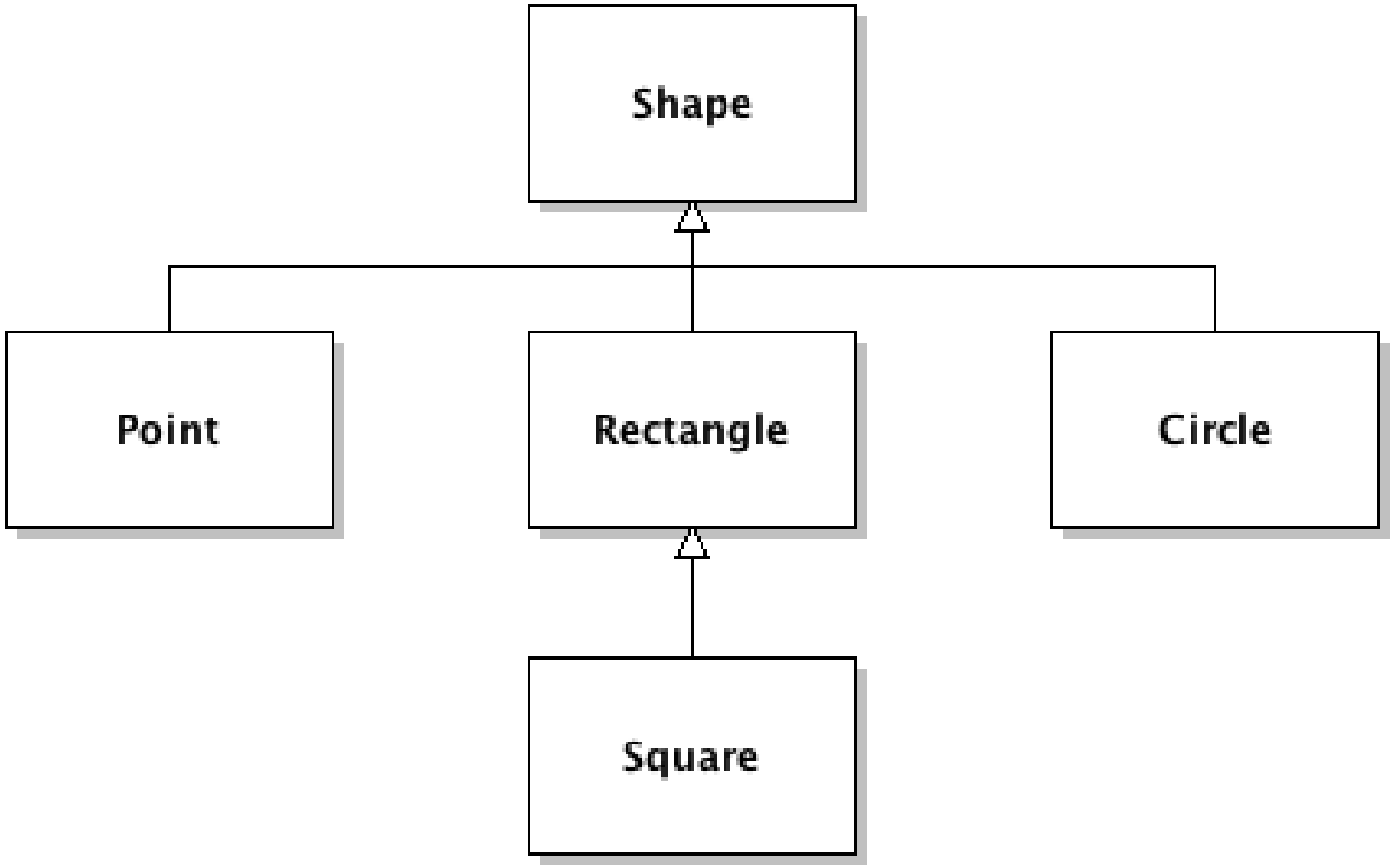
Exemple avec la modélisation des articles de magasin

- 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();
```

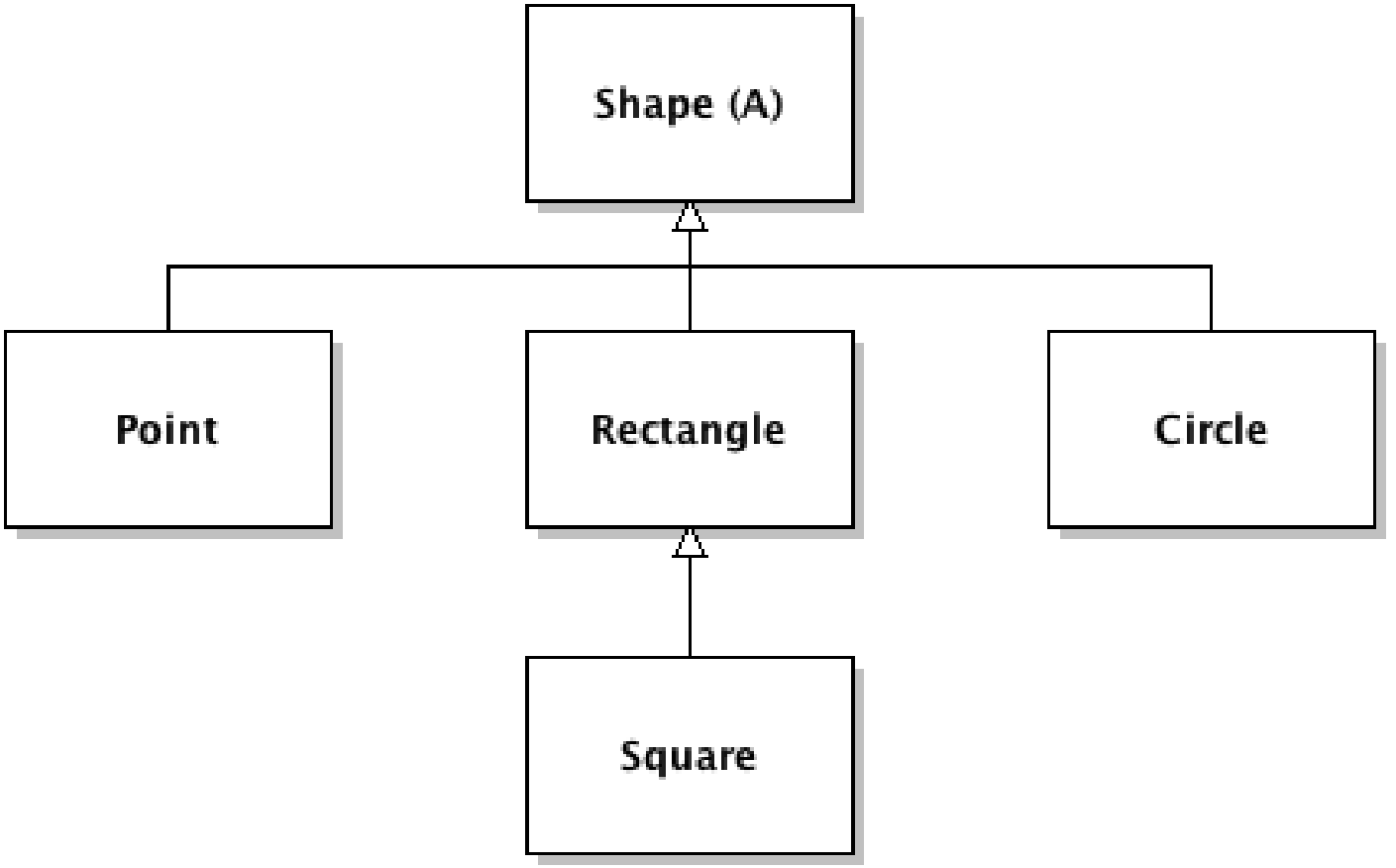
- Ainsi, seules les méthodes de la classe **`LuxuryItem`** pourront être utilisées par la suite, et non pas celles de la classe **`Computer`**.

Exemple avec les figures : quelles sont les classes abstraites?



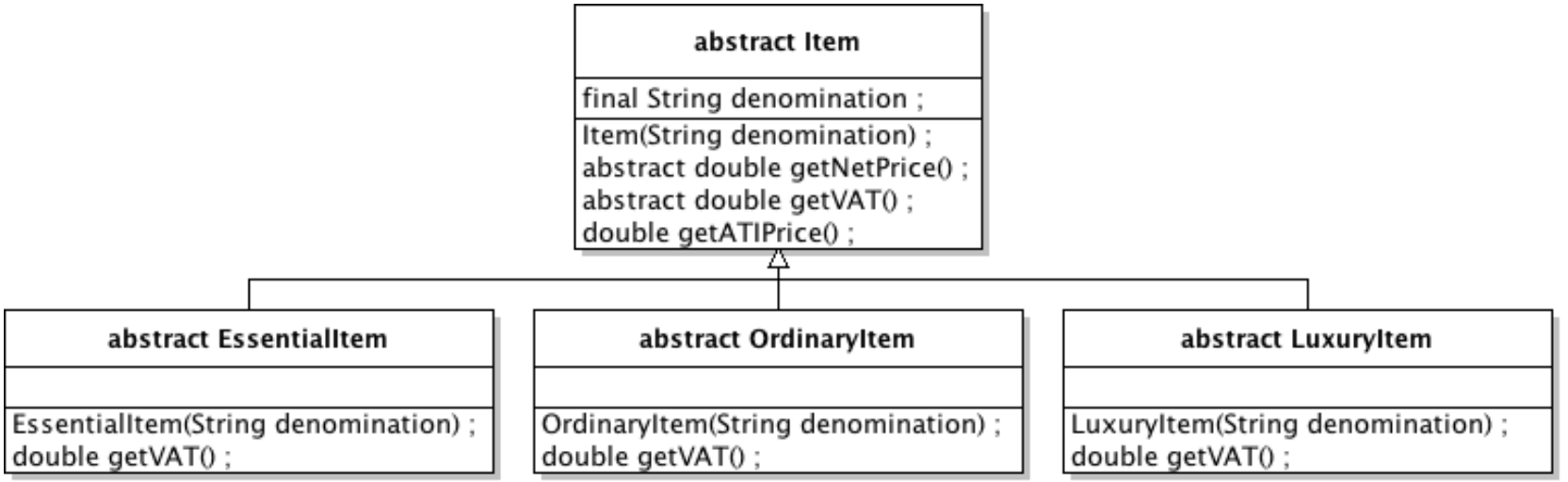
Réponse : classes marquées par (A)

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



Modélisation améliorée des articles grâce aux classes abstraites

- On sous-classe la classe **`Item`** 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%**.



Méthode abstraite

```
public class Item {

    private double netPrice;

    public Item(double netPrice) {
        this.netPrice = netPrice;
    }

    public double getNetPrice() {
        return netPrice;
    }

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

    public double getATIPrice() { // ATI: All Taxes Included
        return getNetPrice() + getVAT();
    }
}
```

```
public abstract class Item {

    private final double netPrice;

    public Item(double netPrice) {
        this.netPrice = netPrice;
    }

    public final double getNetPrice() {
        return netPrice;
    }

    public abstract double getVAT();

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

- La méthode abstraite **`getVAT()`** est nécessaire pour la méthode **`getATIPrice()`.**
- Comme la taxe n'est connue que par les sous-classes, on déclare **`getVAT()`** **abstraite** et on ne fourni **pas de corps de méthode**.

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.
- Par contre, la taxe est connue et la classe peut donc fournit une méthode concrète pour **`getVAT()`**.
- Cette méthode est déclarée **`final`** car toute sous-classe devra calculer la taxe de la **même manière**.
- Toute classe concrète devra fournir une méthode concrète **`getVAT()`** soit dans la classe elle-même ou dans l'une de ses classes mères.

```
public abstract class EssentialItem extends Item {

    public EssentialItem(double netPrice) {
        super(netPrice); // Appel obligatoire au constructeur de Item
    }

    @Override
    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(double netPrice) {
       super(netPrice);
    }

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

Revisitons la classe `LuxuryItem`

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

```
public abstract class LuxuryItem extends Item {

    public LuxuryItem(double netPrice) {
        super(netPrice);
    }

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

Exemple de classe concrète de `LuxuryItem`

```
public class Computer extends LuxuryItem {

    private final int voltage;

    private final String model;

    public Computer(double netPrice,
                    int voltage,
                    String model) {
        super(netPrice);

        this.voltage = voltage;
        this.model = model;
    }
}
```

Modélisation des articles

- Nous venons de proposer un modèle **amélioré** 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 le modèle.

- Les méthodes **`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 double vatRate;

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

    public double getNetPrice() {
        return netPrice;
    }

    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(double netPrice) {
        super(netPrice, 0.05);
    }
}

public abstract class OrdinaryItem extends Item {

    public OrdinaryItem(double netPrice) {
        super(netPrice, 0.185);
    }
}

public abstract class LuxuryItem extends Item {

   public LuxuryItem(double netPrice) {
      super(netPrice, 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.
- Cependant, 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.

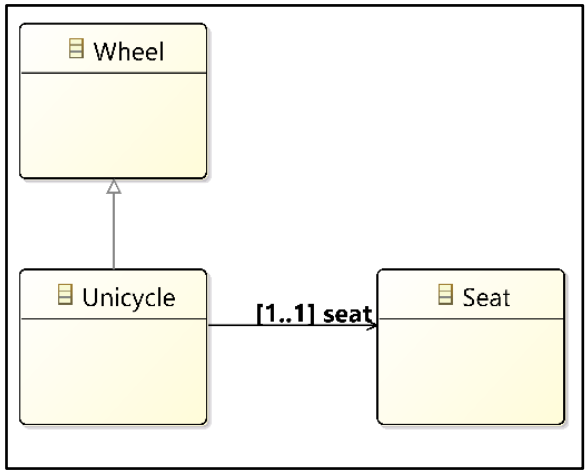
Incrémentalité et modularité de la modélisation

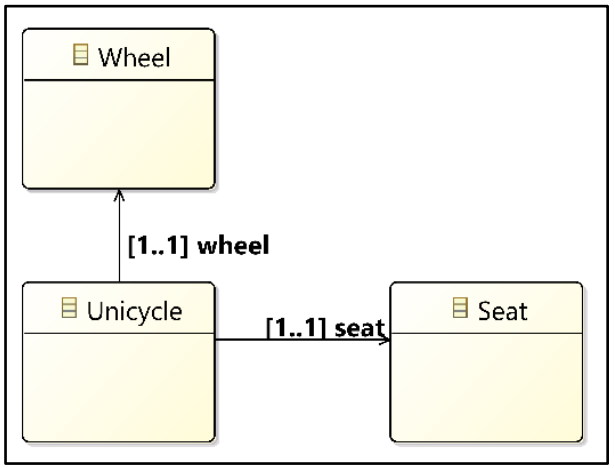
- La modélisation des articles d'un magasin peut être réalisée de manière **incrémentale** et **modulaire** :
  - **Incrémentale** : on n'est pas obligé de développer la totalité de l'application pour qu'elle fonctionne.
    - On peut développer et tester des parties isolées comme les télévisions puis 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.

- Cela 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.

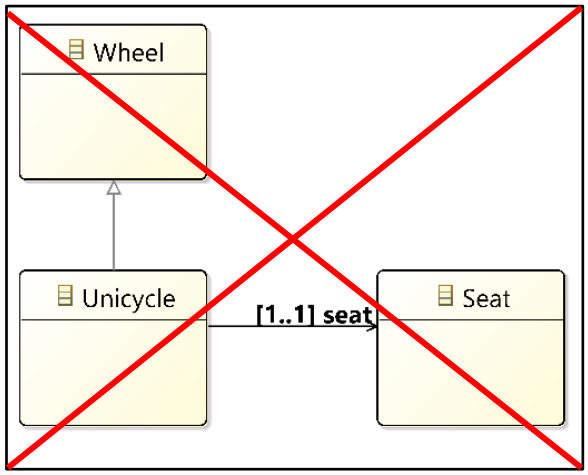
Bonne utilisation de l'héritage

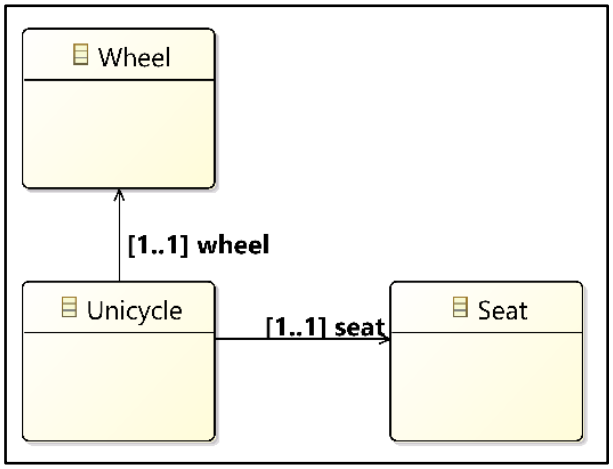
- L'héritage est une notion pas toujours facile à utiliser…
- Il faut se rappeler de la sémantique de l'héritage :
  - Une sous-classe représente un sous-ensemble des objets de la classe mère.
- Parmi les diagrammes de classe suivants, quel est le meilleur modèle pour l'unicycle ?





Bonne utilisation de l'héritage





- Un unicycle **possède** une roue mais **n'est pas** une roue.
- Il faut distinguer l'**héritage** de la **composition**.

Le principe responsabilité unique

- En programmation orientée objet, Robert C. Martin exprime le **principe de responsabilité unique** comme suit :
  - « une classe ne doit changer que pour une seule raison » (a class should have only one reason to change).

- Une classe ne devrait avoir qu'une seule responsabilité, clairement identifiée par le **nom** de la classe.

Exemple : responsabilité unique ou pas?

```
public class Computer extends LuxuryItem {

    private final int voltage;

    private final String brand;

    public Computer(double netPrice,
                    int voltage,
                    String brand) {
        super(netPrice);

        this.voltage = voltage;
        this.brand = brand;
    }

    public void writeToFile(String fileName) {
        …
    }
}
```

Exemple : responsabilité unique ou pas?

- La sauvegarde des objets, que ce soit dans un fichier ou une base de données est une **autre responsabilité** : **persistance des données**.
- On va donc utiliser une classe **dédiée** à cette responsabilité.
  - Il y aura un TP sur ce sujet…
- Toutes les informations techniques du système de stockage de données utilisé seront gérées par cette classe.
- Si on change de type stockage, on ne changera que la classe de persistance sans devoir modifier les classes du modèle de données.
- Exemple :

```
Computer myComputer = new Computer(700,0, 12,0, HP);
EntityPersistenceManager myPersistenceManager = …;
myPersistenceManager.persist(myComputer); // Sauvegarde des
données
```

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.
- Mais on peut bénéficier de l'expérience des autres en consultant quelques bonnes pratiques éprouvées.
  - Par exemple https://www.geeksforgeeks.org/best-practices-of-object-oriented-programming-oop/.