

Documentation et UML

Arnaud Labourel

1 Documentation

Dans tout développement informatique, il est souvent primordial de documenter correctement son code afin qu'il puisse être utilisable par d'autres développeurs. L'idée de la documentation est de détailler certains points (par exemple le comportement d'une méthode pour des cas spécifiques pour des valeurs précises de paramètres) et donner un résumé du fonctionnement des éléments du code de sorte qu'une personne puisse réutiliser les éléments documentés sans regarder le code (dont il n'a pas forcément l'accès). La documentation peut aussi être utile pour le développeur du code lui-même, car elle permet de donner des spécifications sur le comportement que doit avoir le code. Néanmoins, cette utilisation peut être contre-productive, car source de duplication d'informations. En effet, le comportement d'une méthode peut par exemple changer lorsque le développeur réécrit son code. Par conséquent, le développeur doit faire deux fois les changements (une fois dans son code et une autre fois dans sa documentation). C'est pour cela que généralement, il ne faut documenter son code que si cela a une utilité directe. Normalement, le code doit se suffire à lui-même, c'est-à-dire qu'il doit être directement lisible sans ajout de documentation ou de commentaires. Bien sûr, comme toutes les règles en développement logiciel, cette règle n'est pas absolue et autorise donc des exceptions.

1.1 Présentation de Javadoc

L'outil de base en Java pour documenter son code s'appelle Javadoc. La Javadoc est fournie avec le JDK afin de permettre la génération d'une documentation technique directement à partir du code source (et donc des fichiers `.java`). L'intérêt de ce système est de conserver dans le même fichier le code source et les éléments de la documentation qui lui sont associés. Cet outil utilise des commentaires dans un format spécifique qu'on va détailler. Il est notamment utilisé pour la génération de la documentation du JDK. Cette documentation contient :

- une description détaillée pour chaque classe et ses membres `public` et `protected`;
- un ensemble de listes (liste des classes, hiérarchie des classes, liste des éléments *deprecated* et un index général);
- des références croisées et une navigation entre ces différents éléments.

Les commentaires pour Javadoc suivent des règles précises. Le format de ces commentaires commence par `/**` et se termine par `*/`. Ils peuvent contenir un texte libre en HTML et des *tags* (mots-clés définis par Javadoc) précédés par `@`. Le commentaire peut être sur plusieurs lignes. Dans chaque ligne, les espaces qui précèdent le premier caractère `*` de la ligne du commentaire ainsi que le caractère lui-même sont ignorés. Ceci permet d'utiliser le caractère `*` pour aligner le contenu du commentaire.

Le format général de ces commentaires est le suivant :

```
1 /**
2  * Description
3  *
4  * @tag1
5  * @tag2
6  */
```

Le texte du commentaire doit être au format HTML : les balises HTML peuvent donc être utilisées pour enrichir

le formatage de la documentation. Les *tags* prédéfinis par Javadoc permettent de fournir des informations plus précises sur des composants particuliers de l'élément (auteurs, paramètres, valeurs de retour, ...). Ces *tags* sont définis pour un ou plusieurs types d'éléments. Il existe deux types de tags :

- *Inline tag* de la forme `{@tag}` et
- *Block tag* de la forme `@tag`.

1.2 Tags Javadoc

Il existe de nombreux *tags* définis par le format de la javadoc. Les principaux sont les suivants :

Tag	Description
<code>@author</code>	pour préciser l'auteur de la fonctionnalité
<code>@deprecated</code>	indique que l'attribut, la méthode ou la classe est dépréciée
<code>@return</code>	pour décrire la valeur de retour d'une méthode
<code>@param p</code>	pour décrire un paramètre <code>p</code> d'une méthode
<code>{@code literal}</code>	formate <code>literal</code> en code
<code>{@link reference}</code>	permet de créer un lien
<code>@exception e</code>	indique une exception <code>e</code> qui peut être levée par la méthode
<code>@since number</code>	permet de préciser depuis quelle version l'élément a été ajouté

1.2.1 Le tag `@author`

Le tag `@author` permet de préciser le ou les auteurs d'un élément. La syntaxe de ce tag est la suivante :

`@author nom des auteurs/autrices`

Exemple d'utilisation:

```
1 /**
2  * @author Paul Calcul, Jean-Michel Bruitages
3  */
```

1.2.2 Le tag `@deprecated`

Le tag `@deprecated` permet de préciser qu'un élément ne devrait plus être utilisé même s'il fonctionne toujours : il permet donc de donner des précisions sur un élément déprécié (*deprecated*). La syntaxe de ce tag est la suivante :

`@deprecated texte`

Il est recommandé de préciser depuis quelle version l'élément est déprécié et de fournir dans le texte libre une description de la solution de remplacement, si elle existe, ainsi qu'un lien vers une entité de substitution. Exemple d'utilisation pour le constructeur de `Float` :

```
1 /**
2  * Constructs a newly allocated {@code Float} object that
```

```

3      * represents the floating-point value of type {@code float}
4      * represented by the string. The string is converted to a
5      * {@code float} value as if by the {@code valueOf} method.
6      *
7      * @param    s    a string to be converted to a {@code Float}.
8      * @throws    NumberFormatException if the string does not contain a
9      *           parsable number.
10     *
11     * @deprecated
12     * It is rarely appropriate to use this constructor.
13     * Use {@link #parseFloat(String)} to convert a string to a
14     * {@code float} primitive, or use {@link #valueOf(String)}
15     * to convert a string to a {@code Float} object.
16     */
17     public Float(String s) throws NumberFormatException {
18         /* ... */
19     }

```

On peut voir dans l'exemple ci-dessus que le constructeur de `Float` est déprécié (donc normalement son utilisation ne sera plus possible dans une version futures). La documentation conseille d'utiliser d'autres méthodes à la place du constructeur.

1.2.3 Le tag `@throws`

Le tag `throws` permet de documenter une exception levée par la méthode ou le constructeur décrit par le commentaire. Il est possible d'ajouter du texte afin de décrire les conditions de levée de l'exception. La syntaxe d'utilisation de ce tag est la suivante :

`@throws nom_exception description`

Le tag est suivi du nom de l'exception puis d'une courte description des raisons de la levée de cette dernière. Il faut utiliser autant de tag `@throws` qu'il y a d'exceptions. Ce tag doit être utilisé uniquement pour une méthode ou un constructeur. Exemple d'utilisation pour la méthode `add` de `List` :

```

1      /**
2      * Appends the specified element to the end of this list (optional
3      * operation).
4      *
5      * <p>Lists that support this operation may place limitations on what
6      * elements may be added to this list. In particular, some
7      * lists will refuse to add null elements, and others will impose
8      * restrictions on the type of elements that may be added. List
9      * classes should clearly specify in their documentation any restrictions
10     * on what elements may be added.
11     *
12     * @param e element to be appended to this list
13     * @return {@code true} (as specified by {@link Collection#add})
14     * @throws UnsupportedOperationException if the {@code add} operation
15     *        is not supported by this list
16     * @throws ClassCastException if the class of the specified element
17     *        prevents it from being added to this list
18     * @throws NullPointerException if the specified element is null and this
19     *        list does not permit null elements

```

```

20     * @throws IllegalArgumentException if some property of this element
21     *         prevents it from being added to this list
22     */
23     void add(int index, E element);

```

Dans l'exemple ci-dessous, il y a la description des différentes exceptions pouvant être levées par la méthode `add` de `List`

1.2.4 Le tag `@param`

Le tag `@param` permet de documenter un paramètre d'une méthode ou d'un constructeur. La syntaxe de ce tag est la suivante :

```
@param name texte de description
```

Ce *tag* est suivi du nom du paramètre (sans le type) puis d'une courte description de ce dernier. Il faut utiliser autant de *tag* `@param` que de paramètres dans la signature de l'entité concernée. Par convention les paramètres doivent être décrits dans leur ordre dans la signature de la méthode ou du constructeur. Exemple d'utilisation pour la méthode `of` de `List` :

```

1     /**
2     * Returns an unmodifiable list containing nine elements.
3     *
4     * See <a href="#unmodifiable">Unmodifiable Lists</a> for details.
5     *
6     * @param <E> the {@code List}'s element type
7     * @param e1 the first element
8     * @param e2 the second element
9     * @param e3 the third element
10    * @param e4 the fourth element
11    * @param e5 the fifth element
12    * @param e6 the sixth element
13    * @param e7 the seventh element
14    * @param e8 the eighth element
15    * @param e9 the ninth element
16    * @return a {@code List} containing the specified elements
17    * @throws NullPointerException if an element is {@code null}
18    *
19    * @since 9
20    */
21    static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9){
22        /* ... */
23    }

```

On peut observer que dans l'exemple ci-dessus les 9 paramètres sont bien documentés dans l'ordre de la signature de la méthode.

1.2.5 Le tag `@return`

Le tag `@return` permet de fournir une description de la valeur de retour d'une méthode qui en possède une. La syntaxe de ce tag est la suivante :

`@return` description de la valeur de retour

Il ne peut y avoir qu'un seul tag `@return` par commentaire : il doit être utilisé uniquement pour un élément de type méthode qui renvoie une valeur.

Exemple d'utilisation pour la méthode `size` de `List` :

```
1  /**
2   * Returns the number of elements in this list. If this list contains
3   * more than {@code Integer.MAX_VALUE} elements, returns
4   * {@code Integer.MAX_VALUE}.
5   *
6   * @return the number of elements in this list
7   */
8  int size();
```

1.2.6 Le tag `{@link}`

Ce *tag inline* permet de créer un lien vers un autre élément de la documentation. La syntaxe de ce tag est la suivante : `{@link package.class#membre texte}`

1.2.7 Le tag `{@code}`

Ce *tag inline* permet d'afficher un texte dans des tags `<code> ... </code>` qui ne sera pas interprété comme de l'HTML. La syntaxe de ce tag est la suivante : `{@code texte}`

1.2.8 Le tag `@since`

Le *tag @since* permet de préciser un numéro de version de la classe ou de l'interface à partir de laquelle l'élément décrit est disponible. Ce *tag* peut être utilisé avec tous les éléments. La syntaxe de ce *tag* est la suivante :

`@since texte`

Exemple : `@since 2.0`

1.2.9 Documentation des *packages*

En plus de la documentation contenue dans les fichiers de classes et d'interfaces, il est possible de rajouter dans chaque package un fichier `package-info.java` qui sera utilisé par Javadoc pour produire la documentation du *package*. Par exemple, le fichier `package-info.java` du *package applet* d'`awt` est le suivant :

```
1  /**
2   * Provides the classes necessary to create an
3   * applet and the classes an applet uses
4   * to communicate with its applet context.
5   * <p>
6   * The applet framework involves two entities:
7   * the applet and the applet context.
8   * An applet is an embeddable window (see the
9   * {@link java.awt.Panel} class) with a few extra
10  * methods that the applet context can use to
11  * initialize, start, and stop the applet.
```

```
12  *
13  * @since 1.0
14  * @see java.awt
15  */
16  package java.lang.applet;
```

1.3 Génération de documentation

Pour générer la documentation

Pour générer la documentation, il faut donc invoquer l’outil Javadoc. Javadoc recrée à chaque appel la totalité de la documentation. Par défaut, Javadoc la documentation au format HTML, mais il est possible de générer la documentation dans d’autres formats comme du RTF ou du XML.

Pour appeler l’outil Javadoc, on peut passer par le terminal en appelant la commande `javadoc package1 package2 ...` où `package1`, `package2`, ... sont les noms des *packages* dont on souhaite générer la documentation. Sous IntelliJ IDEA, il faut passer par le menu **Tools** puis **generate Javadoc** en choisissant le répertoire dans lequel vous souhaitez sauvegarder la documentation. Vous pouvez aussi utiliser le moteur de production **gradle** soit par le terminal en appelant la commande `gradle javadoc` à la racine du projet ou bien en passant par le menu dédié d’IntelliJ IDEA (situé à droite) avec la tâche `javadoc` dans `documentation`. Si vous utilisez Gradle pour générer la documentation, celle-ci sera placée dans le répertoire `build/doc/javadoc` du projet.

La génération de la documentation crée de nombreux fichiers et des répertoires pour structurer la documentation au format HTML. Les fichiers les plus importants sont :

- Un fichier HTML par classe ou interface qui contient le détail de chaque élément de la classe ou de l’interface. Le fichier a le nom de la classe/interface et se trouve dans le répertoire de la classe ou de l’interface.
- Un fichier HTML par package qui contient un résumé du contenu du package. Le fichier se trouve dans le répertoire du package et porte le nom `package-summary.html`.
- Un fichier `index.html` qui est la page principale de la documentation

2 UML (Unified Modeling Language)

L’UML (Unified Modeling Language) est une norme émanant de l’OMG (Object Management Group, voir <https://www.omg.org>). L’OMG, organisme à but non lucratif, a été créé en 1989 à l’initiative de grandes sociétés. Son rôle est de promouvoir des standards qui garantissent l’interopérabilité entre applications orientées objet, développées sur des réseaux hétérogènes. Actuellement, nous en sommes à la version 2.5.1 de la norme et sa spécification complète est disponible au lien suivant : <https://www.omg.org/spec/UML/2.5.1/PDF>. L’UML est un langage formel défini par un méta-modèle (modèle d’un langage de modélisation). Il permet en autres d’exprimer et d’élaborer des modèles objet, indépendamment de tout langage de programmation. C’est un support de communication performant qui, par ses représentations graphiques, permet de faciliter la représentation et la compréhension de solutions objets. UML, en tant que cadre méthodologique pour une analyse objet permet de représenter un système selon différentes vues complémentaires : les diagrammes.

2.1 Types de diagramme de classes

Un diagramme UML est une représentation graphique, qui s'intéresse à un aspect précis du modèle ; c'est une vue du modèle. Combinés, les différents types de diagrammes UML offrent une vue complète des aspects statiques et dynamiques d'un système. On dénombre 13 types de diagrammes différents permettant de représenter le cahier des charges du projet, les classes et la manière dont elles s'agencent et interagissent entre elles. Dans cette optique, dans ce cours, nous ne détaillerons que quelques types de diagrammes. Voici tout de même une liste de tous les diagrammes disponibles regroupés dans trois grandes familles et leur utilité.

2.1.1 Les diagrammes structurels ou statiques

- Les diagrammes de classes : ils représentent les classes qui vont constituer le système.
- Les diagrammes d'objets : ils permettent de représenter les instances de classes (objets) utilisées dans le système.
- Les diagrammes de composants : ils modélisent les composants d'un système dans la durée (par composant on entend composant physique ou logique).
- Les diagrammes de déploiement : ils permettent de représenter le déploiement du système sur des éléments matériels (ordinateurs, périphériques, etc.).
- Les diagrammes des paquetages : un paquetage est un conteneur logique permettant de regrouper et d'organiser les éléments dans le modèle UML. Le Diagramme de paquetage sert à représenter les dépendances entre paquetages.
- Les diagrammes de structure composite : ils décrivent la structure interne des relations entre les composants d'une classe.

2.1.2 Les diagrammes comportementaux

- Les diagrammes des cas d'utilisation : ils décrivent la fonctionnalité du système qu'un acteur (intervenant extérieur du système) doit exécuter pour obtenir un certain résultat.
- Les diagrammes d'états : ils permettent de décrire sous forme de machine à états finis le comportement du système ou de ses composants.
- Les diagrammes d'activité : ils permettent de décrire sous forme de flux ou d'enchaînement d'activités le comportement du système ou de ses composants.

2.1.3 Les diagrammes d'interaction

- Les diagrammes de séquence : ils indiquent l'interaction entre plusieurs partenaires de communication.
- Les diagrammes de communication : anciennement appelé diagramme de collaboration, il est une sorte de diagramme de séquence simplifié où l'on va se concentrer sur les échanges de messages entre objets.
- Les diagrammes d'interaction globale : ils permettent de décrire comment les scénarios identifiés dans le diagramme de séquences peuvent s'enchaîner.
- Les diagrammes de temps : il permet de décrire les variations d'une donnée au cours dutemps.

2.2 PlantUML

Afin de produire les diagrammes que nous verrons dans le cours, nous allons utiliser l'outil *PlantUML* (lien : <https://plantuml.com/fr/>). PlantUML est un outil *open-source* qui permet de générer des diagrammes à partir d'un fichier texte. Les diagrammes UML supportés sont les suivants :

- diagrammes de séquence
- diagrammes de cas d'utilisation
- diagrammes de classes
- diagrammes d'objet
- diagrammes d'activité
- diagrammes de composant
- diagrammes de déploiement
- diagrammes d'état
- diagrammes de temps

Nous allons nous concentrer sur les diagrammes de classes et de séquences.

2.3 Diagramme de classes

2.3.1 Représentation d'une classe

Le diagramme de classes permet d'exprimer la structure d'un projet en termes de classes et de relations entre ces classes et donc d'effectuer une réflexion sur l'architecture logicielle. Un diagramme de classes va pouvoir comporter tous les éléments de la programmation orienté objet. Les classes seront représentées par des rectangles composés de trois parties :

- le premier compartiment représente le nom de la classe (pour les interfaces et les classes abstraites, ce nom sera noté en italique et en plus, pour les interfaces, il sera précédé de la notation **I** ou "interface").
- le 2ème compartiment représente les attributs de la classe
- le 3ème compartiment représente les méthodes de la classe

La visibilité des attributs et des méthodes est indiquée à l'aide d'un caractère précédant le nom de l'attribut ou de la méthode :

- **public** (mot-clé **public**), symbole \circ ou $+$: l'élément est visible depuis toutes les classes.
- **protégé** (mot-clé **protected**), symbole \diamond ou $\#$: l'élément est visible pour les sous-classes de la classe et les classes du même *package*.
- **paquetage privé** (mot-clé aucun : visibilité par défaut lorsque aucune visibilité n'est indiquée), symbole \triangle ou \sim : l'élément est visible pour les classes du même *package*.
- **privé** (mot-clé **private**), symbole \diamond ou $-$: l'élément n'est visible que dans la classe dans laquelle il est défini.

La syntaxe pour les éléments d'une classe est la suivante :

- **Syntaxe pour les attributs d'une classe** (entre accolades, les mentions optionnelles) :
<visibilité> <nom_attribut> : <type> [[<multiplicité>]] [= <valeur_par_défaut>]

On peut représenter les attributs de classe (*static* en Java) en soulignant la déclaration de l'attribut.

Exemples :

— '- i : int' pour un attribut privé

— '+ point : Point2D' pour un attribut public

— `integers : int[*]` pour un tableau d'entiers de taille quelconque et à la visibilité non définie

— `# points : Point2D[4]` pour un tableau de 4 points exactement et à la visibilité protégée

— `~ x : float = 0.0` pour un nombre à virgule initialisé à 0 et à la visibilité paquetage privé

— **Syntaxe pour les méthodes ou constructeurs d'une classe** (entre accolades, les mentions optionnelles) :

<visibilité> <nom_méthode> ([<paramètre_1>, ... , <paramètre_N>]) : [<type_renvoyé>] [{<propriétés>}]

La syntaxe de définition d'un paramètre (<paramètre>) est la suivante :

[<direction>] <nom_paramètre> : <type> ['['<multiplicité>']'] [=<valeur_par_défaut>]

On peut représenter les méthodes de classe (*static* en Java) en soulignant la déclaration de la méthode.

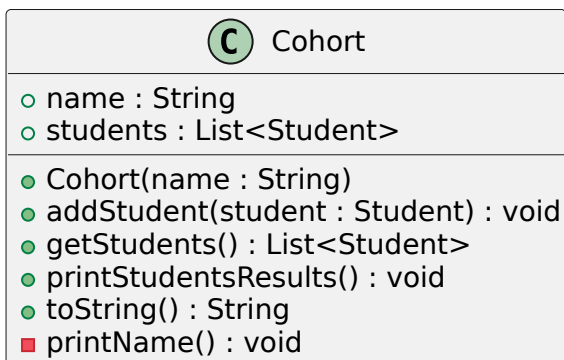
Pour une méthode abstraite (*abstract* en Java), on met la déclaration de la méthode en *italique*.

Exemples :

— '+ toString() : String pour une méthode publique, sans paramètre et retournant une chaîne de caractères de type String'

— `~ setAbscisse(x : float)` pour une méthode visible dans le paquetage, et prenant un nombre à virgule de type `float` en paramètre

La classe `Cohort` de la première planche de TP est représenté par le diagramme ci-dessous (généré à l'aide de PlantUML).



Un des apports les plus importants des diagrammes de classe est de décrire la relation entre les classes d'un projet de développement. Il existe de nombreux types de relations que peuvent avoir les classes.

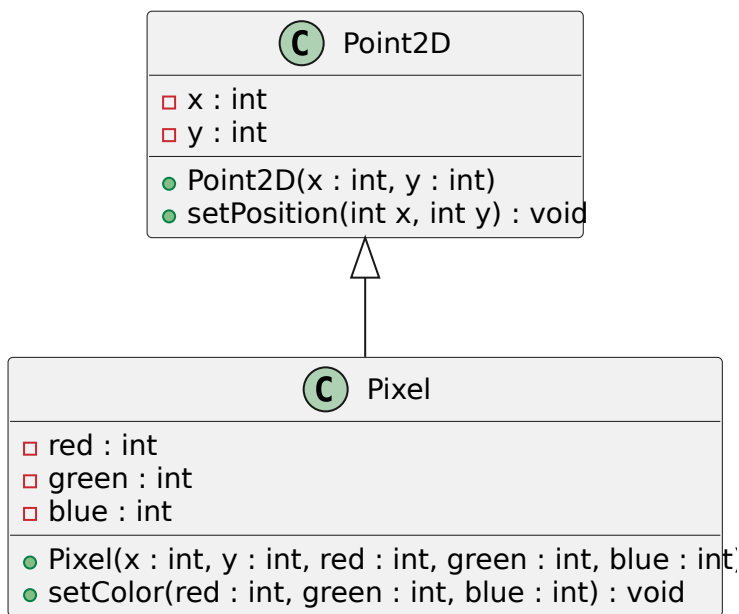
2.3.2 Relation d'extension

L'extension décrit une relation entre une classe parent et une classe enfant. La classe enfant est intégralement cohérente avec la classe parent, mais comporte des informations supplémentaires (attributs, méthodes, ...). Une classe enfant peut redéfinir (même signature) une ou plusieurs méthodes de la classe parent. Sauf indication contraire, un objet utilise les méthodes les plus spécialisées dans la hiérarchie des classes. Toutes les associations

de la classe parent s'appliquent aux classes enfants. Un objet de la sous-classe peut être utilisé partout où un objet de la classe de base est autorisé (principe que l'on verra plus en détails dans la suite du cours).

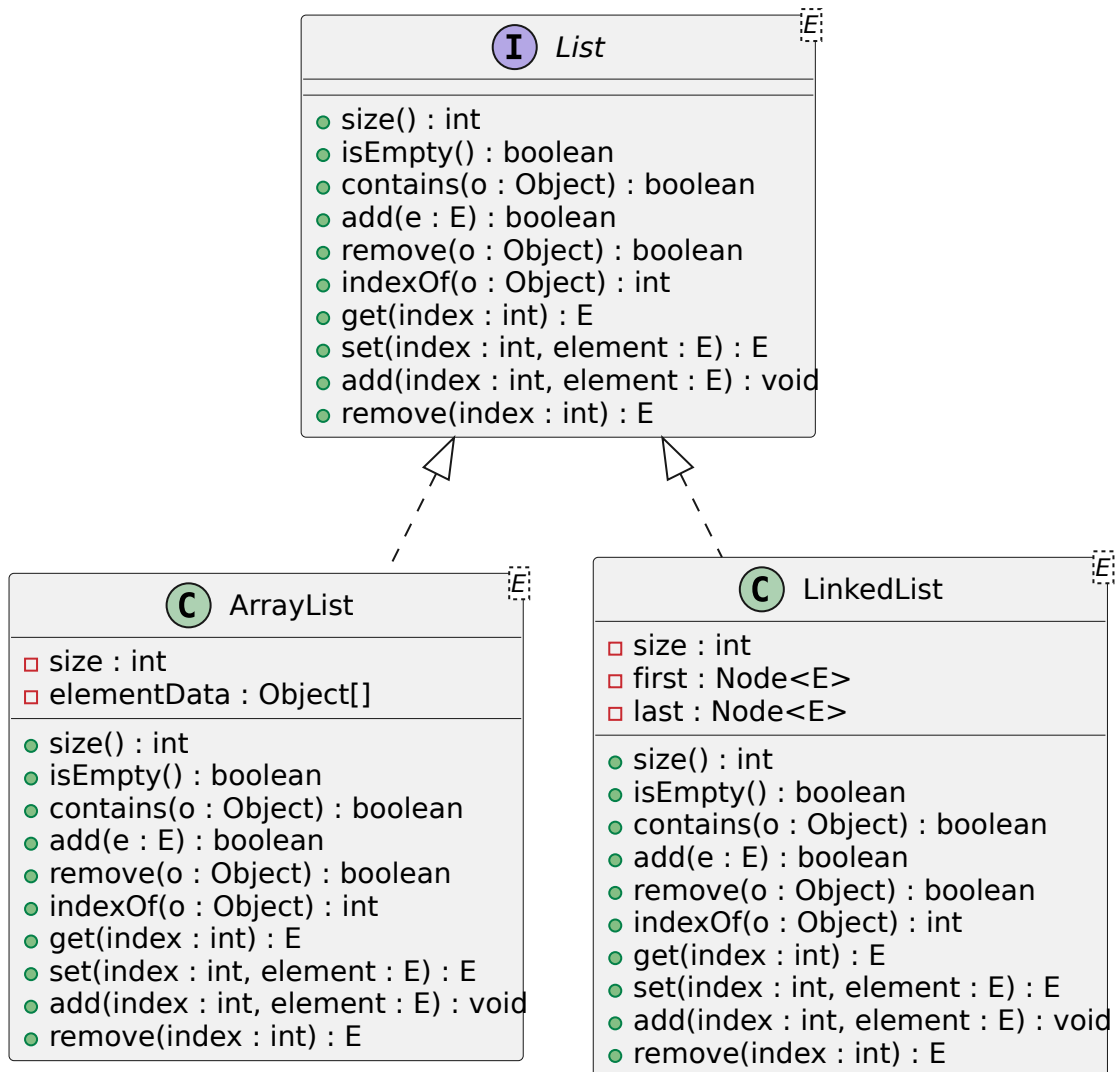
Le symbole utilisé pour la relation d'extension est une flèche avec un trait plein dont la pointe est un triangle vide (blanc) vers la classe parent.

Le diagramme ci-dessous donne la relation entre une classe parent permettant de représenter un point du plan (`Point2D`) et une classe enfant permettant de représenter un pixel (`Pixel`).



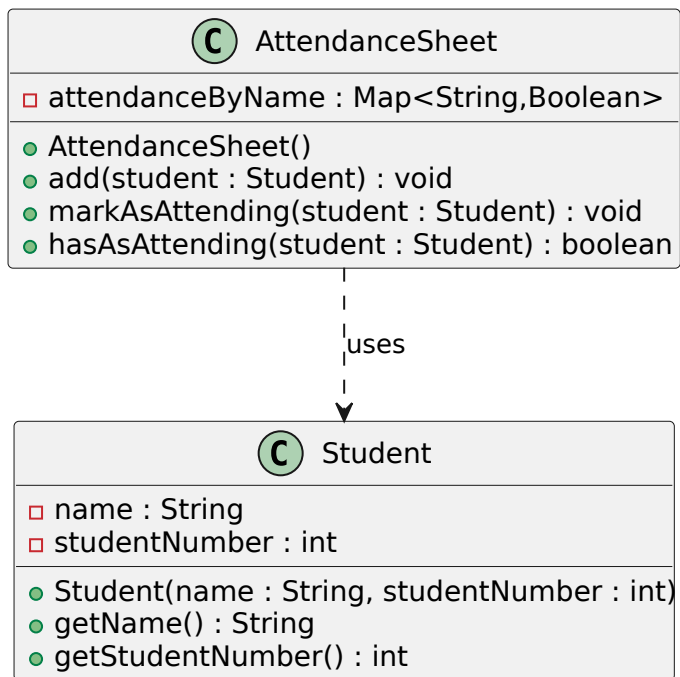
2.3.3 Relation d'implémentation

Une relation d'implémentation existe entre deux classes lorsqu'une d'entre elles doit implémenter ou réaliser le comportement spécifié par l'autre. Dans le cas de Java, c'est généralement la relation entre une classe et une des interfaces qu'elle implémente. Une relation d'implémentation est représentée par une flèche composée d'un trait composé de tirets et d'un triangle vide. Le connecteur part du client (qui réalise le comportement) vers le fournisseur (qui spécifie le comportement). Dans le diagramme ci-dessous, nous avons représenté les classes `ArrayList` et `LinkedList` qui implémentent toutes deux l'interface `List` (toutes les méthodes de `List` ne sont pas décrites afin d'éviter de rendre trop complexe le diagramme).



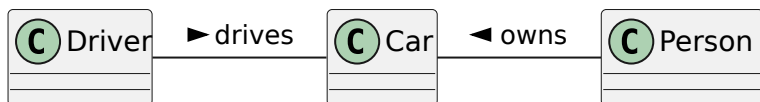
2.3.4 Relation de dépendance

Une dépendance est une relation indiquant qu'une classe en utilise une autre classe comme type dans les arguments d'une de ses méthodes. L'exemple ci-dessous montre la relation entre une classe `AttendanceSheet` permettant de représenter des feuilles de présence et une classe `Student` permettant de représenter des étudiants. Le symbole utilisé pour la relation de dépendance est une flèche avec un trait en tirets partant de la classe dépendante. On ajoute généralement du texte (dans notre exemple *uses*) afin de préciser la nature exacte de la relation entre les classes.

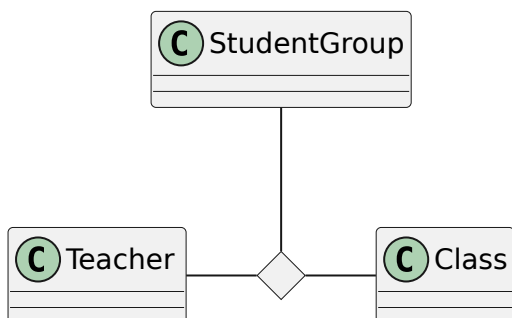


2.3.5 Relation d'association

Une association est une relation structurelle entre objets. Ce type de relation peut être binaire, dans ce cas, elle est représentée par un simple trait, ou n -aire, auquel cas les classes sont reliées à un losange par des traits simples. La relation peut être nommée et on peut indiquer le sens de lecture de son nom avec un symbole ►. Par exemple, le diagramme ci-dessous indique les relations entre des voitures, des conducteurs et des propriétaires de voitures à l'aide d'associations binaires.

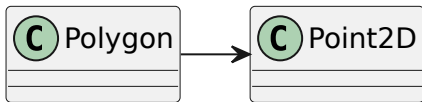


Le diagramme ci-dessous illustre une relation ternaire entre un cours, l'enseignant assurant le cours et les étudiants assistant au cours.



Il est possible d'indiquer un sens à la relation. Dans une relation binaire, on peut indiquer à l'aide d'une flèche que seuls les objets d'une classe source a la connaissance des objets de la classe destination. Dans l'exemple ci-dessous, seuls les objets de classe `Polygon` connaissent ceux de la classe `Point2D` auxquels ils sont liés, mais

pas l'inverse.



2.3.5.1 Relation d'agrégation L'agrégation est une relation d'association dans laquelle une classe **Part** est contenue dans une classe **Aggregate**. Plus précisément, on a les propriétés suivantes :

- une (ou plus) instance(s) de **Part** est stockée dans une instance de **Aggregate** ;
- Chaque instance de **Part** peut appartenir à plusieurs instances de **Aggregate** ;
- La durée de vie des instances de **Part** n'est pas gérée par les instances de **Aggregate** qui l'a contiennent (les instances **Part** contenues dans un objet **Aggregate** ne sont pas détruites quand celui-ci est détruit) ;
- Les instances de **Part** n'ont pas connaissance de l'existence des instances de **Aggregate**.

Pour résumer, lorsqu'il y a une relation d'agrégation, une instance d'une classe contient des objets d'une autre classe, mais ceux-ci ont leur vie propre. Comme exemple intuitif pour une agrégation, on peut considérer des blocs de construction d'une maison jouet. La maison jouet est construite à partir de blocs. Il est possible de la démonter entièrement, mais les blocs resteront disponibles pour une autre construction.

On représente cette relation par un trait reliant la classe **Aggregate** à la classe **Part** avec un losange vide à l'extrémité connecté à **Part**.

Pour un exemple plus proche de la programmation orientée objet, on peut considérer les classes **Point** et **AggregateCircle** suivantes :

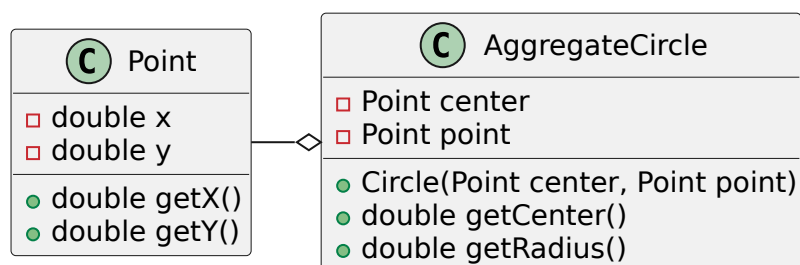
```
1 public class Point {
2     public final double x, y;
3
4     public Point(double x, double y){
5         this.x = x;
6         this.y = y;
7     }
8
9     public double distanceTo(Point p){
10        double dx = this.x - p.x;
11        double dy = this.y - p.y;
12        return Math.hypot(dx, dy);
13    }
14
15    public double getX(){
16        return x;
17    }
18
19    public double getY(){
20        return y;
21    }
22 }
23
24 public class Circle {
25     private Point center, point;
```

```

26
27 public Circle(Point center, Point point){
28     this.center = center;
29     this.point = point;
30 }
31
32 public Point getCenter(){
33     return center;
34 }
35
36 public double getRadius(){
37     double dx = center.x - point.x;
38     double dy = center.y - point.y;
39     return Math.hypot(dx, dy);
40 }
41 }

```

Le diagramme de classes correspondant est le suivant :



2.3.5.2 Relation de composition Tout comme l'agrégation, la composition est une relation d'association dans laquelle une classe **Part** est contenue dans une classe **Composite**. La différence est que pour la composition le cycle de vie des parties est dépendant du cycle de vie des objets composés. Plus précisément, on a les propriétés suivantes :

- une (ou plus) instance(s) de **Part** est stockée dans une instance de **Composite** ;
- Chaque instance de **Part** appartient à une seule instance de **Composite** ;
- Chaque instance de **Part** a sa durée de vie (construction et destruction) gérée par l'instance de **Composite** qui l'a contenue;
- Les instances de **Part** n'ont pas connaissance de l'existence des instances de **Composite**.

Pour résumer, lorsqu'il y a une relation de composition, une instance d'une classe contient des objets d'une autre classe, mais ceux-ci ont leur vie propre. Comme exemple intuitif pour une composition, on peut considérer une maison contenant une ou plusieurs pièces. La durée de vie d'une pièce est contrôlée par la maison, car elle ne peut pas exister sans la maison.

On représente cette relation par un trait reliant la classe **Aggregate** à la classe **Part** avec un losange vide à l'extrémité connecté à **Part**.

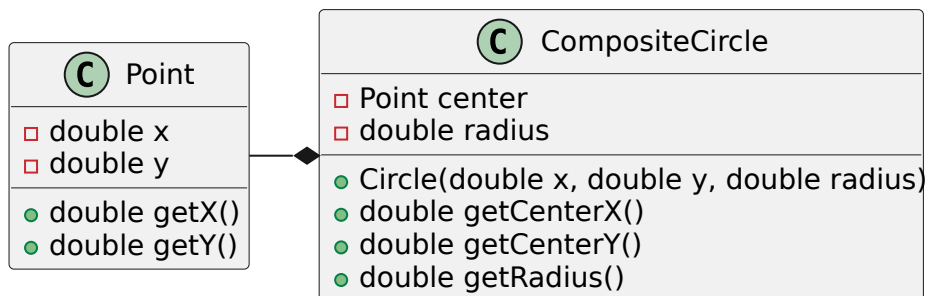
Pour un exemple plus proche de la programmation orientée objet, on peut considérer la classe **Point** déjà vue pour l'agrégation avec la classe **CompositeCircle** suivante :

```

1 public class CompositeCircle {
2     private Point center;
3     private double radius;
4
5     public Circle(double x, double y, double radius){
6         this.center = new Point(x, y);
7         this.radius = radius;
8     }
9
10    public double getCenterX(){
11        return center.getX();
12    }
13
14    public double getCenterY(){
15        return center.getY();
16    }
17
18    public double getRadius(){
19        return radius;
20    }
21 }

```

Le diagramme de classes correspondant est le suivant :

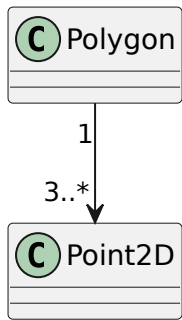


2.3.5.3 Multiplicités Lorsqu'on définit une relation d'association (qui peut être une agrégation ou une composition), il est souvent utile de définir la multiplicité, c'est-à-dire le nombre d'objets possibles pour chaque extrémité de la relation. Dans une association binaire, la multiplicité sur une extrémité contraint le nombre d'objets de la classe connecté à l'extrémité pouvant être associés à un seul objet de l'autre classe. La syntaxe pour une multiplicité est `min..max`, où `min` et `max` sont des nombres représentant respectivement les nombre minimaux et maximaux d'objets concernés par l'association. On peut utiliser une `*` à la place de `max` pour signifier que le nombre d'objets n'est pas borné. De plus, `n..n` se note aussi `n` et `0..*` se note aussi `*`.

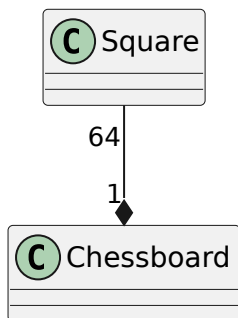
Exemples de multiplicités :

- exactement un : `1` ou `1..1` ;
- plusieurs : `*` ou `0..*`;
- au moins un : `1..*` ;
- de un à six : `1..6`.

Dans l'exemple ci-dessous, on donne les multiplicités de la relation entre `Point` et `Polygon` en précisant qu'un point n'est que dans un polygone et qu'un polygone contient au moins 3 points.



Dans l'exemple ci-dessous, on donne les multiplicités de la relation entre un échiquier (classe `Chessboard`) et une case (classe `Square`).



2.3.6 Exemple de diagramme de classe

Pour cet exemple, nous allons considérer le problème suivant de commande d'articles en ligne. Il est défini avec les contraintes suivantes :

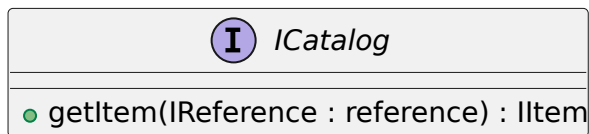
- Un catalogue regroupe des articles, il permet de trouver un article à partir de sa référence.
- Un article est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre.
- Une commande est créée pour un client et un catalogue donnés, on peut ajouter des articles à une commande, accéder à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- Un client peut créer une commande pour un catalogue et commander dans cette commande des articles à partir de leurs références.

Pour réaliser le diagramme de classes, on va tout d'abord déterminer quelles sont les interfaces dont on va avoir besoin pour satisfaire aux besoins posés. La première question à se poser est de déterminer quels sont les objets nécessaires à la résolution du problème. Pour ce problème, on peut identifier les objets suivants : catalogue, article, commande, référence, client, prix. On peut donc considérer 5 types d'objets différents et par conséquent 5 interfaces : `ICatalog`, `IItem`, `IOrder`, `IReference`, `IPrice`, `IClient` (on met souvent un `I` devant le nom des interfaces afin de les différencier des classes concrètes).

La deuxième question à laquelle on doit répondre est de déterminer quelles sont les méthodes des interfaces. Pour cela, il suffit généralement d'extraire de la définition du problème les actions (donc les verbes) que l'on peut réaliser sur les objets.

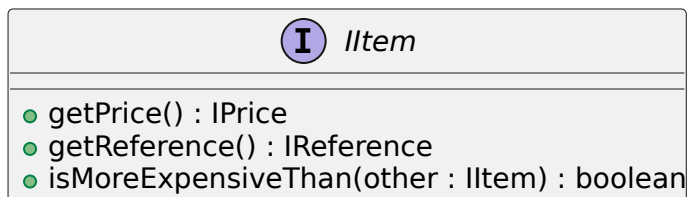
Un catalogue regroupe des articles, il permet de **trouver** un article à partir de sa référence.

⇒ L'interface pour les catalogues peut donc être la suivante :



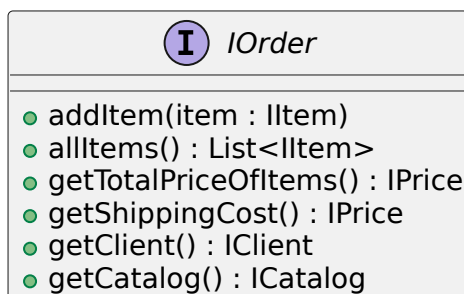
Un article est caractérisé par un prix et une référence que l'on **peut obtenir**. On veut aussi pouvoir **déterminer** si un article est plus cher qu'un autre

⇒ L'interface pour les articles peut donc être la suivante :



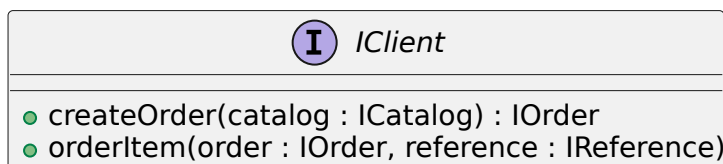
Une commande est **créée** pour un client et un catalogue donnés, on peut **ajouter** des articles à une commande, **accéder** à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.

⇒ L'interface pour les commandes peut donc être la suivante :



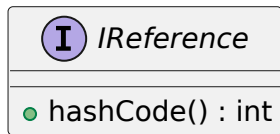
Un client peut **créer** une commande pour un catalogue et **commander** dans cette commande des articles à partir de leurs références.

⇒ L'interface pour les clients peut donc être la suivante :

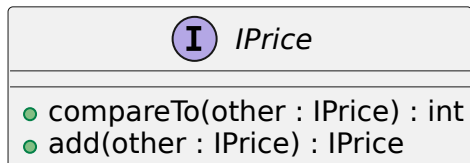


Pour les références, il y a moins de précisions dans la description du problème. Puisqu'une référence sert de clé

dans un catalogue, il semble logique qu'elle implémente la méthode `hashCode` permettant l'utilisation de l'objet dans une table de hashage.

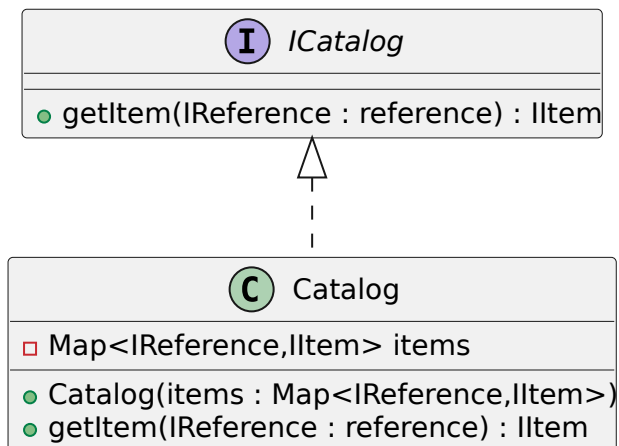


Pour les prix, les deux opérations qui semblent essentielles sont la possibilité de comparer et d'ajouter des prix.

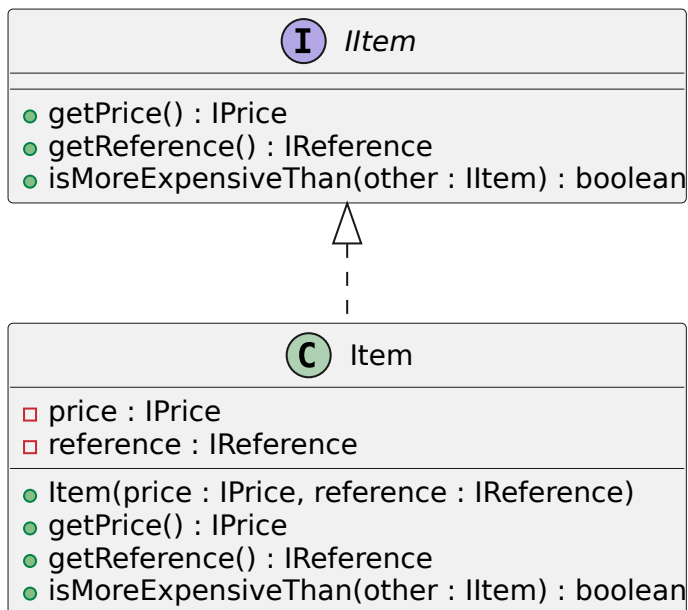


Maintenant qu'on a défini des interfaces, il faut définir les classes qui vont les implémenter en choisissant la structure de leurs données (leurs attributs).

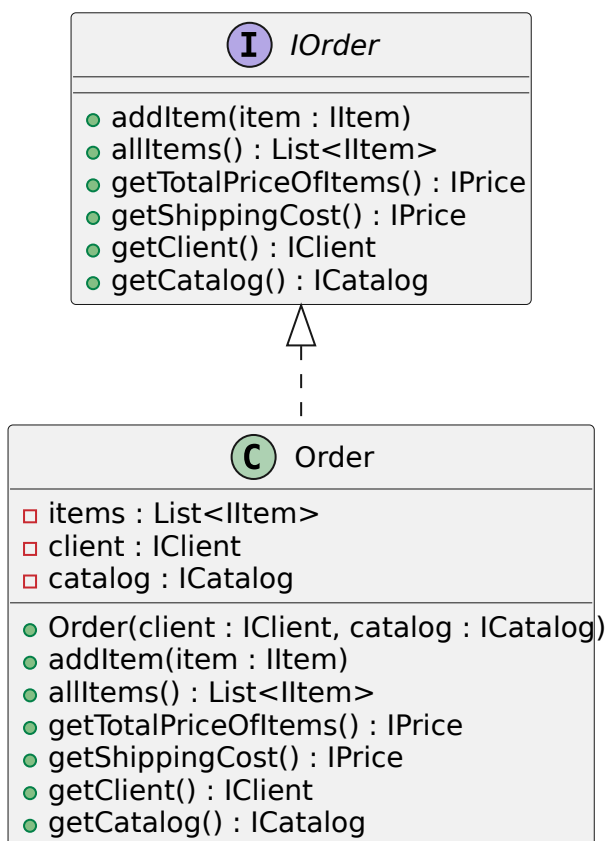
— Classe `Catalog`



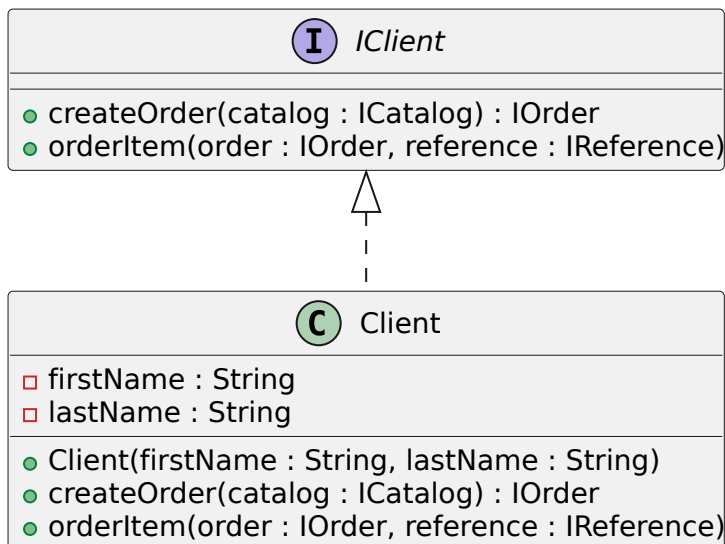
— Classe `Item`



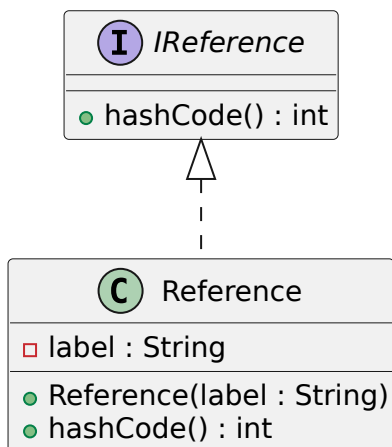
— Classe *Order*



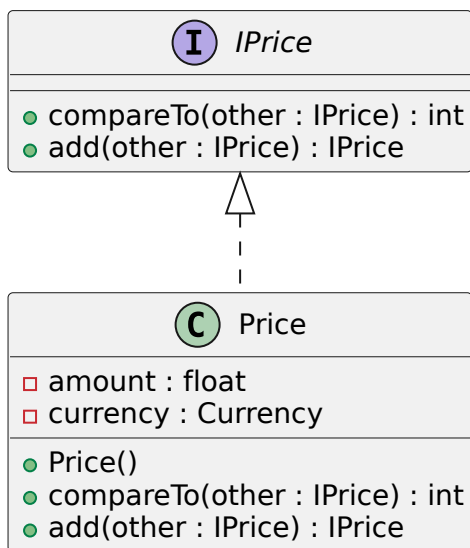
— Classe *Client*



— Classe *Reference*



— Classe *Price* (on utilise la classe *Currency* de Java).



En ajoutant les relations avec leur multiplicité, on obtient le diagramme suivant :

