Penser objet, encapsulation et types primitifs

Arnaud Labourel arnaud.labourel@univ-amu.fr

Section 1

Le penser objet

Analyse d'un problème en programmation objet

Penser objet : décomposer le programme en objets

- Quels sont les objets nécessaires à la résolution du problème ?

  $\Rightarrow$ décomposition du problème en objets

- À quels modèles des objets correspondent-ils ?

  $\Rightarrow$ Quelles sont les classes ?

- Quels sont les fonctionnalités/opérations dont on doit/veut pouvoir disposer sur ces objets ?

  $\Rightarrow$ Quelles sont les méthodes des classes ?

- Quelle est la structure des données de l'objet ?

  $\Rightarrow$ Quelles sont les attributs des classes ?

Exemple de problème

- 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 leur référence.

Classes du problème

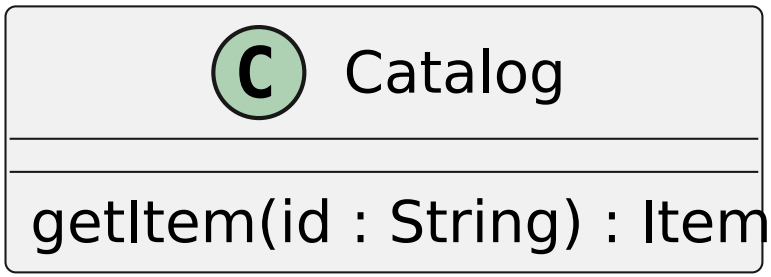
- un **catalogue** regroupe des **articles**, il permet de trouver un article à partir de sa **référence**.
- un **article** est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre
- une **commande** est créée pour un **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 leur référence.

Méthodes

- 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 leur référence.

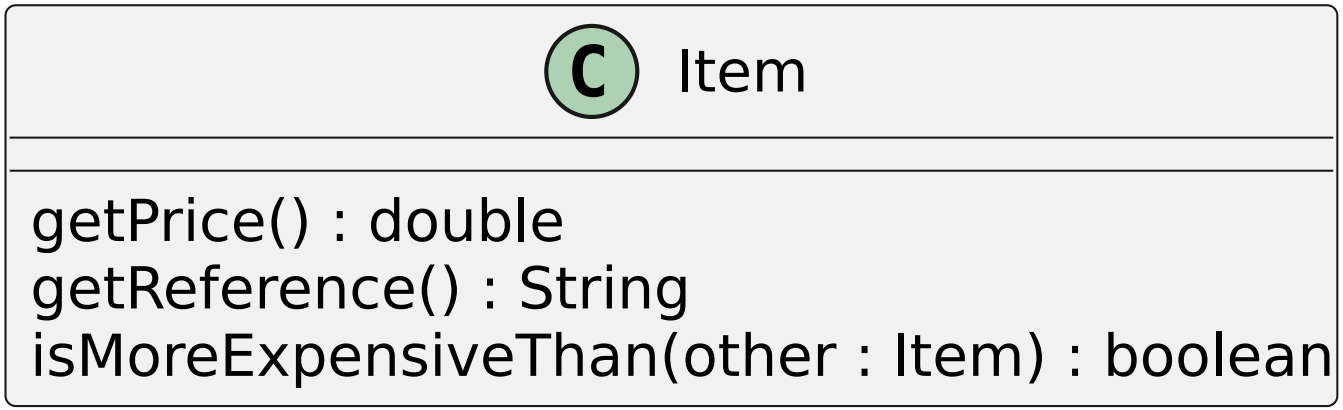
Description d'un catalogue

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



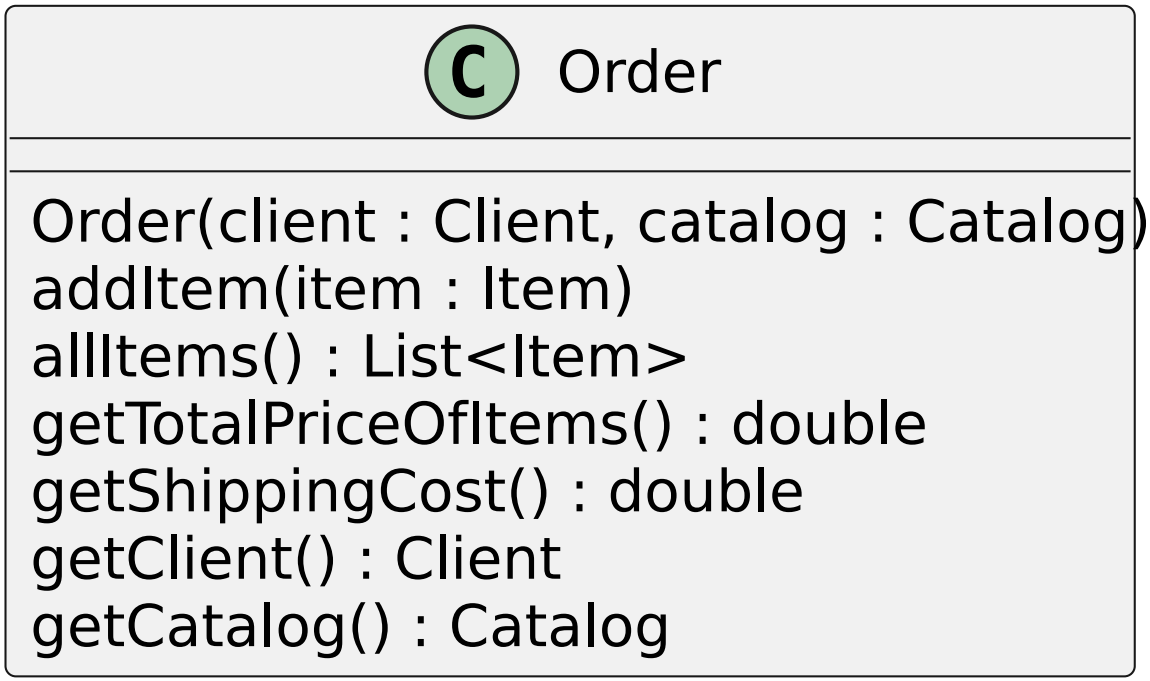
Description d'un article

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



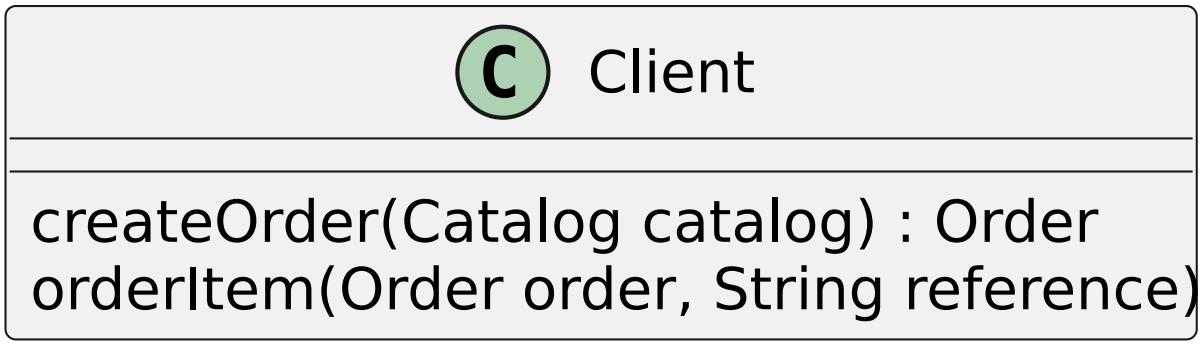
Description d'une commande

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.



Description d'un client

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



Utilisation des classes

Créer une commande pour un client, faire commander 2 articles par le client, obtenir le prix total des articles de la commande

on suppose les références disponibles et initialisées :

```
Client client = new Client(...)
Catalogue cata = new Catalogue(...)

Order order = client.createOrder(cata);
client.orderItem(order,"A0527");
client.orderItem(order,"B3879");
float price = order.getTotalPrice();
```

Code des méthodes

La méthode `orderItem` de `Client` permet d'ajouter un article à une commande à partir de son identifiant quel code pour cette méthode ?

```
public void orderItem(Order order, String id) {
  // récupérer le catalogue de la commande
  Catalog cata = order.getCatalog();
  // récupérer l'article dans le catalogue
  // à partir de la référence
  Item item = cata.getItem(id);
  // ajouter l'article à la commande
  order.addItem(item);
}
```

Section 2

Encapsulation

Exemple classe `Client`

Pour le moment tout code ayant accès à un `Item`, a accès à son attribut `price` en lecture et écriture.

```
public class Client{
  void orderItem(Order order, String id) {
    Catalogue cata = order.getCatalogue();
    Item item = cata.getItem(id);
    item.price = 0; // l'item devient gratuit
    order.addItem(item);
  }
}
```

Comment « empêcher » cela ?

Comment interdire la modification des attributs depuis l'extérieur ?

Contrôle d'accès

- **Restreindre** la visibilité des attributs ou méthodes d'une classe.
- En Java : **modificateurs** d'accès précisés lors de la définition d'attributs ou méthodes.

Mot-clés `private` et `public`

Utilisables à la déclaration des attributs, méthodes et constructeurs.
Deux modificateurs d'accès possibles :

- `private` : accessible uniquement pour les instances de la classe, c'est-à-dire uniquement depuis le code des méthodes de la classe
- `public` : accessible pour tout le monde, c'est-à-dire dans le code de n'importe quelle méthode

La classe Item

```
public class Item {
  private float price;
  private String id;
  public float getPrice() { return this.price; }
  public float getId() { return this.id; }
  public boolean moreExpensiveThan(Item otherItem) {
    return this.price > otherItem.price;
  }
  public Item(float p, String id) {
    this.price = p;
    this.id = id;
  }
}
```

Méthode getTotalPrice dans Order

```
public class Order {
  List<Item> items;
  public float getTotalPrice() {
    float total = 0;
    // cumuler les prix de tous les articles
    for(Item item : this.items) {
      total += item.price; // interdit private !
    }
  return total;
  }
}
```

Erreur à la compilation : `price` étant un attribut privé, il n'est pas possible d'y accéder depuis une autre classe (ici `Order`).

Méthode getTotalPrice dans Order

```
public class Order {
    List<Item> items;
  public float getTotalPrice() {
    float total = 0;
    // cumuler les prix de tous les articles
    for(Item item : this.items) {
      total += item.getPrice();
      // autorisé public !
    }
  return total;
  }
}
```

On peut accéder à la valeur de price grâce à un getter qui lui est public.

Règle d'encapsulation

Règle

Rendre privés les attributs caractérisant l'état de l'objet et fournir si besoin des méthodes publiques permettant de modifier/accéder à l'attribut

`get` et `set`

accesseur/mutateur = getter/setter

Pour un attribut `float price` :

- `getPrice()` : accesseur
- `setPrice(float newPrice)` : mutateur/modificateur

Exemple getter/setter

```java
private String name;

public String getName() {
  // accès en lecture
  return this.name;
}

public void setName(String newName) {
  // accès en écriture
  this.name = newName;
}
```

Principe d'encapsulation

Ne pas laisser l'état d'un objet en libre accès en modification

Intérêt de l'encapsulation

- **masquer l'implémentation**

  → toute la décomposition du problème n'a pas besoin d'être connue du « programmeur utilisateur »

- **protéger**

  → l'objet a le contrôle sur son état (contrôle sur les modifications)

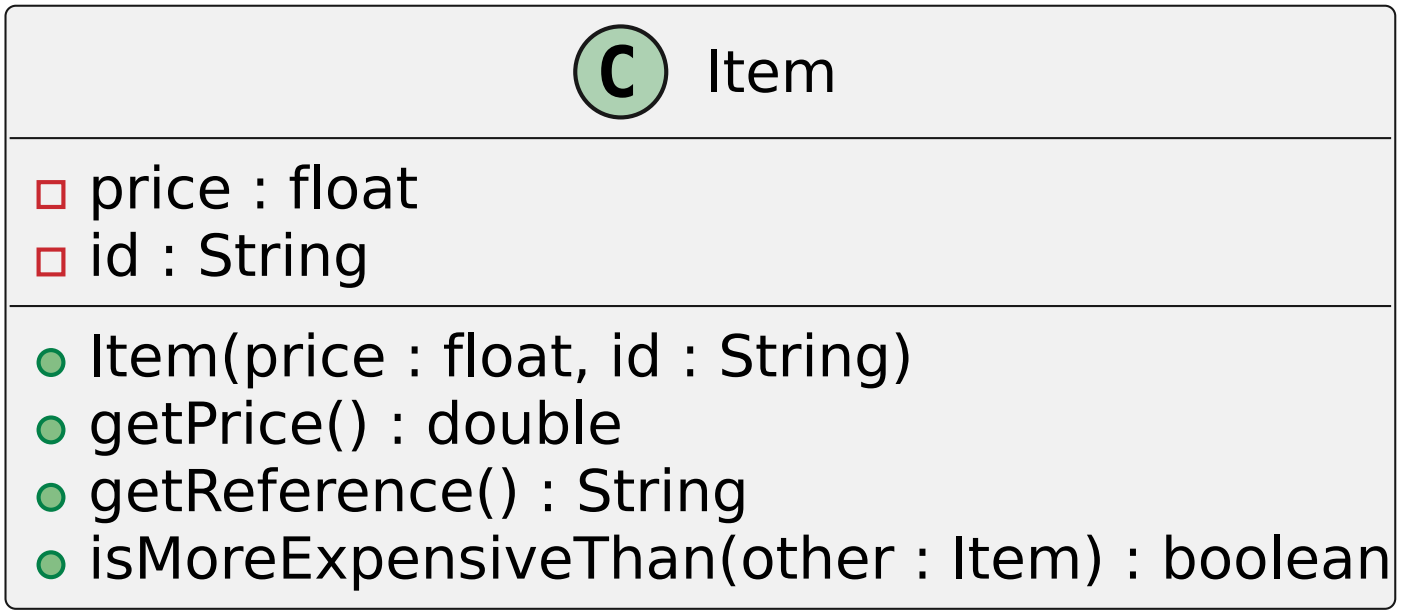
  → préserver l'intégrité des objets

  → le « programmeur créateur » contrôle (et est responsable) de son interface par rapport au « programmeur utilisateur »

- **permettre l'évolutivité**

  → il est possible de modifier tout ce qui n'est pas public sans impact pour le « programmeur utilisateur »

`private` et `public` dans les diagrammes de classes



Règles

- ○ = `public` (autre symbole possible **+**)
- □ = `private` (autre symbole possible **-**)

Encapsulation et contrôle de l'état

Toutes les valeurs autorisées par le type des attributs ne sont pas forcément correctes pour l'objet :

- attribut `int month` dans `Date` doit être entre 1 et 12
- attribut `double celsiusValue` dans `Temperature` doit être supérieur à `-273.15` (zéro absolu)
- attribut `double price` dans `Item` doit être positif

Toutes les valeurs autorisées par le type des paramètres d'une méthode ne sont pas forcément correctes :

- paramètre `double amount` dans `withdraw` doit être supérieur ou égal à 0 (pas de retrait d'argent d'un montant négatif)
- paramètre `int divisor` dans `divide` ne doit pas être égal à 0 (pas de division par zéro)

$\Rightarrow$ interdire la modification dans certains cas (exceptions)

Section 3

Les types primitifs vs objets

Les types primitifs

En java, il existe des types **primitifs** qui ne sont pas des objets :

type	catégorie	taille	valeurs possibles	affichage
byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31}-1$	0
long	entier	64 bits	-2^{63} à $2^{63}-1$	0
float	flottant	32 bits		0.0
double	flottant	64 bits		0.0
char	caractère	16 bits	caractère unicode	'\000'
boolean	booléen	non définie	`false` ou `true`	`false`

Types primitifs vs objets

- Les noms des types primitifs commencent par un minuscule (vs majuscule pour les types objets).
- Il n'y a pas de classe associée aux types primitifs et ils ne peuvent pas être utilisé pour appeler une méthode.
- Il n'y a pas de constructeurs pour les types primitifs (pas d'instanciation).
- Les variables de type primitif contiennent directement la valeur et **pas** une référence comme c'est le cas pour les objets.
- On affecte les variables de type primitifs avec :
  - des littéraux : `'a'`, `12`, `42.0`, `42.0f`, ...
  - des résultats d'opérations : ‘`2. + x,2 / 5,3 * 4`‘, ...

Comportement types primitifs

Lors d'un appel de méthode les arguments sont passés par valeur : une copie de la valeur de l'argument est créé lors de l'appel.

Cela un impact différent suivant que l'argument soit un objet ou un type primitif :

- Pour les objets, cela signifie passer une copie de la **référence** : il est donc possible de modifier l'état de l'objet.
- Pour les types primitifs, cela signifie que l'argument est une **copie** uniquement créée pour l'appel et toute modification de sa valeur n'aura pas d'impact en dehors de l'appel.

Exemple comportement objet

```
public class Point {
  private double x, y;
  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }
  public double getX() { return x; }
  public double getY() { return y; }
  public void setX(double x) { this.x = x; }
  public void setY(double y) { this.y = y; }
  public String toString() {
    return "(" + x + ", " + y + ")";
  }
}
```

Exemple comportement objet receveur

```
public class App{
  public static void main(String[] args){
    Point p = new Point(0., 0.);
    p.setX(1.);
    p.setY(2.);
    System.out.println(point.toString());
    // => affiche (1.0, 2.0)
  }
}
```

L'objet passé en notation pointée est modifié.

Exemple comportement objet en argument

```
public class Point {
  public void copyInto(Point p){
    p.setX(getX());
    p.setY(getY());
  }

  public static void main(String[] args){
    Point p1 = new Point(1, 2);
    Point p2 = new Point(0, 0);
    p1.copyInto(p2);
    System.out.println(p2.toString());
    // => affiche (1.0, 2.0)
  }
```

L'objet passé en argument est modifié.

Exemple comportement type primitif en argument

```
public class Point {
  public double distanceTo(Point p){
    return Math.hypot(p.x - x, p.y - y);
  }
  public void addDistanceTo(double d, Point p){
    d += this.distanceTo(p);
  }
  public static void main(String[] args){
    double d = 0;
    Point p1 = new Point(1, 2);
    Point p2 = new Point(0, 0);
    p1.addDistanceTo(d, p2);
    System.out.println(d); // => affiche 0.0
  }
```

Le contenu de la variable passée en argument n'est pas modifié.