Types & Interfaces : Quelques exemples

Sébastien Mosser - INF5153
Chapitre 2 - Capsule 5
Automne 2020

Premier exemple : Comparer des objets

ENCART CAMÉRA
BY NC SA

- Java fournit des "**services**" utiles sous la forme d'interface "***-able***"
  - ***Cloneable, Serializable, Comparable, Iterable, Runnable, Callable, Observable, Closeable, …***
- *Une classe peut réaliser "**X-able**" pour dire qu'elle sait faire **X***
  - ***Serializable** est un cas à part en java, c'est plus un **marqueur** qu'une interface*
- *Les abstractions de la bibliothèque Java reposent sur les interfaces*
- ***Exemple** :*
  - *Pour **trier une collection**, il faut savoir **comparer deux objets***
  - *Si on implémente **Comparable**, les **tris deviennent disponibles***

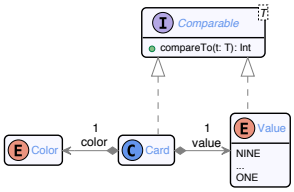
Le contrat de comparaison

ENCART CAMÉRA
BY NC SA

- L'interface ***Comparable<C>*** est utilisé pour définir ***une relation d'ordre sur l'ensemble des instances de la classe C***
- **Relation d'ordre** $\leq$ sur C :
  - $\forall c \in C, c \leq c$ *(réflexivité)*
  - $\forall (c_1, c_2, c_3) \in C^3, (c_1 \leq c_2 \wedge c_2 \leq c_3) \Rightarrow c_1 \leq c_3$ *(transitivité, préordre)*
  - $\forall (c_1, c_2) \in C^2, (c_1 \leq c_2 \wedge c_2 \leq c_1) \Rightarrow c_1 = c_2$ *(antisymétrie, ordre)*
- *Comment se comparer à* `null` *? (l'ordre est partiel, c'est un peu gonflant)*
- *On peut définir des classes d'équivalence (≠ égalité)*
  - $c_1.compareTo(c_2) = 0 \Leftrightarrow c_1 \equiv c_2$

Comparer deux cartes Schotten Totten

ENCART CAMÉRA



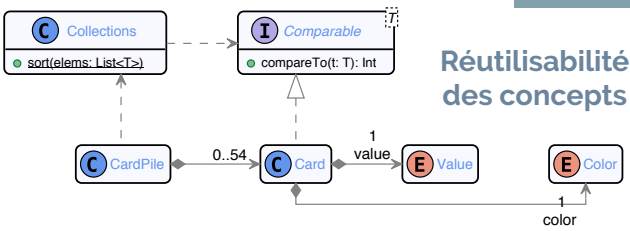
Les énumérations Java sont gratuitement comparable
(selon leur ordre de déclaration)

Classes d'équivalence définies par les cartes :
- *de même valeur*
- *et de couleurs ≠*

```
public class Card implements Comparable<Card> {

  private Value value;

  public int compareTo(Card c) {
    return value.compareTo(c.value)
  }
}
```


Utiliser Comparable plutôt qu'un truc maison

ENCART CAMÉRA



Réutilisabilité des concepts

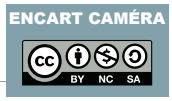
La classe utilitaire Collections fourni un ensemble de méthodes de tri, qui utilise le service de comparaison (et font l'hypothèse du respect du contrat de relation d'ordre)

Limitations de la réalisation d'interface

ENCART CAMÉRA

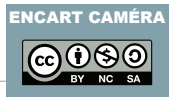
- Implémenter **Comparable** rend une classe C … **comparable** !
  - *C **offre le service** de comparaison*
  - ***Les instances** de C **sont** comparables*
- **Et si on voulait comparer de différentes manières ?**
  - **Carte classiques** : `{A, R, D, V, 10, 9, 8, 7, 6, 5, 4, 3, 2}`
  - **A la belote** :
    - A l'atout : `{V, 9, A, 10, R, D, 9, 8, 7, 6, 5, 4, 3, 2}`
    - Aux autres couleurs : `{A, 10, R, D, V, 9, 8, 7, 6, 5, 4, 3, 2}`

C'est un problème de Responsabilité



- On a donné à la Carte la **responsabilité de se comparer**
- **Ce choix n'est plus adapté** pour des comparaisons variés
  - Le type de Jeu **influencerait le comportement** des cartes
  - *Et mettrait un grand coup de pelle à l'encapsulation …*
- **On utilise classiquement des comparateurs dédiés**
  - Chaque jeu dispose de son comparateur
    - *C'est le comparateur qui porte la responsabilité*
- **Le comparateur est un "function object"** *(pas d'attributs)*
  - *C'est **pas fou** d'un point de vue OO. Mais c'est le meilleur **compromis***

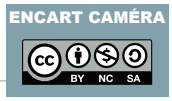
Mise en oeuvre d'un Comparator



```
public class Card {

  private Value value;

  static class CompareByValue implements Comparator<Card> {
    public int compare(Card c1, Card c2) {
      return c1.value.compareTo(c2.value);
    }
  }
}

public class CardPile {

  public void sort() {
    Collections.sort(this.cards, new Card.CompareByValue());
  }

}
```

Avec une classe interne, on peut accéder aux attributs privés

Définition d'un Comparator "à la volée"



```
public class CardPile {

  public void sort() {
    Collections.sort(this.cards, new Comparator<Card>() {
      public int compare(Card c1, Card c2) {
        return c1.getValue().compareTo(c2.getValue());
      }
    }
  }
}
```

- *Classe anonyme :*
  - *définie juste quand on en à besoin*
    - *pas réutilisable (par définition)*
  - *Repose sur l'interface publique pour comparer*

Comparateurs & "Lambda-expressions"


```
public class CardPile {

  public void sort() {
    Collections.sort(this.cards,
                     (c1, c2) -> {
      c1.getValue().compareTo(c2.getValue());
    });
  }
}
```

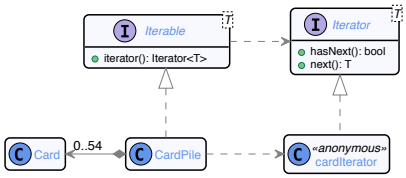
- *Lambda-expression :*
  - *Sucre syntaxique sur l'instanciation de classes anonymes*
  - *Attention à la lisibilité du code.*

Second exemple : Accéder aux cartes de la pile

- Quels sont les **services** dont on à besoin ?
  - **Itérer** sur le contenu de la pile
  - **Obtenir un moyen** d'itérer

- Java propose deux mécanismes pour cela :
  - L'interface ***Iterator<T>*** pour traverser une collection d'objets
    - Fournit les méthodes "***hasNext()***" et "***next()***"
  - L'interface ***Iterable<T>*** qui produit un *itérateur* de T
    - De manière uniforme, avec la méthode "***iterator()***"

Responsabilisation des classes




```
public class CardPile implements Iterable<Card> {

  private List<Card> cards;

  public Iterator<Card> iterator() { return cards.iterator(); }

}
```

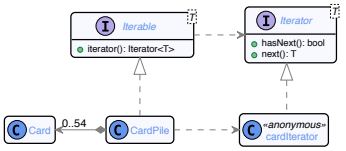
Avantage des abstractions réutilisables


```
List<Student> students = ...
for(Student s: students) {
  // ...
}
```

La construction de langage "foreach" est en fait du sucre syntaxique qui repose sur un Iterable en Java

```
List<Student> students = ...
for(Iterator<Student> it = students.iterator(); it.hasNext()) {
  Student s = it.next();
  // ...
}
```


Donc, par extension pour notre pile de carte ...




```
public class Main {

  public static void main(String[] args) {
    CardPile cP = new CardPile();
    for(Card c: cP) {
      // ...
    }
  }
}
```

Comment gérer la fourniture de plusieurs services ?

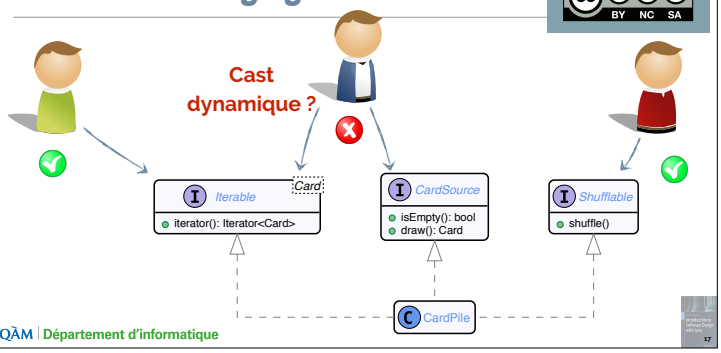
- *Quels sont les **services** proposés par la Pile de carte ?*
  - ***Se mélanger, Fournir des cartes, Pouvoir être parcourue***
- *Fait :*
  - ***on interagit avec les objets en envoyant des messages conformes à leur interface publique***
- *Comment faire pour notre pile de carte ?*
  - ***Une interface dédiée** à la pile, incluant tous les services ?*
  - *Un jeu de **plusieurs interfaces découpés par services**, et réalisées par la pile ?*

Comparaison des deux approches

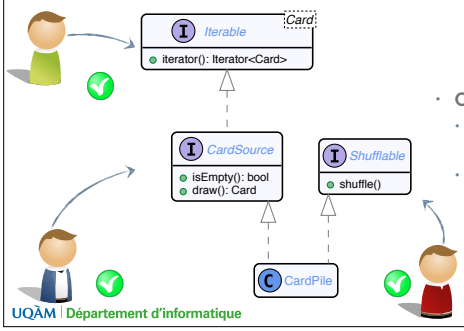
- Interface dédiée
  - *Très orienté sur la logique d'affaire*
  - *Est-ce que l'interface sera réalisée par d'autres classes ?*
  - *Couplage fort avec des services pas forcément utilisés*
- Jeu de plusieurs interfaces
  - *Coût supplémentaire de maintenance*
  - *Modularité maximale*
  - *Couplage minimal* (je consomme uniquement ce dont j'ai besoin)

Un consommateur de service ne doit pas dépendre d'interfaces dont il n'a pas besoin

Comment définir la ségrégation des interfaces ?



On peut définir des hiérarchies de réalisation



- Choix de conception
  - *Une pile de carte est une source de carte*
  - *On peut itérer sur une source de carte*

Principes de conception objet : SOLID

ENCART CAMÉRA

- **Les principes SOLID caractérisent de bonnes propriétés, qui sont attendues dans une conception objet de qualité.**
- Pour le moment, on s'est intéressé à deux principes :
  - **S : "*Single Responsibilty*"**
    - *Une classe doit porter une seule responsabilité identifiée.*
  - **I : "*Interface Segregation*"**
    - *Définir des interfaces spécialisées qui sont réalisées par les classes.*
- On verra dans les chapitres suivants **les principes O et L** *(et un peu D)*.

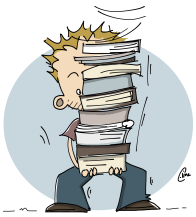
Digression : Principes, Paradigmes et Effets de Mode

ENCART CAMÉRA

- Les **principes de conceptions** sont stables :
  - *p.-ex.*, objets *stateless*, *statefull*, *immuables*, *mutables*, …
- Les **paradigmes** se construisent au-dessus des **principes**
- Par exemple :
  - ***"Les MICRO-SERVICES sont des entités stateless qui s'échangent des messages immuables."***
- Il faut **maîtriser les principes** pour bien **utiliser les paradigmes**
  - *Et les ~~modes~~ paradigmes changent, pas les principes.*

UQÀM | Département d'informatique
FACULTÉ DES SCIENCES
Université du Québec à Montréal

ENCART CAMÉRA



https://mosser.github.io/

https://ace-design.github.io/

***Abonne toi** à la chaine,*
et met un pouce bleu !