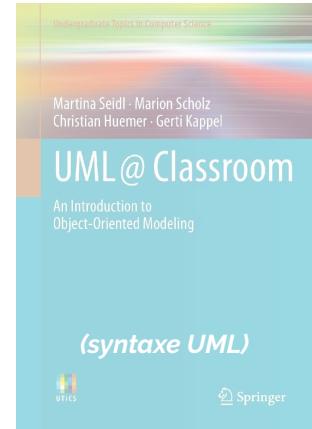




## Bibliographie de ce cours



Chapitres 2, 3 & 4



Chapitres 4, 5 & 6

## Intermède Publicitaire

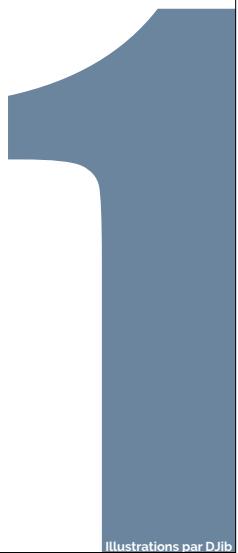
- Ce cours utilise intensivement les principes mis en avant dans le livre de Martin Robillard (McGill) aux chapitres cités.
  - Ainsi que le cheminement pour passer de l'un à l'autre
- Les exemples dans le livre sont ceux d'un jeu de cartes, on utilise pour INF-5153 aussi **un jeu mais de facture différente**.
  - On va aussi prendre de la liberté sur certains exemples.
- Ne recopiez pas le code du livre ou du cours dans vos projets,
  1. Parce que ça serait du **plagiat** ...
  2. Ce ne sont pas les même problématiques qui sont rencontrée dans les spécifications de l'arbitre de Poker.

- 1 Étude de cas : Schotten Totten
- 2 Encapsulation
- 3 Types & Interfaces
- 4 État des Objets
- 5 Choix de conceptions illustrés
- 6 Q&R sur le TP#1



# Schotten Totten

(Un jeu par Reiner Knizia)



## Un jeu de batailles basée sur des cartes

- **Neuf (9) bornes** représentant la **frontière** entre deux clans
- **Cinquante quatre (54) cartes régulières**, dites **cartes "clan"**.
  - De **valeur** un (1) à neuf (9), et de six (6) **couleurs**
  - Il existe des **cartes "tactiques"** dans la version avancée
- On peut poser **jusqu'à trois (3) cartes clan** devant une **borne**
- A son tour, un joueur :
  - **Pose une carte** devant une borne parmi les six (6) de sa main;
  - **Pige une carte** dans la pioche pour revenir à six (6) cartes.

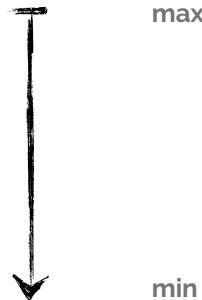
**“Qui sera le plus borné ?”**

## Conditions de victoire

- **Un** des deux **joueurs** est **considéré gagnant** quand :
  - **Il remporte cinq (5) bornes** parmi les neuf (9);
  - OU **Il remporte trois (3) bornes consécutives**.
- Pour **remporter une borne** :
  - En cas d'attaque complète, l'**attaque la plus forte** remporte.
    - *Attaque complète : trois (3) cartes clan de chaque côté*
  - On peut remporter une borne **si on peut démontrer que l'adversaire n'a plus aucune chance de battre l'attaque** d'un joueur **vu les cartes en jeu sur la table** (*pas dans les mains*).

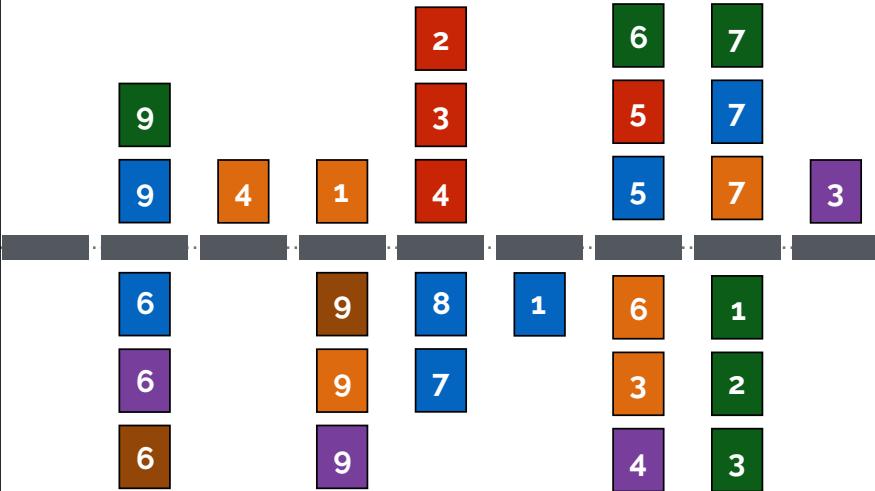


# Force des attaques

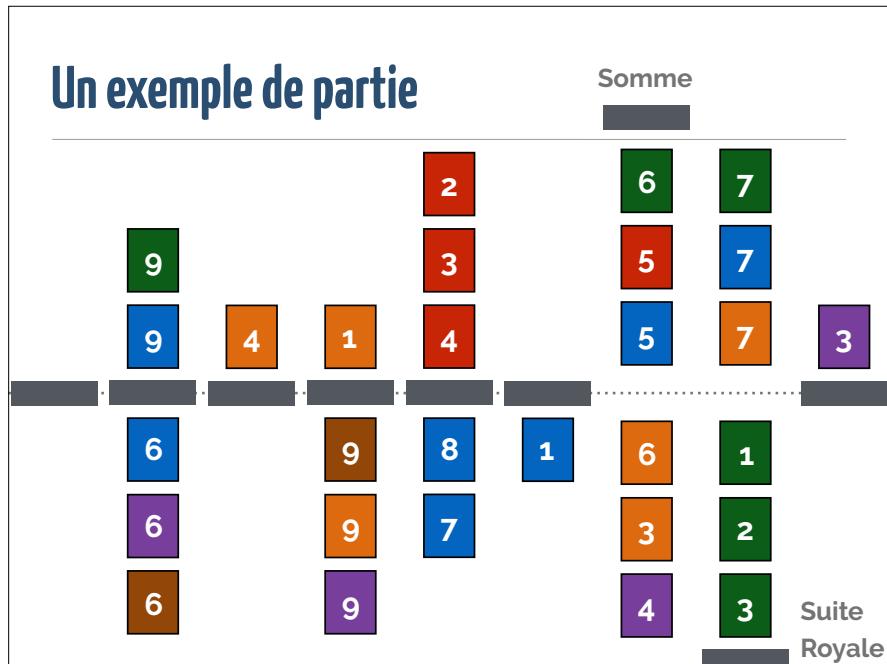


Toute ressemblance avec un Jeu de Poker est fortuite ... 😊

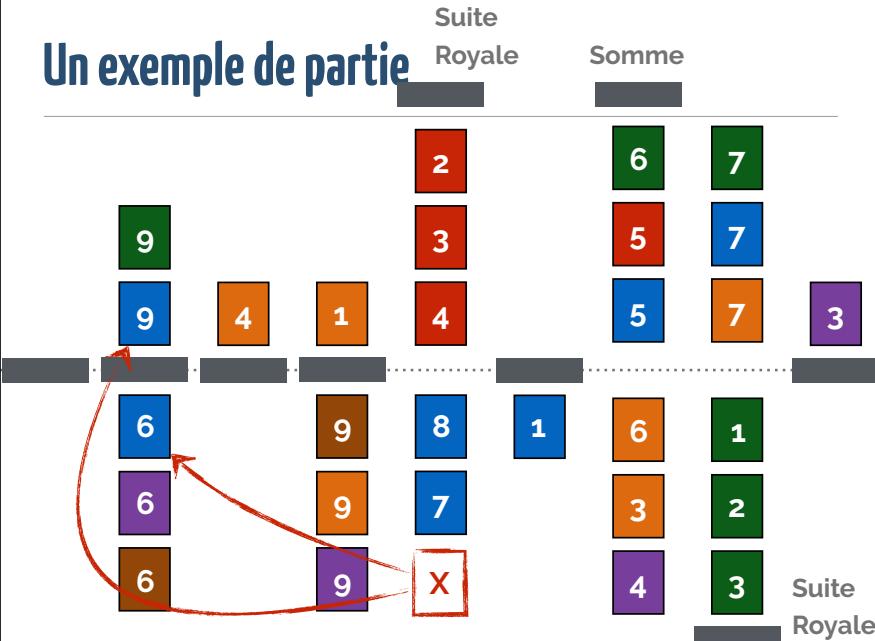
## Un exemple de partie



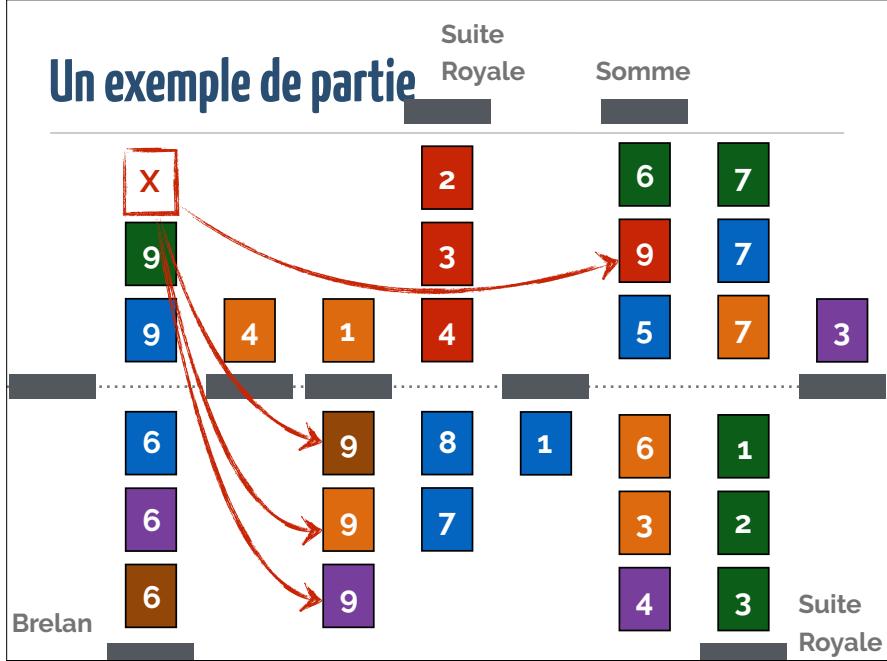
## Un exemple de partie



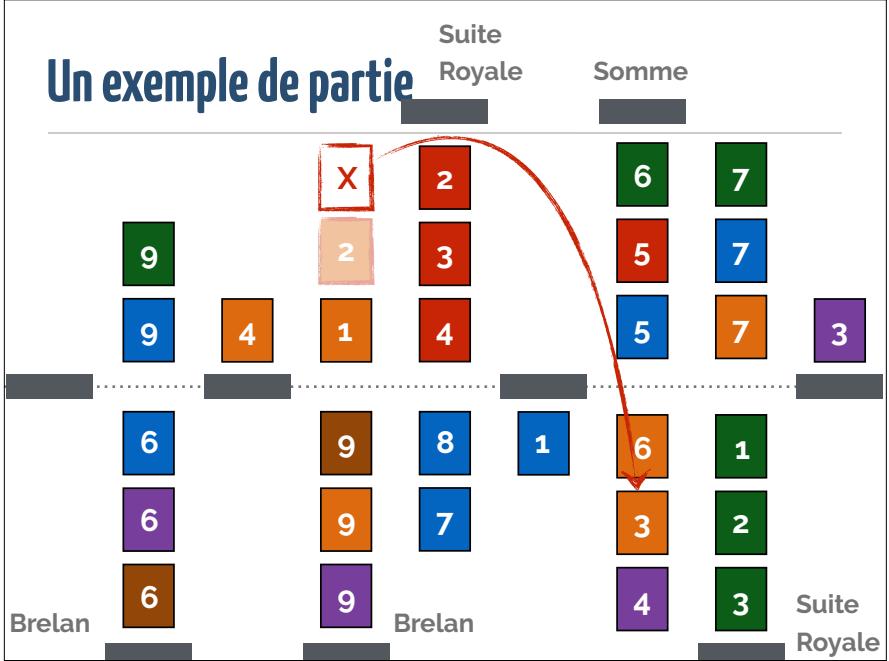
## Un exemple de partie



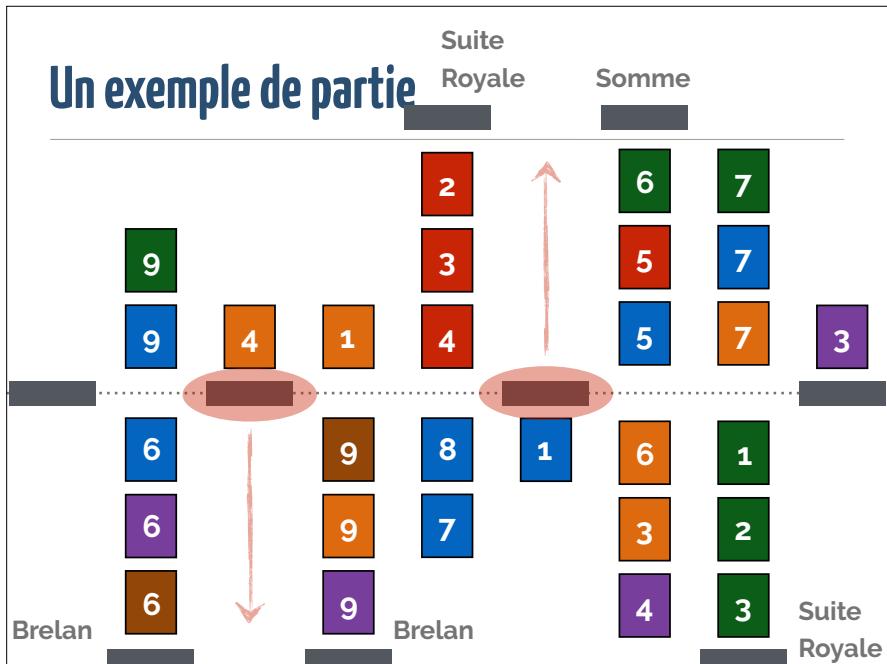
## Un exemple de partie



## Un exemple de partie



## Un exemple de partie

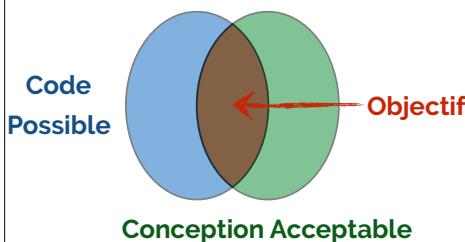


Encapsulation

2

## Pourquoi “Encapsuler” ?

- On **décompose** un système en objets pour :
  - maîtriser la **complexité** globale (*diviser pour mieux régner*)
  - travailler de manière **séparée** sur les concepts (*conflits git*)
- Tout système décomposé est-il bon ?



Éviter au maximum le problème dit de “*ignorant surgery*”

## Exemple de conception des cartes

- Description extraite depuis la **spécification** :
  - Il y a neufs (9) cartes clan par couleurs
  - Il y a six (6) couleurs de cartes
- On **encode** l'information dans un **entier card**  $\in [0, 9 \times 6[$ 
  - Pour trouver la **valeur** de la carte : On **divise** par neuf, plus un
    - `public static int value(int card) { return (card / 9) + 1; }`
  - Pour trouver la **couleur** : on prend le **modulo** neuf
    - `public static int color(int card) { return card % 9; }`

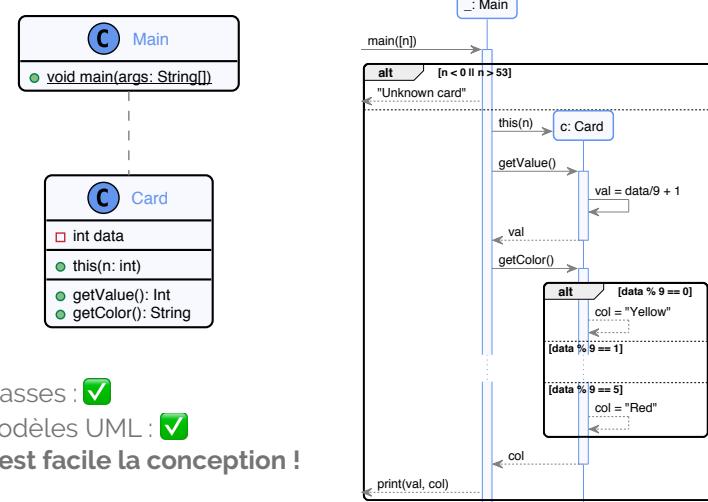
“Hé tavu, il est opti mon code...”

## Quels sont les problèmes ?

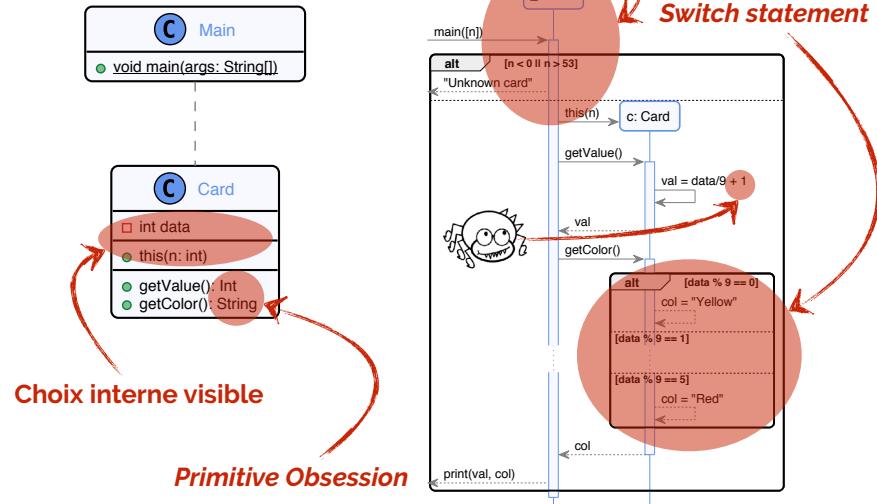
- Représentation dans le **code différente** du **domaine**
  - Vous jouez avec des cartes, ou avec des nombres entiers ?
- Représentation **couplée** à l'implémentation
  - Dans tout le programme, on utilise le type *int* pour une carte
- On peut **corrompre** la structure de donnée aisément
  - Est-ce que -42 est une carte valide ? Pourtant c'est un *int* ...

On parle de “**Primitive Obsession**” quand un code utilise de manière inadéquate des types primitifs au lieu d'objets du domaine. C'est une **représentation possible** dans le code, mais **pas acceptable** du point de vue de la conception.

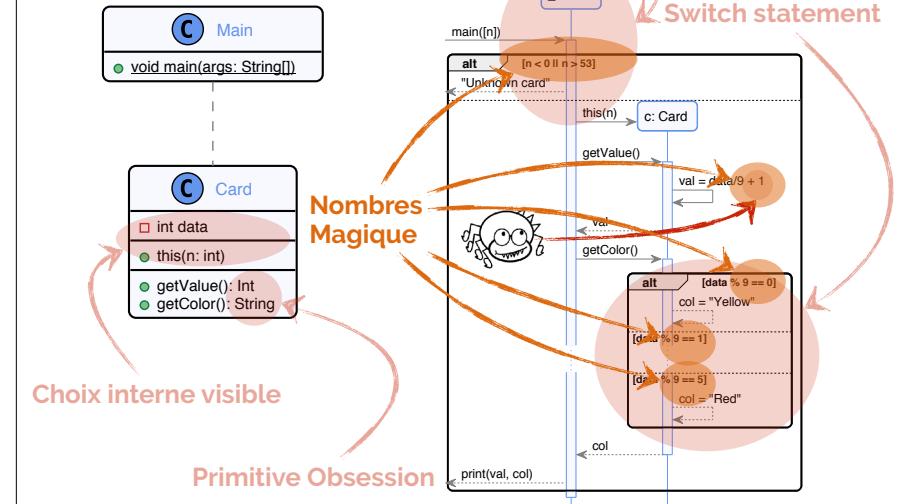
## On va “encapsuler” dans une classe alors ...



## Quels sont les problèmes ?



## Quels sont les problèmes ?

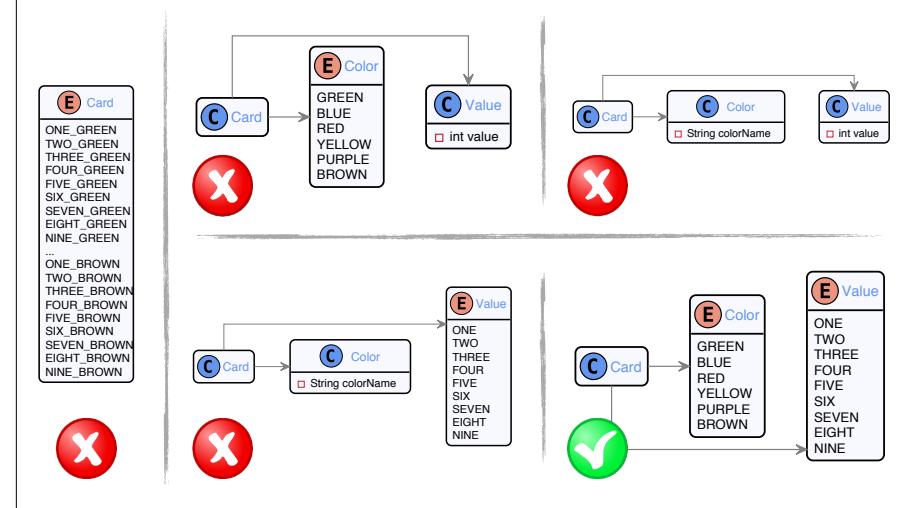


# Énumérations versus Classes

- La plupart des langages propose la **notion d'énumération**
    - Souvent avec le mot clé "enum"
  - Quelle **différence** avec une classe ?
    - Une **énumération** repose sur l'**hypothèse du monde fermé**
      - $\approx$  Extension (*Ensemble des nombres pairs < 10 = [0, 2, 4, 6, 8]*)
    - Une **classe**, à l'inverse, fonctionne en **monde ouvert**
      - $\approx$  Intention (*Ensemble des nombres pairs = { n | n % 2 == 0 }*)
  - Dans notre cas, **quelles seraient les énumérations** ?

On verra dans quelques diapos qu'on peut implémenter l'une de ces notions avec l'autre

## Exploration de l'espace des solutions



# “Est-ce que ça sera à l’examen ?”

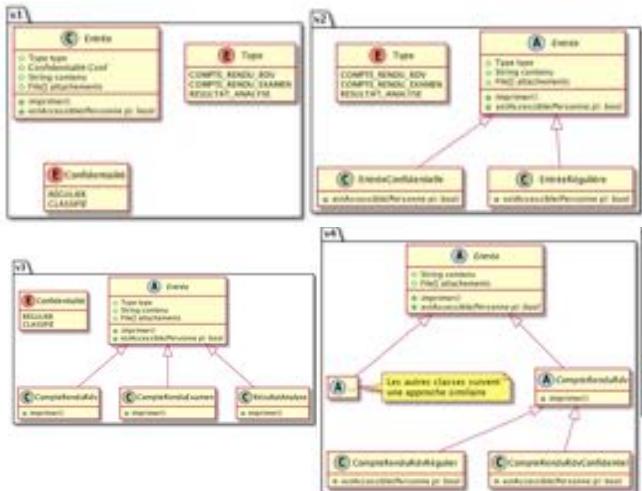


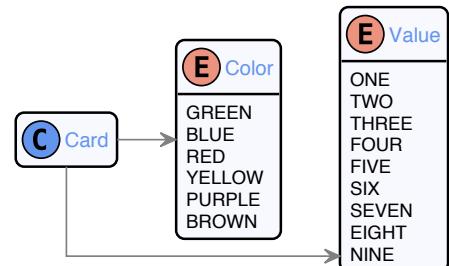
Figure 1. Modélisations différentes du concept d’Entrée dans le DMP.

OUI  
*Intra H19*

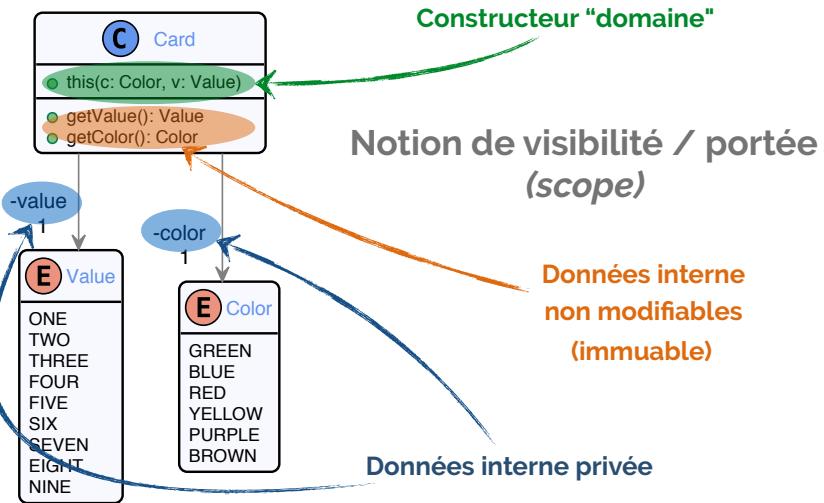
```
class Card {
    Value value;
    Color color;
}

enum Value {
    ONE, TWO, THREE,
    FOUR, FIVE, SIX,
    SEVEN, EIGHT, NINE
}

enum Color {
    GREEN, BLUE, RED,
    YELLOW, PURPLE, BROWN
}
```



## Renforcement de l’encapsulation



## Les énumérations sont du “sucre syntaxique”

```
enum MyEnum {
    VALUE_1, VALUE_2
}

class MyEnum {
    public static final MyEnum VALUE_1 = new MyEnum();
    public static final MyEnum VALUE_2 = new MyEnum();

    private MyEnum() {}
}
```

The diagram compares UML and Java enum representations. It shows:
 

- An UML enum 'MyEnum' with two values: 'VALUE\_1' and 'VALUE\_2'.
- A Java class 'MyEnum' with a private constructor and static final fields 'VALUE\_1' and 'VALUE\_2'.
- A note: "'final'" is ALL\_CAPS by convention in UML.
- A note: "presque équivalent" (almost equivalent).
- A note: "La visibilité est un mécanisme puissant, quand elle est bien utilisée" (Visibility is a powerful mechanism, when it is well used).

## Les cartes c'est correct. Quid de la pioche ?

- La pioche contient toutes les cartes au début
  - Il existe une relation d'ordre entre les cartes dans la pioche
  - C'est une **relation de précédence** (*selon le mélange*)
- Ça ressemble quand même beaucoup à une **Liste de Cartes**
  - *(On pourrait aussi envisager autre chose, p.-ex. un tableau)*

```
public static void main(String[] args) {  
    List<Card> drawPile = new ArrayList<>();  
    drawPile.add(new Card(ONE, GREEN));  
    // ...  
    drawPile.add(new Card(NINE, BROWN));  
    // ...  
    System.out.println(drawPile.get(2).color.toString())  
}
```

## “Oops, I did it again”



```
public static void main(String[] args) {  
    List<Card> drawPile = new ArrayList<>();  
    drawPile.add(new Card(ONE, GREEN));  
    // ...  
    drawPile.add(new Card(NINE, BROWN));  
    // ...  
    System.out.println(drawPile.get(2).color.toString())  
}
```

- La représentation dans le code diffère de celle du domaine;
- Pour passer à un tableau, il faudrait changer toutes les références;
- On peut corrompre la pioche (p.-ex. doublons) par accident.

## Encapsulons la liste dans une CardPile

```
class CardPile {  
    public List<Card> cards = new ArrayList<>();  
}  
  
public static void main(String[] args) {  
    CardPile drawPile = new CardPile();  
    drawPile.cards.add(new Card(ONE, GREEN));  
    // ...  
    drawPile.cards.add(new Card(NINE, BROWN));  
    // ...  
    System.out.println(drawPile.cards.get(2).color.toString())  
}
```

- La représentation dans le code **définitivement diffère** de celle du domaine;
- Pour passer à un tableau, il faut **changer toutes les références**;
- On peut **corrompre** la pioche (p.-ex. doublons) par accident.

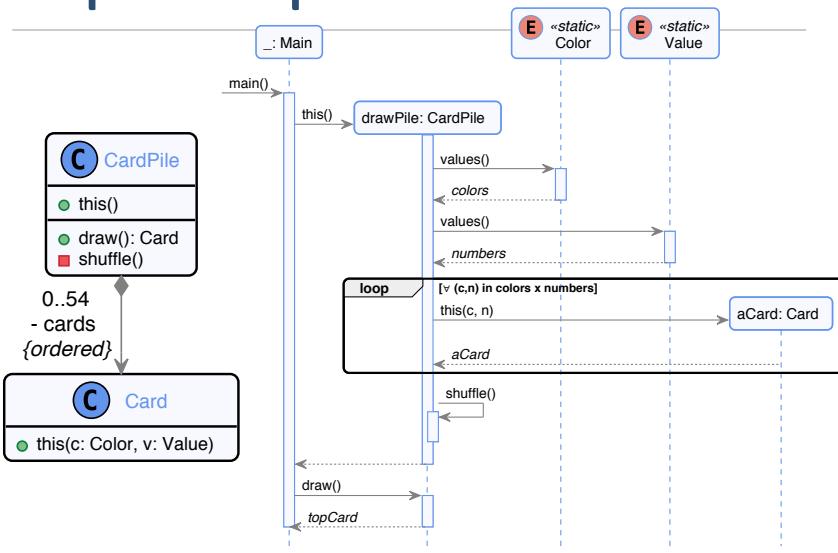
## Quels sont les problèmes ?

- Il y a une **fuite d'abstraction** !
  - *On expose à l'extérieur un choix interne (la liste)*
- Il faut **cacher ce choix interne** aux consommateurs
- On doit réfléchir aux **services offerts** par la CardPile
  - *“La pioche contient toutes les cartes au début”*
  - *“A son tour, un joueur [...] pige une carte dans la pioche”*



On peut réparer la fuite d'abstraction en jouant sur la **visibilité ET** sur l'**interface** de la classe CardPile.

## Une pioche encapsulée



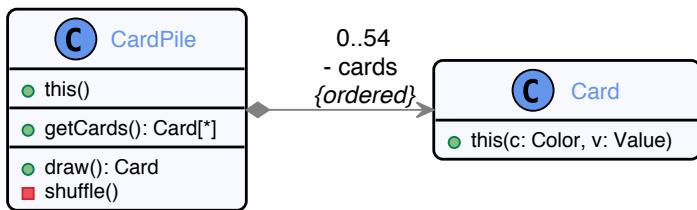
## Quels sont les problèmes

- La classe est **bien encapsulée**.
  - Accès contrôlés par l'**interface** publique
  - Rien ne dépasse (**attributs privés**)
- **N'est-ce pas un peu trop limité ?**
  - **Et si on voulait inspecter la pioche ?**
- Est-ce **nécessaire** ?
  - Attention à l'**over-engineering** (*et si ?*)

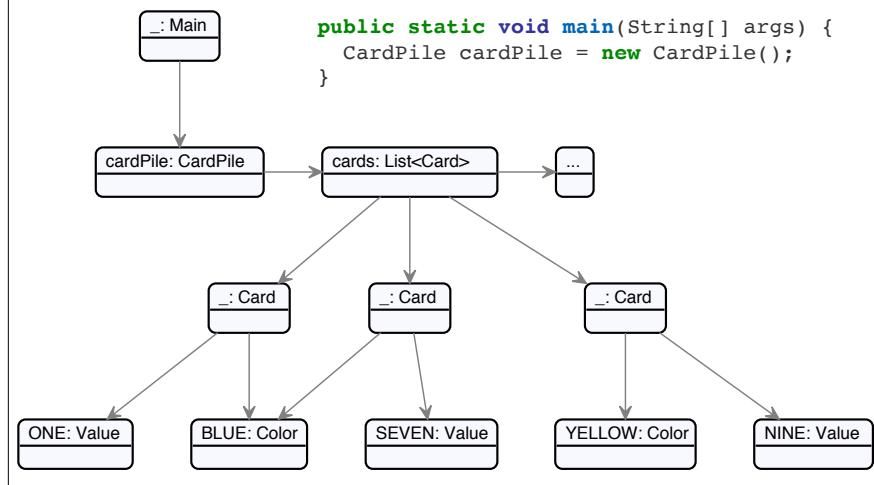
## Solution Naive : donner un accesseur

- Il faut accéder à la liste de carte
- Il suffit de la renvoyer avec une méthode getCards() !

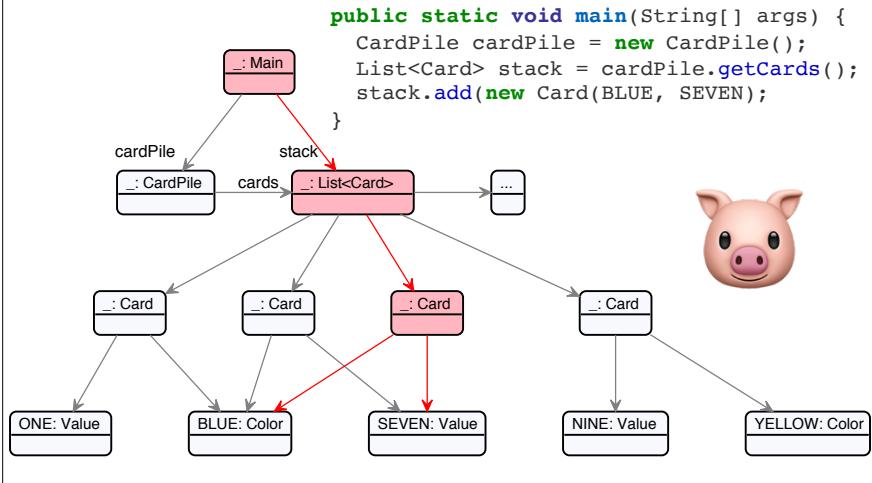
```
public void List<Card> getCards() {
    return this.cards;
}
```



## Diagramme d'objets = Cliché à l'exécution



## Fuite de données !



## “Inappropriate Intimacy”

- On parle d'**intimité inapproprié** quand “*une classe passe trop de temps à fouiller dans les “parties privées” d’une autre classe*”\*.
- On s’était mis d’accord que la **List<Card>** était un choix interne
  - En l’exposant, on “ouvre la porte à toutes les fenêtres”
- Définir un accesseur (get\*) provoque une fuite de données**
  - On aurait le même problème avec un modificateur (set\*)
- Dans l’absolu, ce qui est privé doit rester privé.**

\* Martin Fowler

## “Ce qui est privé doit rester privé”

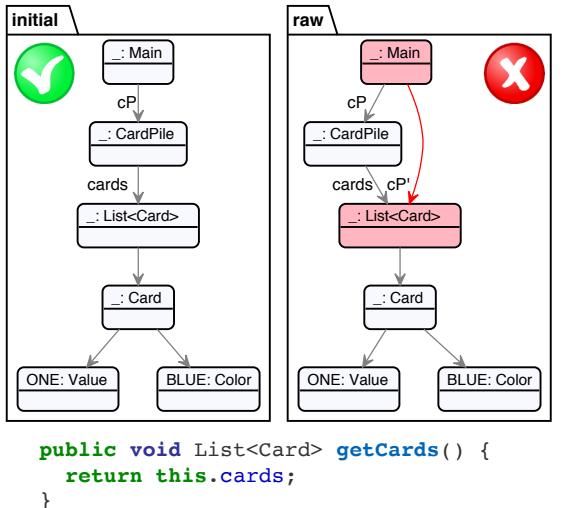
- Comment faire si ce qui est encapsulé n'est jamais accessible ?
  - En fait, “**ça dépend**” ... !
- On peut **retourner une donnée privée** (*sans trop de soucis*) :
  - Si celle-ci est **immutable** (p.-ex. *String*, type énuméré)
  - Si on en **fait une copie** avant de l'envoyer
- On peut **contrôler l'accès** à la donnée privée :
  - En donnant une méthode “*get(i: Int): Card*” dans la *CardPile*

**Mais dans tous les cas ça brise (un peu) l'encapsulation**

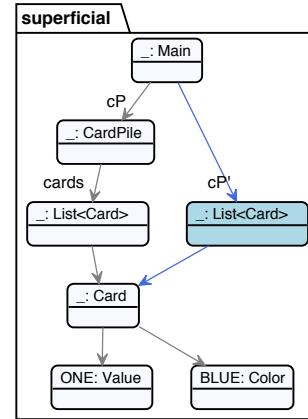
## Retourner une copie de la donnée

- “Pour tout **problème complexe**, il existe une **solution simple, claire, directe**, et **fausse**”. Albert Einstein.
  - La copie peut être **superficielle** ou **profonde**
  - La copie peut être en **lecture seule** (p.-ex, “enrobée”)
  - Le choix est complexe, mais l’impact est vraiment fort.
- Comment copier** ?
  - Méthode **helper** : *copy(cp: CardPile): CardPile*
  - Constructeur** par copie : *CardPile cp' = new CardPile(cp)*
  - Méthode de **clonage** : *CardPile cp' = cp.clone()*
- Cloner à un coût, et si on appelle l’accesseur dans une boucle** ?

## Exemples de mécanismes de copie d'objets



## Exemples de mécanismes de copie d'objets



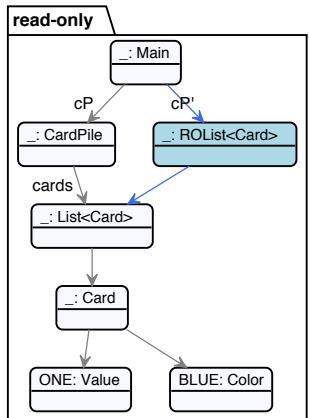
```

public void List<Card> getCards() {
    return new ArrayList<>(this.cards);
}

```

- Constructeur par copie
- On peut modifier cP' sans modifier cP
- Si Card est *mutable*
  - On a juste décalé le problème

## Exemples de mécanismes de copie d'objets



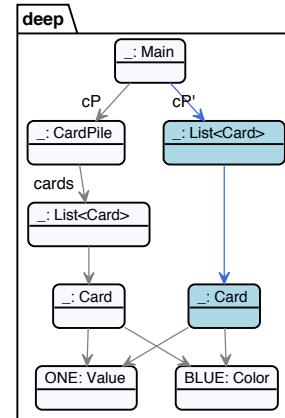
```

public void List<Card> getCards() {
    return Collections.immutableList(this.cards);
}

```

- Méthode *helper*
- On produit un *wrapper* de la liste
  - Avec la *même interface* que la liste
  - Mais qui *intercepte* les modifications
    - throw UnsupportedOperationException
- On peut oublier qu'on est *read-only*
  - Erreur à l'exécution, c'est pas fou

## Exemples de mécanismes de copie d'objets



```

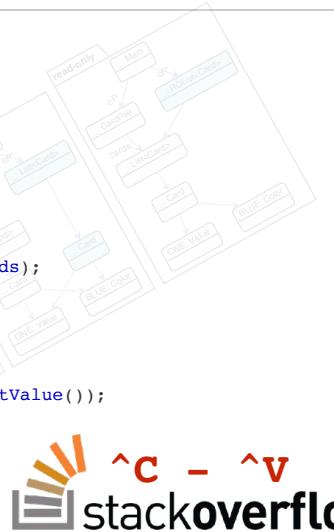
public void List<Card> getCards() {
    List<Card> result = new ArrayList<>();
    for(Card c: this.cards) {
        Card copy = new Card(c.getColor(), c.getValue());
        result.add(copy);
    }
    return result;
}

```

- Si Card est mutable, il faudrait **copier**
  - *Quelle profondeur de copie ?*
- En dupliquant tout, on a de la marge
  - *Quelle place en mémoire ?*

## Bref, comprenez ce que vous utilisez !

```
public void List<Card> getCards() {  
    return this.cards;  
}  
  
public void List<Card> getCards() {  
    return new ArrayList<>(this.cards);  
}  
  
public void List<Card> getCards() {  
    return Collections.immutableList(this.cards);  
}  
  
public void List<Card> getCards() {  
    List<Card> result = new ArrayList<>();  
    for(Card c: this.cards) {  
        Card copy = new Card(c.getColor(), c.getValue());  
        result.add(copy);  
    }  
    return result;  
}
```



## Types & Interfaces



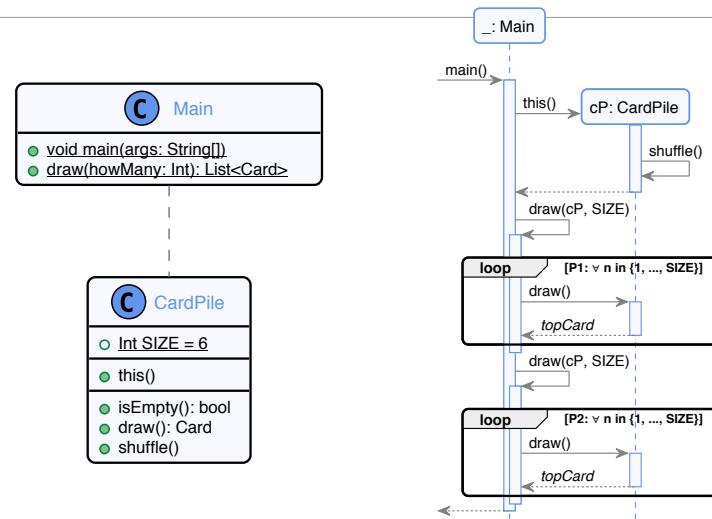
## Différence entre “Type” et “Classe”

- Un “**Type**”, c'est :
  - “une annotation qui aide à garder le code cohérent” (Privat 2019)
  - un contrat entre consommateur et fournisseur de service
- Une “**Classe**” :
  - “c'est une capsule qui décrit des objets similaires” (Privat 2019)
  - peut être conforme à plusieurs **types**\*

```
public void List<Card> getCards() {  
    List<Card> result = new ArrayList<>();  
    for(Card c: this.cards) {  
        Card copy = new Card(c.getColor(), c.getValue());  
        result.add(copy);  
    }  
    return result;  
}
```

\* on ira plus loin (substituabilité) au prochain cours

## Distribuer les six premières cartes



## Découpler l'interface de l'implémentation

```
public static List<Card> drawCards(CardPile cp, int howMany) {  
    List<Card> result = new ArrayList<>();  
    for (int i = 0; i < howMany; i++) {  
        if (cp.isEmpty()) {  
            return result;  
        }  
        result.add(cp.draw());  
    }  
    return result;  
}
```

- Pour piger "n" cartes, il faut :
- Savoir s'il en reste
  - En piger une

(Mais ça pourrait être autre chose que des "cartes" en fait.)

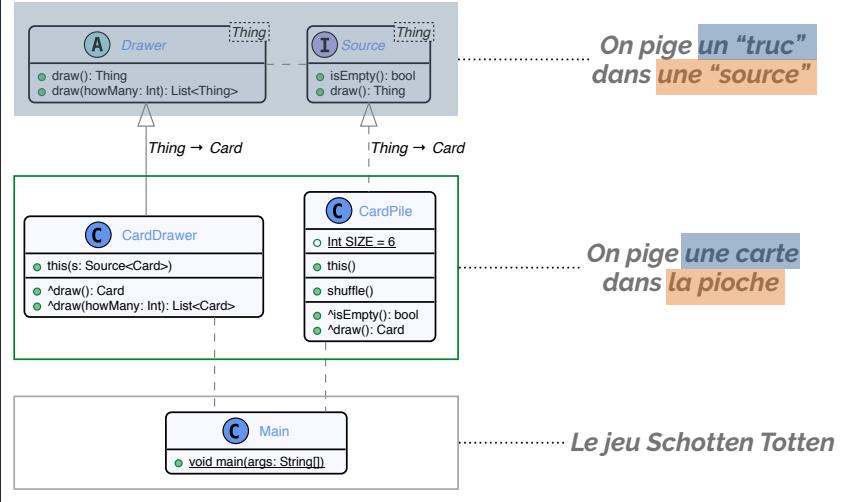
[ItSDWJ, p41]



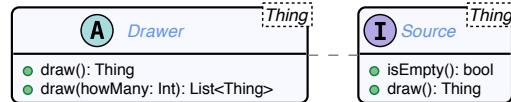
## Couplage faible

## Extensibilité

## Abstraction du comportement réutilisable



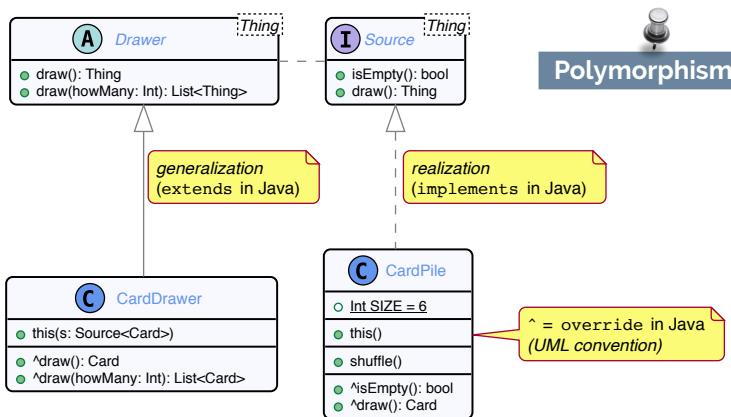
## Classe Abstraite ou Interface ?



**Drawer** est une classe **Abstraite** :  
On sait décrire son comportement en l'état des services offerts par **Source**

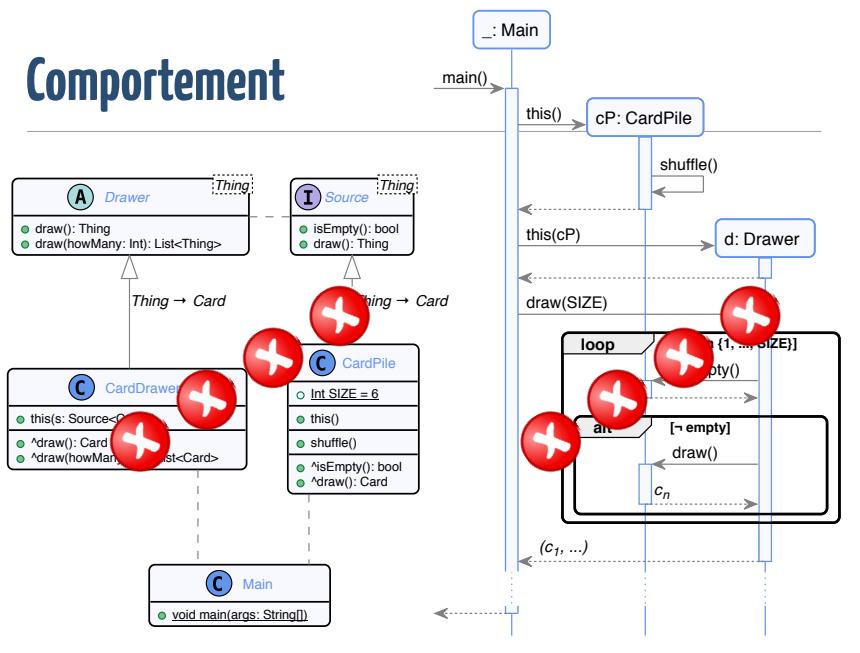
- Une **Interface** :
  - Ensemble de services fournis par les entités conformes à ce Type
- Une **classe abstraite** :
  - Comportement abstrait que l'on sait décrire en l'état
  - Et une interface avec des **méthodes "par défaut"** ?
    - C'est avant tout une interface + sucre syntaxique

## Généralisation et Réalisation



Dans les deux cas, la relation implique une relation de sous-typeage

## Comportement



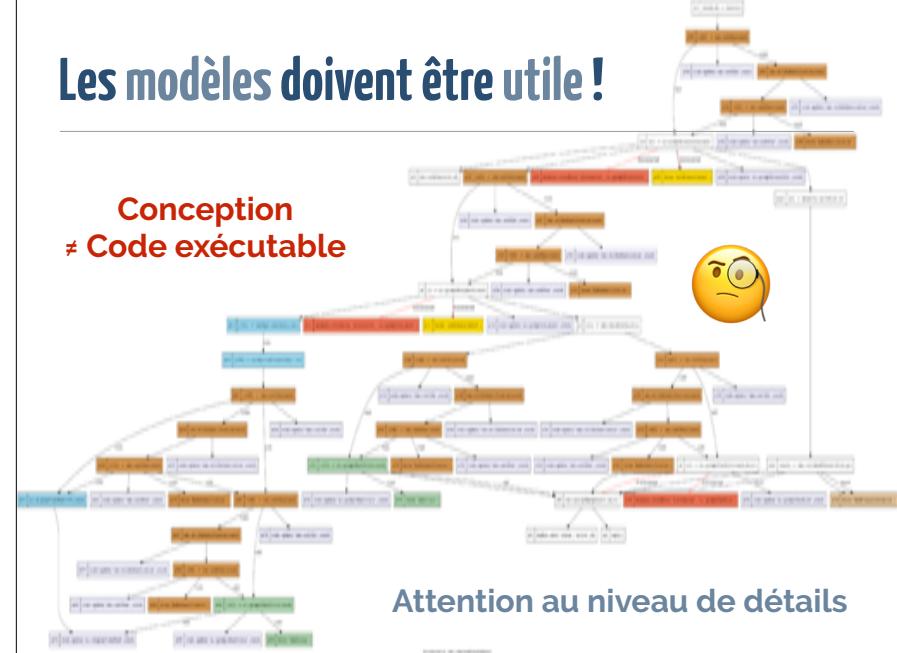
## Généralisation VERSUS Réalisation

- La **généralisation** est une relation **taxinomique**
  - Taxinomie = **classification** (p.-ex. les espèces en biologie)
- Relation entre un élément **général** et un autre **spécifique**
  - Un chat est un mammifère. Une carotte est un légume.
- La **réalisation** est une relation de **mise en oeuvre**
  - C'est un lien entre une **spécification** et sa **réalisation**
- Une collection sait se trier. Une pioche sait piger une carte.

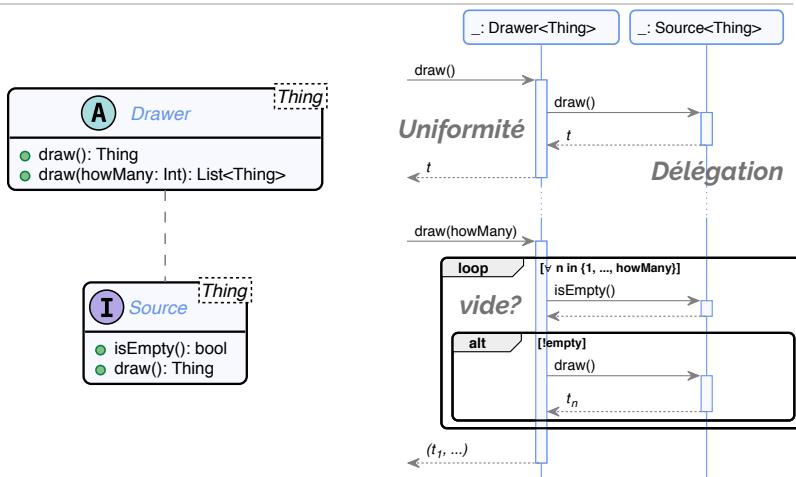
*En pratique vous utilisez usuellement très mal ces deux relations.  
On parlera d'héritage (généralisation) au prochain cours.*

## Les modèles doivent être utile !

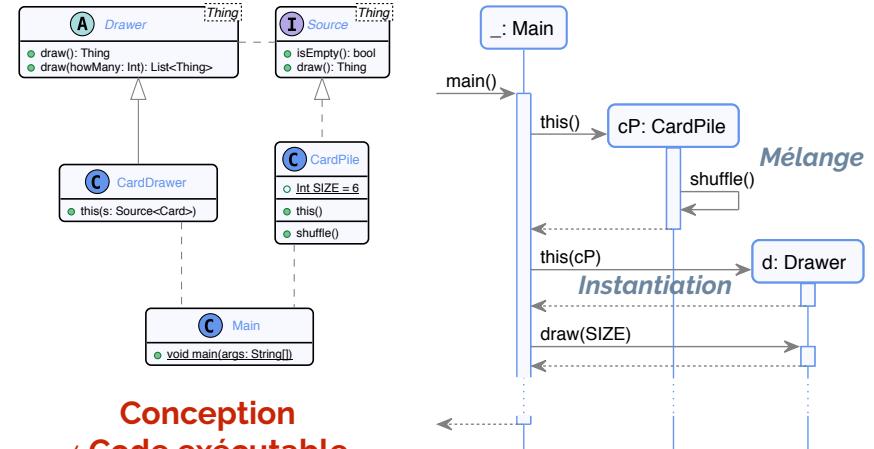
Conception  
≠ Code exécutable



## Séparation des préoccupations !!



## Les modèles doivent être pertinents !



## Questions de conception

- Est-ce que j'ai besoin de créer une nouvelle interface ?
  - Ça dépend. 😊
- Que devrait spécifier cette interface ?
  - Ça dépend. 😊
- Ne pas tomber dans le piège de l'**over-engineering**
- Ne pas créer trop de **dette technique** non plus
- C'est un processus **essais-erreurs** ⊕ **expérience**
  - Les solutions dogmatique sont souvent inutilisable en pratique
  - Les séances de laboratoires INF-5153 sont faites pour ça



# Types & Interfaces (la suite)



le retour

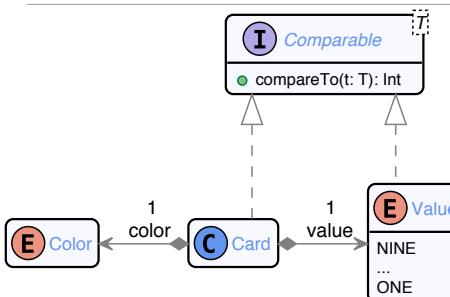
## Le contrat de comparaison

- 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 ( $\neq$  égalité)
  - $c_1.compareTo(c_2) = 0 \Leftrightarrow c_1 \equiv c_2$

## Un exemple utile : La comparaison dans Java

- Java fournit des "services" utile sous la forme d'interface "`-able`"
  - `Cloneable, Serializable, Comparable, Iterable, Runnable, Callable, Observable, Closeable, ...`
- Une classe peut implémenter "`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*

## Comparer deux cartes de Schotten Totten



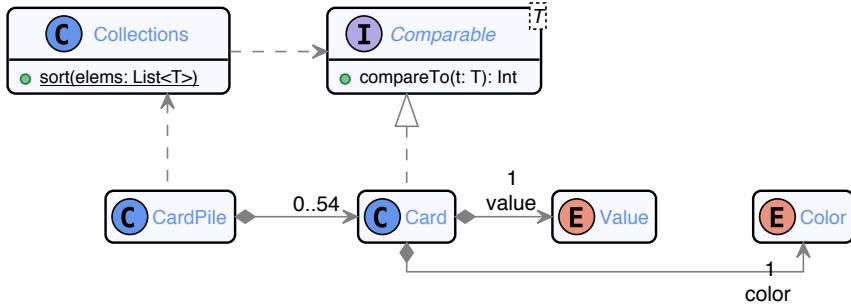
Les énumérations Java sont gratuitement comparable (selon leur ordre de déclaration)

```
public class Card implements Comparable<Card> {
    private Value value;

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

Classes d'équivalence :  
Cartes de même valeur et de couleurs ≠

## Bénéfices de l'utilisation de Comparable<T>



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

[ItSDwJ, p46]

## Choisir la responsabilité ...

- 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}

## 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. C'est le meilleur  **compromis**

## Implémentation “Comparator”, #1

```
public class Card {
    private Value value; Classe interne, accès aux attributs privés
    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());
    }
}
```

[ItSDwJ, p48]

## Implémentation “Comparator”, #2

```
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 quand on en a besoin  
    • pas réutilisable  
    • Repose sur l'interface publique pour comparer  
}
```

[ItSDwJ, p49]

## Implémentation “Comparator”, #3

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

### Lambda-expression :

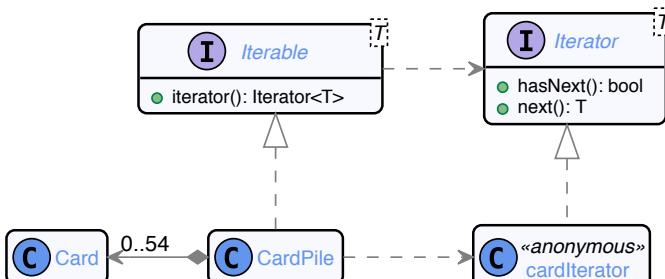
- Sucré syntaxique sur l'instantiation de classes anonymes
- Attention à la lisibilité du code.

[ItSDwJ, p49]

## Accéder aux cartes de la pioche

- Quels sont les **services** dont on a besoin ?
  1. **Itérer** sur le contenu de la pioche
  2. **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(); }  
}
```

[ItSDwJ, p51]

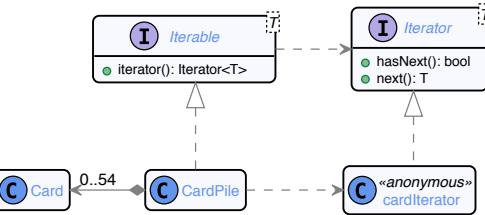
## Avantages des abstractions réutilisables

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

*La construction de langage "foreach"  
repose sur un Iterable en Java*

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

## Réutilisation Gratuite !



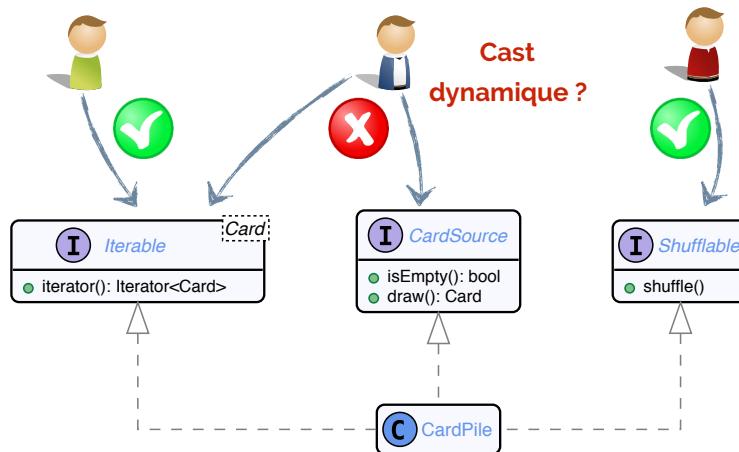
```
public class Main {  
  
    public static void main(String[] args) {  
        CardPile cP = new CardPile();  
        for(Card c: cP) {  
            // ...  
        }  
    }  
}
```



## Principe de ségrégation des interfaces

- Faits : On interagit avec les objets via leurs interfaces
- Définition : "**Interface Segregation Principle**"
  - "*Many client specific interfaces are better than one general purpose interface*" [Design principles, Martin, 2000]
  - Ou encore : "*Un consommateur de service ne doit pas dépendre d'interfaces dont il n'a pas besoin*" [ItSDwJ, p56]
  - C'est le I de l'acronyme SOLID
- Quels sont les services proposés par la Pile de carte ?
  - Se mélanger, Fournir des cartes, Pouvoir être parcourue

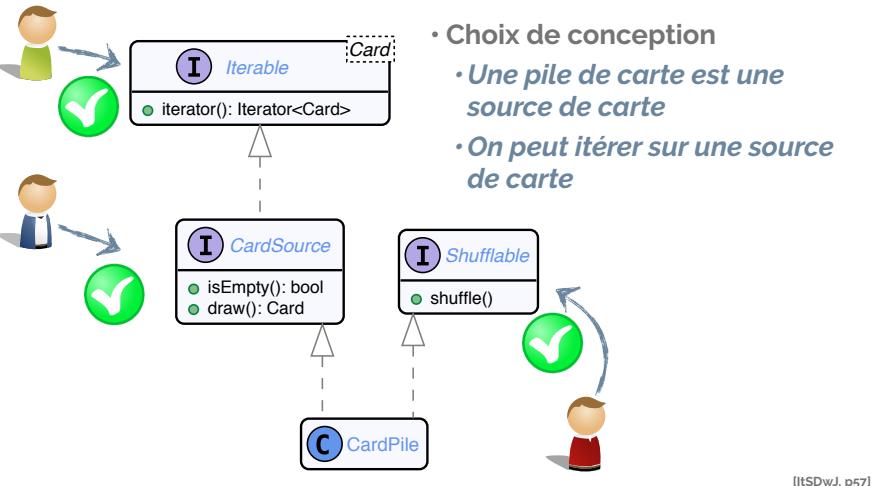
## Proposition #1



[ItSDwJ, p57]

## Proposition #2

(solutions au cas par cas ...)



## État des Objets



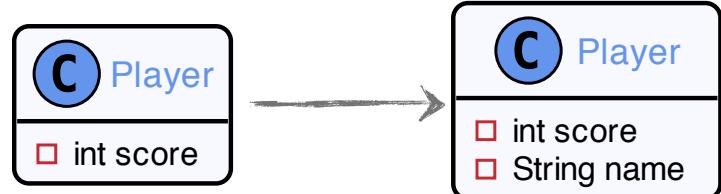
## Digression : Principes versus Paradigmes

- Les **principes de conception** 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 **modèles** paradigmes changent, pas les principes.

## Pourquoi s’intéresser à l’état ?

- Les concepts manipulés ont de **grands ordres de grandeurs**
- Considérons la classe **Player**, qui contient le score du joueur
  - Le score est un Int, **on peut créer  $2^{32}$  instances ≠ de Player !**
  - Et si on rajoute le nom du joueur ?
    - Le **nombre d’instances** ≠ devient **infini** (*limité par la mémoire*)



## La Pile de Cartes

- **Rappel** : "les modèles doivent être pertinents et utile"

- On considère la pile de cartes de **Schotten Totten**

• Il y a  $A_k^n = \frac{n!}{(n-k)!}$  **arrangements sans répétitions**

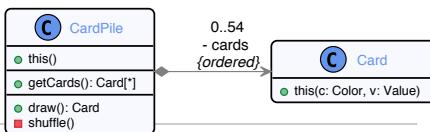
de  $k$  cartes parmi  $n$  dans la pile de cartes (ordonnée)

- La pile peut contenir de zéro (0) à cinquante quatre (54) cartes

- Donc **on a modélisé**  $N = \sum_{k=0}^{54} A_k^{54}$  **états possible** de la pioche

- $N = 6,27 \times 10^{71}$  **pioches différentes !**

- **On estime à  $10^{80}$  le nombre d'atomes dans l'univers**



## On considère uniquement les états utiles

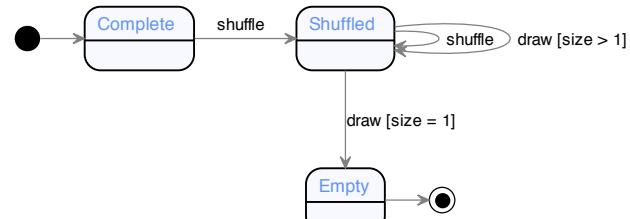
- L'**utilité** peut être à deux niveaux :

- **Technique** : "piger une carte dans la "pile vide" ?"

- **Logique d'affaire** : "la pioche est mélangée"

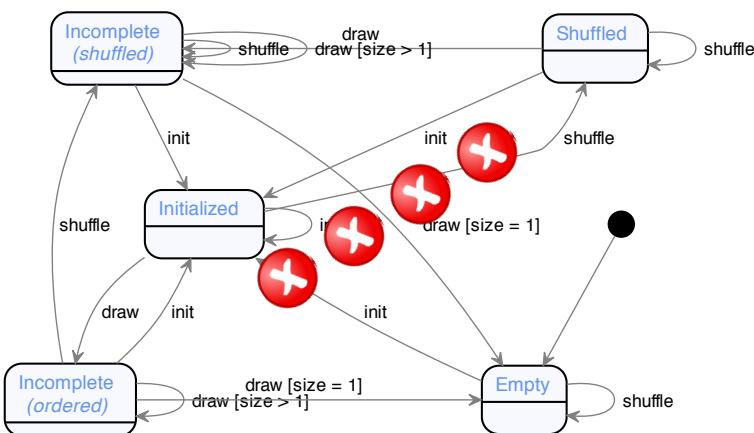
- C'est une modélisation des **états "abstraits"** du système

- Ça n'a de sens que pour les objets **avec états et mutables**



## Attention au niveau de détail !

[ItSDwJ, p66]



Un diagramme d'état n'est pas un automate fini déterministe

## Problèmes classiques de conception

- Utiliser un State Diagram comme **un flot de données**

- *Un état n'est pas une étape de calcul sur des données !*

- Prévoir beaucoup d'états inutiles

- *Over-engineering*, ou encore "*Speculative Generality*"

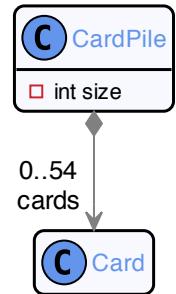
- *Ça peut toujours servir ... Non ! (Dette technique)*

- Augmenter l'espace d'état pour "**optimiser**"

- *Souvent l'optimisation est minime*

- *C'est source de bugs dans le futur*

- Problème appelé "**Temporary Fields**"



## Le problème de la nullité

- Comment représenter l'absence d'information ?
  - C'est le même problème que celui du "zéro" en math !
  - Il faut une info concrète représentant l'absence d'info ...
- En Java, pour représenter l'absence d'un élément de type T :
  - Utiliser `null`
  - Concevoir notre propre zéro pour le type en question:
    - p.-ex. une valeur spéciale, un objet dédié.
  - Utiliser un `Optional<T>`



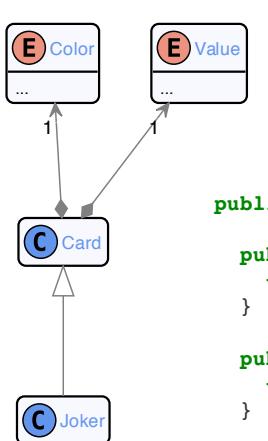
## Exemple : Carte Spéciale Joker

- La carte spéciale joker remplace n'importe quelle carte clan lors d'une attaque.
- On choisit son contenu au moment où on la pose.
  - Elle n'a donc ni `Value` ni `Color` jusqu'à ce qu'elle soit jouée.
- Comment faire avec la conception actuelle ?



[Illustration par Djib]

## Un héritage "naïf" ne résout rien



```
public class Joker extends Card {  
  
    public Color getColor() {  
        throw new UnsupportedOperationException();  
    }  
  
    public Value getValue() {  
        throw new UnsupportedOperationException();  
    }  
}
```

## Utiliser null comme zéro

NullPointerException

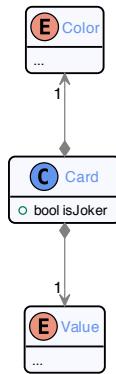


```
public class Card {  
    private Value value = null;  
    private Color color = null;  
    private boolean isJoker = false;  
  
    public Card(Value v, Color c) {  
        this.value = v;  
        this.color = c;  
    }  
    public Card() {  
        this.isJoker = true;  
    }  
}
```

- On rajoute un attribut booléen "isJoker"
- Le constructeur vide met `null` dans les champs `value` et `color`

## Choisir une valeur nulle

### Usurpation d'identité

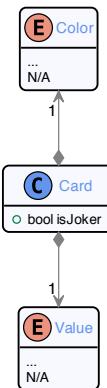


```
public class Card {  
    private Value value;  
    private Color color;  
    private boolean isJoker;  
  
    public Card(Value v, Color c) {  
        this.value = v;  
        this.color = c;  
        this.isJoker = false;  
    }  
  
    public Card() {  
        this.value = Value.ONE;  
        this.color = Color.BROWN;  
        this.isJoker = true;  
    }  
}
```

- Le joker est un "1 marron". Parce que c'est comme ça.
- Et si on oublie de tester si c'est un joker ? Ça sera un "1 marron".

## Créer la valeur nulle

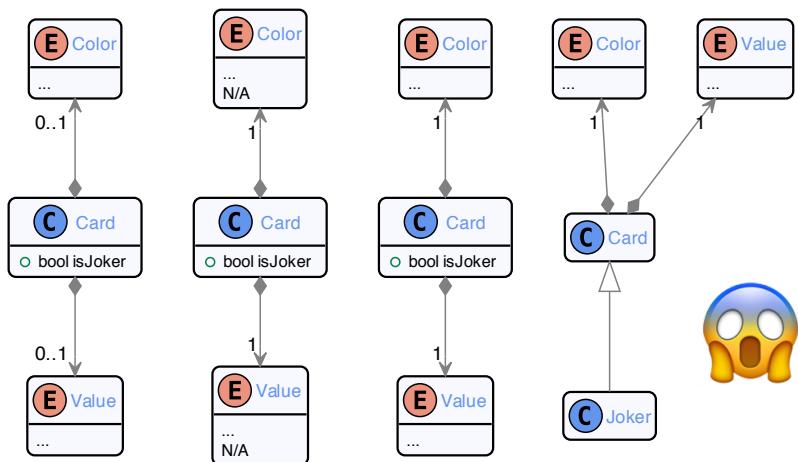
### Technique versus Affaire



```
public class Card {  
    private Value value;  
    private Color color;  
    private boolean isJoker;  
  
    public Card(Value v, Color c) {  
        this.value = v;  
        this.color = c;  
        this.isJoker = false;  
    }  
  
    public Card() {  
        this.value = Value.NA;  
        this.color = Color.NA;  
        this.isJoker = true;  
    }  
}
```

- On rajoute une valeur "N/A" dans l'énumération
  - Pour l'initialisation de la pioche on bouclait sur les énumérations ...

## Différence subtile, mais GROS impact !



Est-ce que ça sera à l'examen ? OUI.

## Gérer l'absence de valeur : Optionnel

- Construction présente dans beaucoup de langages

- P.-ex. Python, Haskell, Scala, ...

- Pourquoi `null` en Java de 1995 à 2014 ? (19 ans !!)

- "I call it my **billion-dollar mistake**. It was the invention of the null reference in 1965.... I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement."

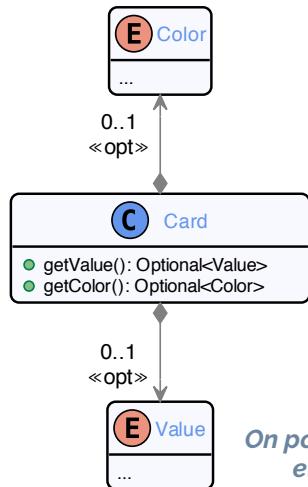
- Tony Hoare, London QCon 2009 [Java 8 in Action, ItSDWJ pg0]*

- Définition du **type optionnel** : `Optional<T> = T ∪ None`

- Le **type Optional<Int>** représente n'importe quel **Int**, ou **None**

## Optionnel visible

Fuite des optionnels dans le code client.



```
public class Card {
    private Optional<Value> value;
    private Optional<Color> color;

    public Card(Value v, Color c) {
        this.value = Optional.of(v);
        this.color = Optional.of(v);
    }

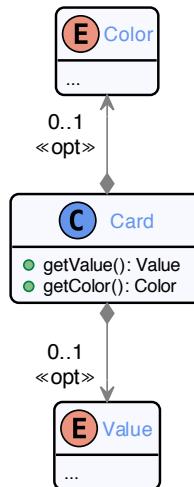
    public Card() {
        this.value = Optional.empty();
        this.color = Optional.empty();
    }

    public boolean isJoker() {
        return this.value.isPresent();
    }
}
```

On pourrait aussi encapsuler "null" dans l'attribut et retourner des Optional dans les getters.

## Optionnel invisible

NoSuchElementException plutôt qu'une NPE plus tard



```
public class Card {
    private Optional<Value> value;
    private Optional<Color> color;

    public Card(Value v, Color c) { ... }

    public Card() { ... }

    public Color getColor() {
        return this.color.get();
    }

    public Value getValue() {
        return this.value.get();
    }

    public boolean isJoker() {
        return this.value.isPresent();
    }
}
```

Attention aux Optionnels et à la Sérialisation

## Unicité des Objets : Égalité & Équivalence

- Comparable (et Comparator) permettent de représenter un ordre ( $\neq 0$ ), mais aussi une relation d'équivalence ( $= 0$ ).
  - Les cartes de même valeur mais de couleur différentes sont considérée comme équivalente dans Schotten Totten
- Implémenter equals permet de définir une relation d'égalité.
  - qui est symétrique, réflexive et transitive elle aussi !
- L'égalité est une relation d'équivalence parmi d'autres
  - C'est la seule dont le quotient (l'ensemble des classes d'équivalence) contient uniquement des singletons.
  - Autrement dit, pour tout élément e, le seul élément équivalent à e par la relation d'égalité est e lui-même.

## Documentation officielle de Java

The screenshot shows the Java Object.equals() method documentation. The title is "equals". The text describes the method as implementing an equivalence relation on non-null object references. It lists several bullet points about the behavior of equals:

- It is reflexive: for any non-null reference value x, x.equals(x) should return true.
- It is symmetric: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is transitive: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
- It is consistent: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- For any non-null reference value x, x.equals(null) should return false.

Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

**Parameters:**  
obj - the reference object with which to compare.

**Returns:**  
true if this object is the same as the obj argument; false otherwise.

**See Also:**  
hashCode(), HashMap  
"ah, mais fallait vraiment écouter le cours de Maths pour l'Info ?"

**C'est pas "juste" un truc de prof qui sert à rien.**

## Implémentation de l'égalité

```
public class Card implements Comparable<Card> {  
  
    @Override  
    public int compareTo(Card that) {  
        return this.value.compareTo(that.value);  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof Card)) return false;  
  
        Card card = (Card) o;  
  
        if (color != card.color) return false;  
        return value == card.value;  
    }  
}
```

Attention à la signature

Référence Identique

Null, Classes ≠

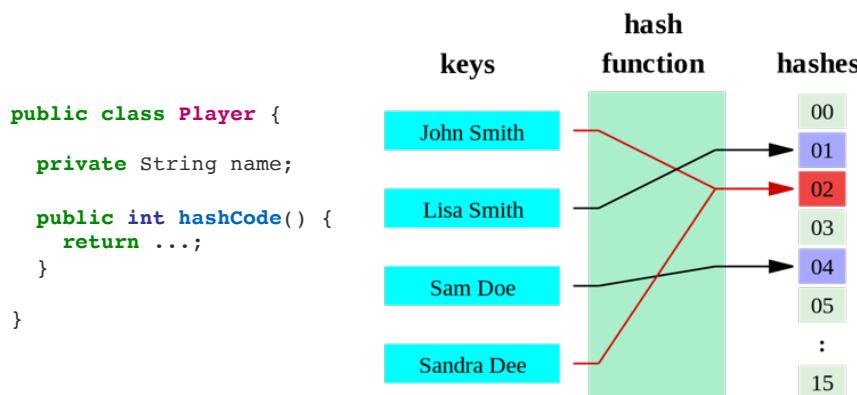
Égalité Structurelle

On peut utiliser `Objects.equals(o,o')` dans les cas courants

## Raccourci sur l'égalité : Fonction de hachage

- Le "hashCode" d'un objet correspond à une "empreinte"
- Deux objets égaux doivent avoir le même hashCode
- Deux objets ≠ peuvent avoir le même hashCode
  - C'est une **collision**, il y a un impact de performances.
- Les structures de données à base de hachage reposent dessus
  - P.-ex. HashMap, HashSet, ...
- En java, si on redéfinit `equals()`, alors on redéfinit `hashCode()`.
  - Sinon les Collections risque de faire n'importe quoi.

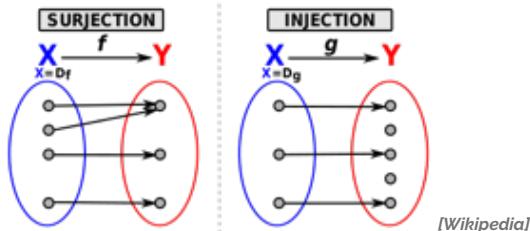
## Principe du Hachage d'un objet



Pour chercher dans une HashMap, on regarde le hash, et en cas de collision on vérifie l'égalité un par un.

[Wikipedia]

## Injectivité & Surjectivité du Hachage



- Un **hachage surjectif** produit des **collisions**
  - Il faut vérifier avec `equals` si les objets sont les mêmes
  - $\mathcal{O}(|X|)$  dans le pire des cas (`int hashCode() { return 42; }`)
- Un **hachage injectif** est dit "parfait"
  - C'est très rare en réalité, p.-ex.  $|CardPile| \ggg |Int|$

## On “génère” souvent ces méthodes

```
public class Card {  
    @Override  
    public int hashCode() {  
        int result = color != null ? color.hashCode() : 0;  
        result = 31 * result + (value != null ? value.hashCode() : 0);  
        return result;  
    }  
  
    @EqualsAndHashCode  
    public class Card {  
  
        private Color color;  
        private Value value;  
  
        @EqualsAndHashCode.Exclude  
        private String uselessField  
    }  
}
```

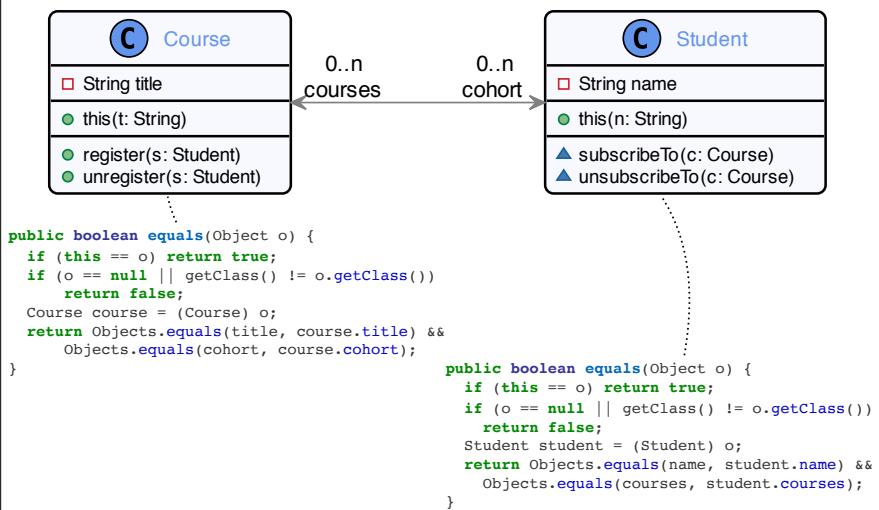
*Template IntelliJ*

*Approche par Annotation et métaprogrammation (ici avec Lombok)*

*Il faut comprendre le principe pour utiliser correctement l'outil*

Il faut comprendre le principe pour utiliser correctement l'outil

## L'exemple qui tue : la dépendance cyclique



## On va même produire des tests !

```
@Test  
public void testOnTitle() {  
    Course inf5151 = new Course("INF-5151");  
    assertEquals(inf5151, new Course("INF-5151"));  
  
    Course inf5153 = new Course("INF-5153");  
    assertNotEquals(inf5153, inf5151);  
}  
  
@Test  
public void testDifferentCohorts() {  
    Course inf5153 = new Course("INF-5153");  
    Student jane = new Student("Jane Doe");  
    inf5153.register(jane);  
    assertNotEquals(inf5153, new Course("Inf-5153"));  
}  
  
@Test  
public void testWithCopy() {  
    Student jane = new Student("Jane Doe");  
  
    Course inf5153 = new Course("INF-5153");  
    Course copy = new Course("INF-5153");  
  
    inf5153.register(jane);  
    copy.register(jane);  
  
    assertEquals(inf5153, copy);  
}
```



StackOverflowError

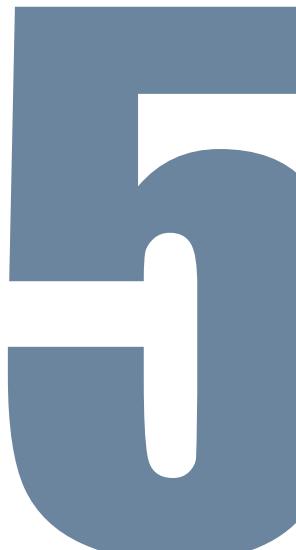


## Fonction equals mutuellement récursive !

```
public boolean equals(Object o) {  
    if (this == o) return true;  
    if (o == null || getClass() != o.getClass())  
        return false;  
    Course course = (Course) o;  
    return Objects.equals(title, course.title) &&  
        Objects.equals(cohort, course.cohort);  
}  
  
StackOverflowError  
  
course.equals(student.courses)  
  
cohort.equals(course.cohort)  
  
public boolean equals(Object o) {  
    if (this == o) return true;  
    if (o == null || getClass() != o.getClass())  
        return false;  
    Student student = (Student) o;  
    return Objects.equals(name, student.name) &&  
        Objects.equals(courses, student.courses);  
}
```

Propagation des appels à `T::equals` dans les `Collections<T>`

## Choix de conception (avec Shotten Totten)



## Comment “justifier” une conception ?

### • Étape 1 : Illustrer le choix de conception

- Avec un **modèle adapté**, par exemple un diagramme ...
  - ...de classe, de séquence, d'état, de cas d'utilisations, ...
- Certains choix peuvent nécessiter **plusieurs visions**
  - p.-ex. une vision **statique** (classes) et **dynamique** (séquence)

### • Étape 2 : Analyser le choix fait (`README.md`, *doc de conception*)

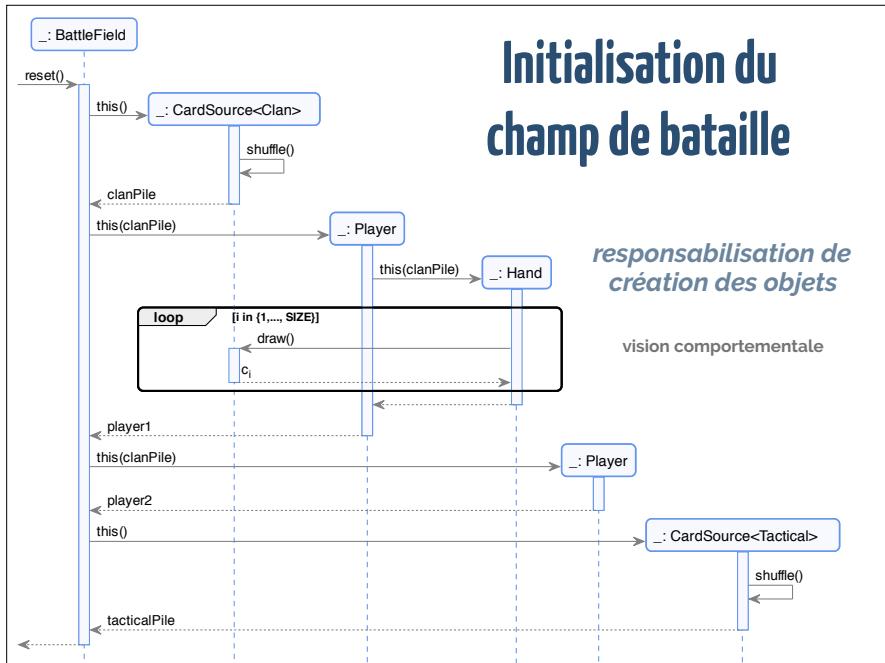
- Décrire brièvement les **alternatives** envisagées
- Caractériser les **forces** (qu'est-ce qu'on y gagne ?)
- Caractériser les **faiblesses** (où sont les limites ?)

## Lisibilité des modèles de conception

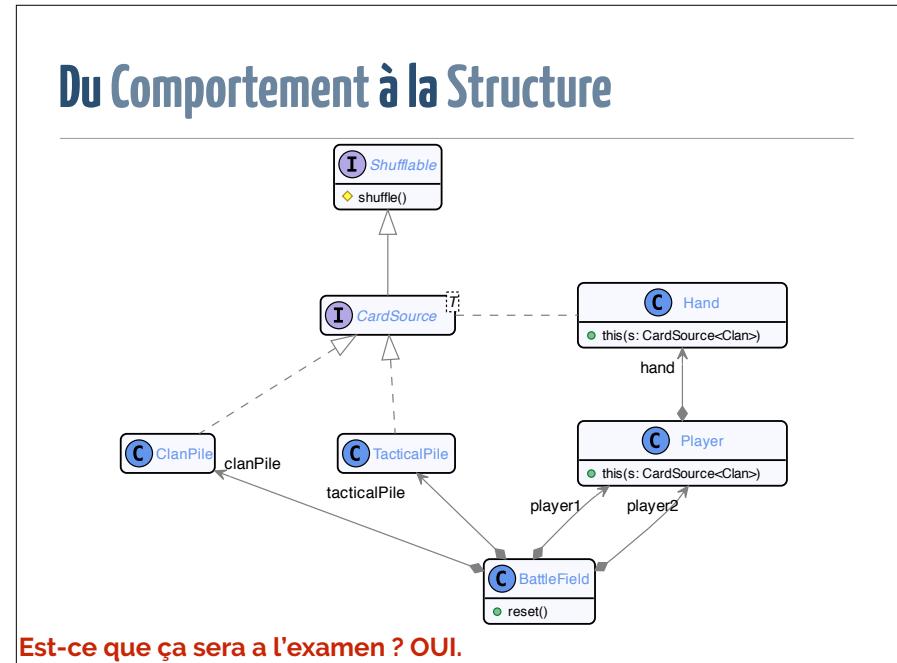
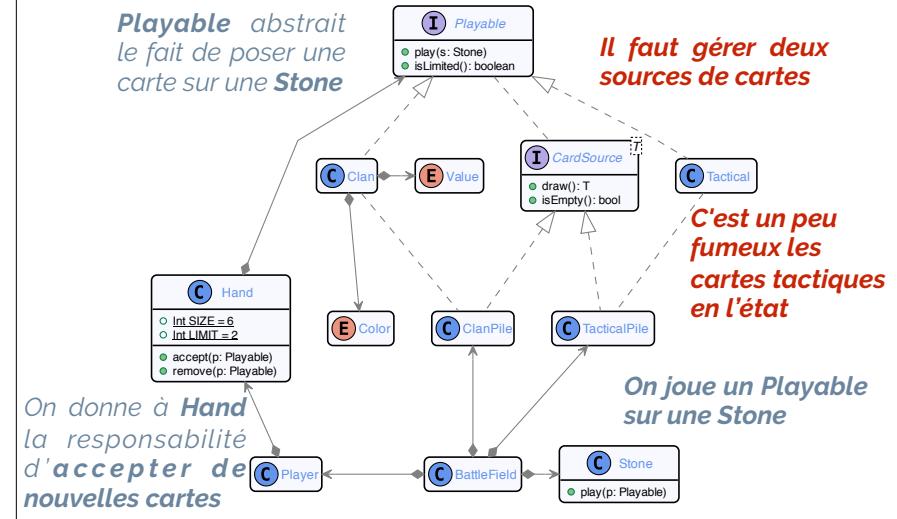
- *Un diagramme de classe avec 357 classes ayant chacune 29 attributs et 31 méthodes en moyenne, et avec 117 relations de généralisation ou de réalisation n'apporte aucune valeur.*
- Les modèles de conception doivent **aider les développeurs**
- Pour illustrer un point de détail :
  - On essaye de rester dans la **“douzaine d'éléments”**
  - Quitte à faire **plusieurs diagrammes** pour montrer les choses
- Pour illustrer la vision globale du projet :
  - On va **limiter au maximum les détails** des entités
  - Plutôt **montrer les relations** que viser **l'exhaustivité**

# Cartes Clan versus Cartes Tactiques

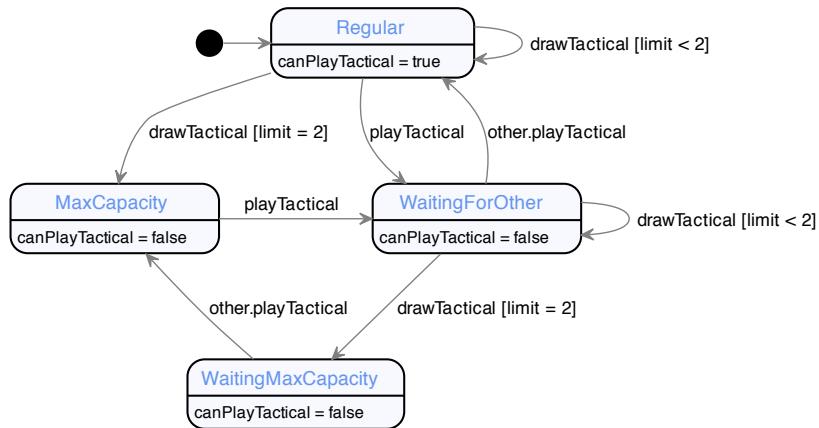
- Les **cartes clan** sont les cartes normales
  - Les **cartes tactiques** sont :
    - Les **jokers** dont on peut choisir la valeur
    - Des cartes qui **modifient la sémantique** d'une attaque
  - Ce sont **deux piles** de cartes **séparées**
    - Quand un joueur a joué une carte tactique, il ne peut plus en jouer une autre tant que l'autre joueur n'en a pas joué une;
    - Un joueur ne peut pas avoir plus de deux (2) cartes tactique dans sa main, qui est limitée à six (6) cartes au maximum.



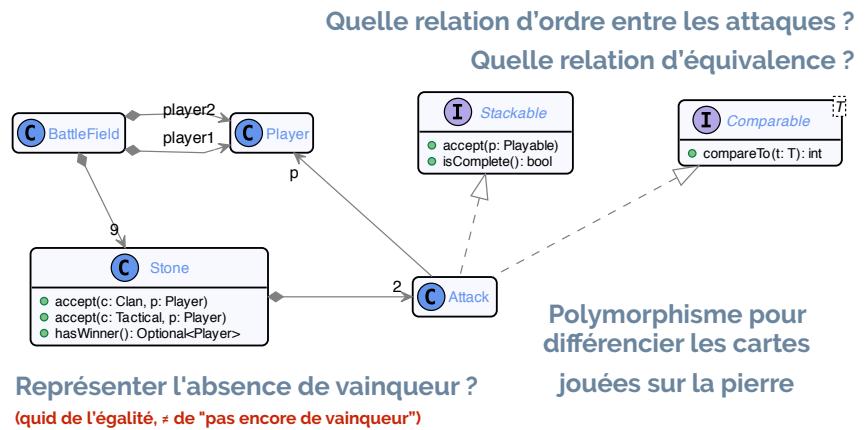
## Vision Structurelle



## Cartes Jouables par un Joueur



## Pose d'une Carte Clan sur une Pierre



## Exercice d'entraînement

- On peut réclamer une pierre si on peut prouver qu'avec toutes les cartes en jeu, il n'est pas possible de battre l'attaque d'un joueur.
- Questions :
  - Quel(s) diagrammes sont les plus adaptés pour représenter les choix de conception que l'on doit faire ?
  - Qui porterait cette responsabilité ?
  - Quel impact sur la conception actuelle ?
- Rejoignez le canal #schotten\_totten sur slack et discutez des solutions potentielles. Je vous donnerais du feedback sur vos propositions.

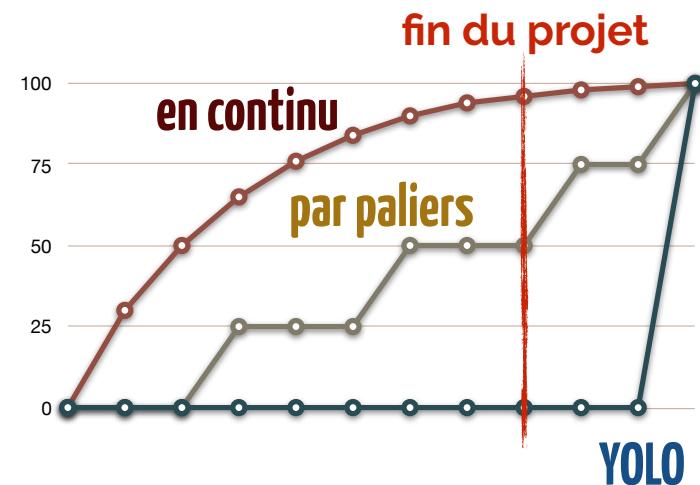
## Projet #1 Questions / Réponses



Qui demande un travail  
**continu**  
**et régulier**  
durant la session.

[Cours 01]

## Valeur cumulée livrée : Git Push !



[Cours 01]

118

Poussez régulièrement votre travail !

The screenshot shows the Travis CI dashboard for the repository 'inf5153-a19 / SKEL-P1-JeuDePoker'. The build status is 'build passing'. The current branch is 'master' with a commit from 'Sébastien Mosser'. The build log shows a successful run for 27 seconds, completed 6 days ago. The Travis CI logo at the bottom left includes the text 'travis-ci.com vous donne de la rétroaction "syntaxique"'.

[travis-ci.com](https://travis-ci.com) vous donne de la rétroaction "syntaxique"

## Petite déception

13 projets sur Git, mais seulement 6 images

(Oui, il y a eu un ex-aequo...)

Si vous êtes arrivé à ce stade du document, c'est que vous l'avez lu en entier, comme demandé dans la consigne au tout début du sujet. Envoyez en message privé au professeur sur le Slack une photo d'un animal mignon (p.-ex. une loutre, un chaton, un chiot, un capybara) pour en témoigner. Les cinq (5) premiers étudiants à remplir cette condition bénéficieront de cinq (5) points bonus sur l'évaluation de ce projet (la note maximale reste de 100).



## Attention au **plagiat** (règlement 18)

- Rendre une **solution de l'année précédente** est du **plagiat**
  - *Le sujet n'est pas même que l'atelier A2 de H19 !*
- Rendre la **même solution qu'un autre étudiant** est un **plagiat**
- Rendre le **code du livre de cours** ou des **diapos** est un **plagiat**
- Attention :
  - *Cela ne doit pas pour autant vous empêcher de **discuter** (Slack)*
  - *Le projet #1 est petit, il est normal qu'il y ait des similitudes*

