Eléments de **Génie Logiciel**

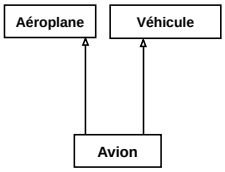
Cours 3

Pr. Jean-Marc Jézéquel
IRISA – Université de Rennes
Campus de Beaulieu
F-35042 Rennes Cedex
e-mail : jezequel@irisa.fr
http://people.irisa.fr/Jean-Marc.Jezequel
X @jmjezequel

8 – Héritage multiple

Héritage entre classes

- Héritage multiple, exemple :
  - Un avion est *à la fois*
    - Un Aéroplane
    - Un Véhicule de transport
  - L'ensemble des avions est donc *l'intersection* entre
    - l'ensemble des aéroplanes et
    - l'ensemble des véhicules de transport

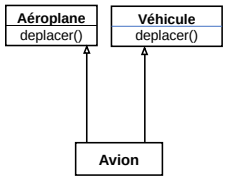


- Réalisation de ce concept très différente selon les langages de programmation
  - Seuls Python, C++ et Eiffel le supportent directement
  - Les autres (dont Java) le supportent indirectement

```
#Python
class
Avion(Aéroplane,Véhicule):
```


Problème avec l'héritage multiple

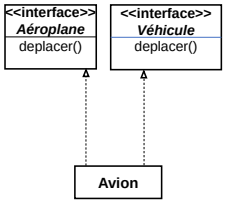
- Avion hérite de quelle version de déplacer()?
  - Aéroplane::déplacer()
  - Véhicule::déplacer()
- Des algorithmes complexes pour fonctionner dans le cas général (Python)
- Des constructions *ad hoc* pour aider le développeur à résoudre ces ambiguïtés (C++, Eiffel)
- Java : seul l'héritage multiple d'interfaces est autorisé



Héritage multiple d'interfaces en Java

```
// Valide en Java?
class Avion implements Aéroplane,
Véhicule {
//…
}
```

```
// Valide en Java?
class Avion extends Aéroplane, Véhicule {
//…
}
```



Avec la notion de Trait (Java 8)

```
public interface Aéroplane {
  public default void deplacer() {
    System.out.println("deplacer un aéroplane");
  } }
```

```
public interface Véhicule {
  public default void deplacer() {
    System.out.println("deplacer un véhicule");
  } }
```

```
public class Avion implements Aéroplane, Véhicule {
  public void deplacer() {
    Véhicule.super.deplacer();
  } }
```

9 – Typage statique

Détecter des erreurs dès la compilation

Typage dans les langages objets

- Type de chaque objet est déterminé par la classe dont il est instance
  - Défini quelles opérations peuvent être appelées sur l'objet, et avec quels paramètres
  - x.f() légal <=> l'objet désigné par x dispose d'une méthode f()
- liaison dynamique :
  - la *bonne* interprétation de f est choisie, selon le type de x

Typage dynamique

- Type de chaque objet est déterminé par la classe dont il est instance
  - Défini quelles opérations peuvent être appelées sur l'objet, et avec quels paramètres
- **Typage dynamique :** les variables peuvent désigner n'importe quel objet
  - c = Cercle() ; c = Compte()
  - Les erreurs de type sont **détectées pendant l'exécution** du programme
    - c.crediter(100) // OK
    - c.deplacer(10,20) // Erreur

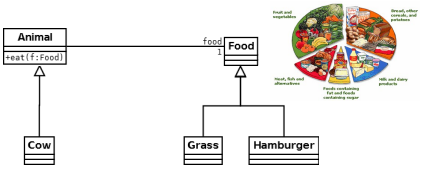
Typage statique

- Type de chaque objet est déterminé par la classe dont il est instance
  - Défini quelles opérations peuvent être appelées sur l'objet, et avec quels paramètres
- **Typage statique :** les variables doivent être déclarées avec un type spécifique
  - Compte c = new Compte();
  - Ce type ne peut généralement pas être modifié par la suite.
  - Les opérations sur les variables sont vérifiées par le compilateur pour s'assurer qu'elles respectent les règles du type déclaré
  - Les erreurs de type sont **détectées avant l'exécution** du programme
    - ce qui permet d'identifier et de corriger les problèmes liés aux types dès le stade de développement

Héritage et typage

- Typage statique :
  - x.f() légal <=> vérification avant l'exécution que tout objet potentiellement désigné par x dispose d'une méthode f()
- liaison dynamique :
  - la bonne interprétation de f est choisie
- Combinaison de ces notions
  - Python, JavaScript : typage dynamique, liaison dynamique
  - Java, C#, Swift, Dart, Kotlin : typage statique, liaison dynamique
  - C++ : typage statique, liaison dynamique
    - pour les fonctions "virtuelles"
  - UML : au choix (!)

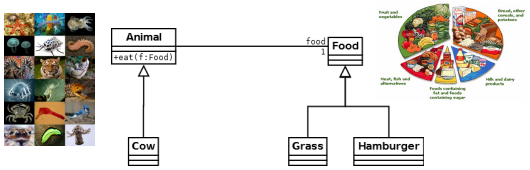
Co-variance et contra-variance

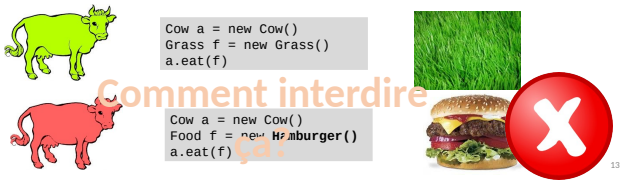


```
Animal a = new Animal()
Food f = new Food()
a.eat(f)
```

```
Animal a = new Cow()
Food f = new Food()
a.eat(f)
```

Co-variance et contra-variance



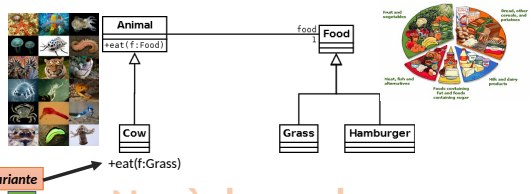


```
Cow a = new Cow()
Grass f = new Grass()
a.eat(f)
```

Comment interdire ça ?

```
Cow a = new Cow()
Food f = new Hamburger()
a.eat(f)
```


Co-variance et contra-variance



Redéfinition *covariante*

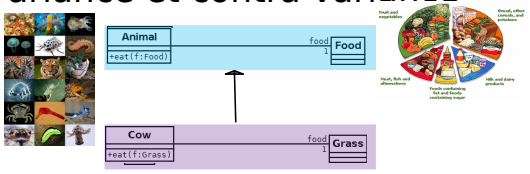
Légal en Java ?

Illégal?

Ne règle pas le problème !

```
Animal a = new Cow()
Food f = new Hamburger()
a.eat(f)
```


Co-variance et contra-variance



- En Java (& autres langages à typage statique) : généricité paramétrique
  - **Class** Animal<T **extends** Food> {void eat(f:T){}}
  - **Class** Cow **extends** Animal<Grass>{} // *devient implicitement* eat(f:Grass)
- Erreur *cow.eat(new Hamburger())* maintenant détectée **à la compilation**

Co-variance et contra-variance

- **Covariance**
  - **Définition :** Autorise la substitution d'un type dérivé (sous-type) à la place d'un type de base (super-type)
- **Contravariance**
  - Autorise la substitution d'un type de base à la place d'un type dérivé
- Règle en Java lors de la redéfinition de méthodes
  - Covariance pour les types de retours
  - Contravariance pour les paramètres
    - Exemples :

Légal

Illégal

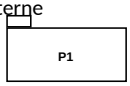
10 – Notion de package

Modularisation à l'échelle supérieure

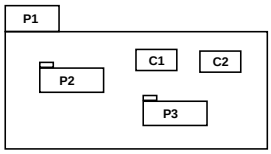
Notion de package

- Élément structurant les classes
  - Modularisation à l'échelle supérieure
  - Un package partitionne l'application :
    - Il référence ou se compose des classes de l'application
    - Il référence ou se compose d'autres packages
  - Un package réglemente la visibilité des classes et des packages qu'il référence ou le compose
  - Les packages sont liés entre eux par des liens d'utilisation, de composition et de généralisation
  - Un package est la représentation informatique du contexte de définition d'une classe

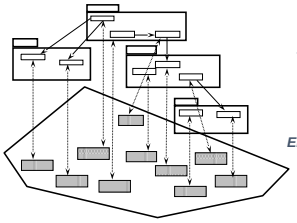
- Vue graphique externe



- Vue graphique externe et interne



- Définition de vues partielles d'une application



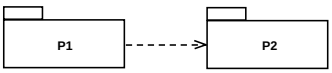
N.B.: ***une classe appartient à un et un seul package***

Visibilité dans un package

- Réglementation de la visibilité des classes
  - **Classes de visibilité publique** :
    - classes utilisables par des classes d'autres packages
  - **Classes de visibilité privée** :
    - classes utilisables seulement au sein d'un package
- Représentation graphique

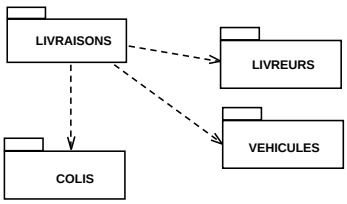
Utilisation entre packages

- Définition
  - Il y a utilisation entre packages si des classes du package utilisateur accèdent à des classes du package utilisé
  - Pour qu'une classe d'un package p1 puisse utiliser une classe d'un package p2, il doit y avoir au préalable une déclaration **explicite** de l'utilisation du package p2 par le package p1
- Représentation graphique



Vue externe du package P1

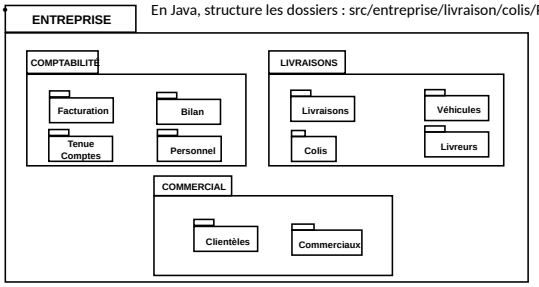
- Exemple (vue externe du package livraisons)



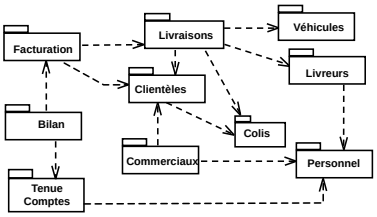
Architecture logicielle : vue hiérarchique

- Composition des packages en sous-packages

En Java, structure les dossiers : src/entreprise/livraison/colis/Produit.java



Architecture logicielle : dépendances

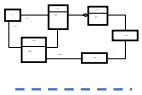


Utilité des packages

- Réponses au besoin
  - Contexte de définition d'une classe
  - Unité de structuration
  - Unité d'encapsulation
  - Unité d'intégration
  - Unité de réutilisation
  - Unité de configuration
  - Unité de production

Synthèse diagrammes de classes et d'objets

- Un diagramme de classes
  - Défini l'ensemble de tous les états possibles
  - les contraintes doivent toujours être vérifiées



- Un diagramme d'objets
  - décrit un état possible à un instant t, un cas particulier
  - doit être conforme au modèle de classes

- Les diagrammes d'objets peuvent être utilisés pour
  - expliquer un diagramme de classe (donner un exemple)
  - valider un diagramme de classe (le "tester")