Java Avancé

Cours 2 : Polymorphisme

Khitem BEN ALI BLIDAOUI
Slides d'après Florian Sikora

Khitem.blidaoui@dauphine.psl.eu

Cours 2 : Polymorphisme et compagnie

Cours 2 : Polymorphisme et compagnie

Types

- Typage à la compilation et/ou l'exécution.
- Seulement à la compilation :
  - C++, OCaml...
- Seulement exécution :
  - PHP, Javascript, Python, Ruby...
- Les deux :
  - Java, C#...

Types et Objet

```
1 Object o = new Integer(3) ;
```

- Qu'est-ce qui est le type de l'objet, qu'est-ce qui est la classe de l'objet ?

Types et Objet

```
1 Object o = new Integer(3) ;
```

- Qu'est-ce qui est le type de l'objet, qu'est-ce qui est la classe de l'objet ?
- **Type** de o : son interface : ensemble des méthodes pouvant être appelées (ici `Object`).
  - Connu et vu par le compilateur.
- **Classe** de o : ensemble des propriétés et méthodes utilisées pour créer l'objet (ici `Integer`).
  - Connu par la VM.

Types et Objet

```
1 Object o = new Integer(3) ;
```

- Qu'est-ce qui est le type de l'objet, qu'est-ce qui est la classe de l'objet ?
- **Type** de o : son interface : ensemble des méthodes pouvant être appelées (ici `Object`).
  - Connu et vu par le compilateur.
- **Classe** de o : ensemble des propriétés et méthodes utilisées pour créer l'objet (ici `Integer`).
  - Connu par la VM.
  - Opérations dynamiques en ont besoin.
    - Création, `getClass()`, `instanceof`...

Object

- Toutes les classes héritent directement (ajouté par le compilateur quand on définit une classe) ou indirectement (héritage d'une classe) de `java.lang.Object`.
- 3 méthodes universelles :

Object

- Toutes les classes héritent directement (ajouté par le compilateur quand on définit une classe) ou indirectement (héritage d'une classe) de `java.lang.Object`.
- 3 méthodes universelles :
  - `toString()` : affichage debug de l'objet.
  - `equals()` : renvoie vrai si deux objets sont égaux.
  - `hashCode()` : "résumé" de l'objet sous forme d'entier.

Object

- Une implémentation par défaut des 3 méthodes.

```
1  public class Miage {
2    private final int year ;
3    public Miage(int year) {
4      this.year = year ;
5    }
6    public static void main(String[] args) {
7      Miage m = new Miage(2014) ;
8      System.out.println(m.toString()) ; //Miage@264532ba : class+@+hashCode()
9      System.out.println(m.equals(new Miage(2014))) ; // false car ==
10     System.out.println(m.equals(m)) ; //true car ==
11     System.out.println(m.hashCode()) ; //random sur 24bits
12   }
13 }
```

Object

```
1 Miage m = new Miage(2014) ;
2 System.out.println(m) ; //Miage@40f92a41
```

- Comment peut-on appeler une méthode avec un objet de type inconnu lors de son écriture ?
- Signature de println : `public void println(Object x) ;`

Object

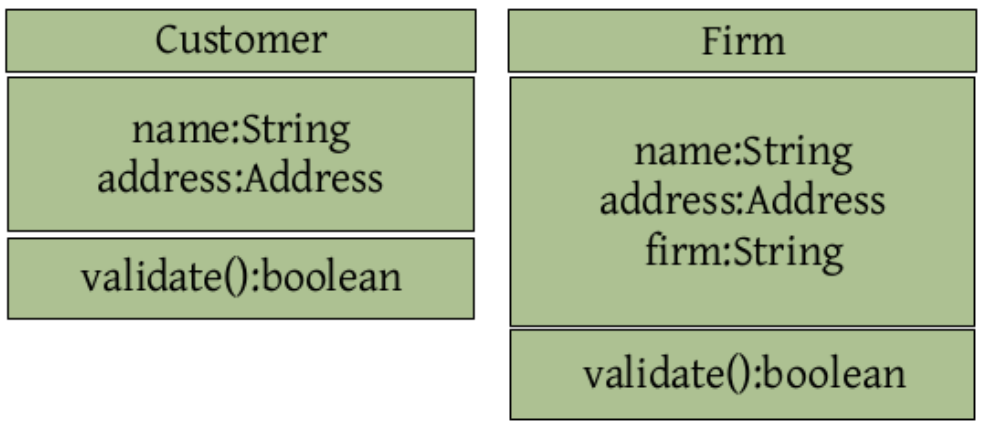
```
1 Miage m = new Miage(2014) ;
2 System.out.println(m) ; //Miage@40f92a41
```

- Comment peut-on appeler une méthode avec un objet de type inconnu lors de son écriture ?
- Signature de println : `public void println(Object x) ;`
- Miage **hérite** d'Object.
- Objets de la classe Miage peuvent être manipulés par des références sur Object : `Object o = new Miage(2014)`.
- Miage est un sous-type d'Object, Object est un super-type de Miage.

Cours 2 : Polymorphisme et compagnie

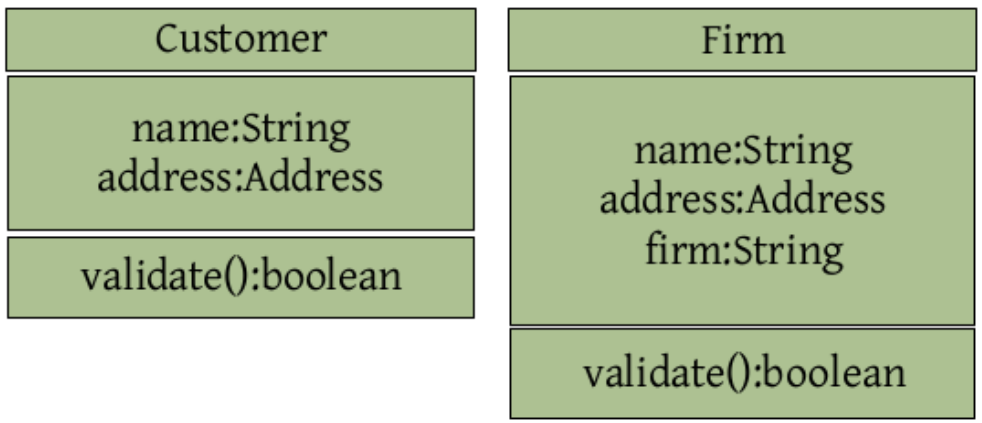
Héritage - Exemple

- Application de commerce en ligne.
- Deux types de clients :
  1. Des particuliers.
  2. Des entreprises.
- Design possible :



Héritage - Exemple

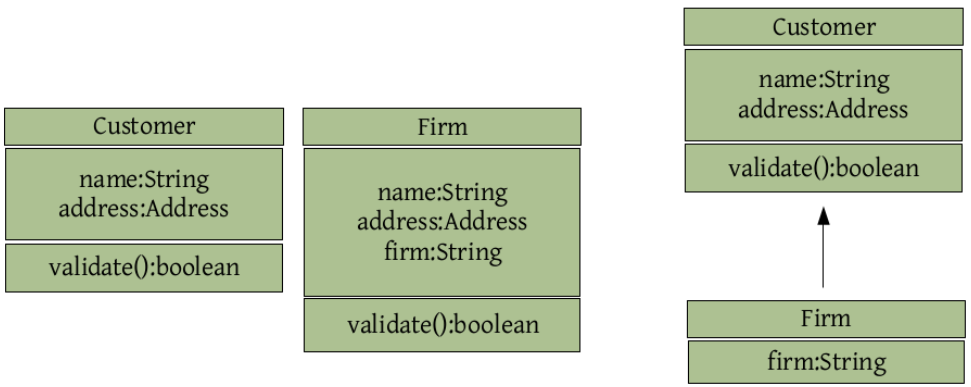
- Application de commerce en ligne.
- Deux types de clients :
  1. Des particuliers.
  2. Des entreprises.
- Design possible :



- Similarités ?

Héritage - Exemple

- Firm est comme un Customer, mais avec un champ en plus.



- Customer super-classe.
- Firm sous-classe.

Héritage - code

- Sans héritage.

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9  }
```

```
1  public class Firm {
2    private final String name ;
3    private final Address address ;
4    private final String firm ;
5
6    public Firm(String name, Address
           address, String firm) {
7      this.name = name ;
8      this.address = address ;
9      this.firm = firm ;
10   }
11 }
```

Héritage - code

- Avec héritage.

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9  }
```

```
1  //mot clef extends
2  public class Firm extends Customer {
3    private final String firm ;
4    //autres champs hérités
5
6    public Firm(String name, Address
           address, String firm) {
7      this.name = name ;
8      this.address = address ;
9      //compile pas
10     this.firm = firm ;
11   }
12 }
```

Héritage - code

- Avec héritage.

```
1 public class Customer {
2   private final String name ;
3   private final Address address ;
4
5   public Customer(String name,
          Address address) {
6     this.name = name ;
7     this.address = address ;
8   }
9 }
```

```
1 public class Firm extends Customer {
2   private final String firm ;
3
4   public Firm(String name, Address
          address, String firm) {
5     //appel au constructeur de la
            super-classe
6     super(name, address) ;
7     this.firm = firm ;
8   }
9 }
```

Héritage - code

- Champs et méthodes hérités !

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
           name.isEmpty() ;
12   }
13 }
```

```
1 public class Firm extends Customer {
2   private final String firm ;
3
4   public Firm(String name, Address
        address, String firm) {
5     super(name, address) ;
6     this.firm = firm ;
7   }
8   //bug potentiel !
9 }
```

- Pourquoi ?

Héritage - code

- Champs et méthodes hérités !

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
           name.isEmpty() ;
12   }
13 }
```

```
1 public class Firm extends Customer {
2   private final String firm ;
3
4   public Firm(String name, Address
        address, String firm) {
5     super(name, address) ;
6     this.firm = firm ;
7   }
8   //bug potentiel !
9 }
```

- Pourquoi ?
- On hérite de validate()...qui ne vérifie pas le champ firm !

```
1 Firm f = new Firm(..) ;
2 if(firm.validate()){...
```

Héritage - code

- Bug ?

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
             name.isEmpty() ;
12   }
13 }
```

```
1  public class Firm extends Customer {
2    private final String firm ;
3
4    public Firm(String name, Address
           address, String firm) {
5      super(name, address) ;
6      this.firm = firm ;
7    }
8    @Override
9    public boolean validate() {
10     return DB.exists(address) &&  !
             name.isEmpty() &&  !firm.
             isEmpty() ;
11   }
12 }
```

Héritage - code

- Bug ?

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
             name.isEmpty() ;
12   }
13 }
```

```
1  public class Firm extends Customer {
2    private final String firm ;
3
4    public Firm(String name, Address
           address, String firm) {
5      super(name, address) ;
6      this.firm = firm ;
7    }
8    @Override
9    public boolean validate() {
10     return DB.exists(address) &&  !
             name.isEmpty() &&  !firm.
             isEmpty() ;
11   }
12 }
```

- Compile pas !

Héritage - code

- Bug ?

```
1  public class Customer {
2    private final String name ;
3    private final Address address ;
4
5    public Customer(String name,
           Address address) {
6      this.name = name ;
7      this.address = address ;
8    }
9
10   public boolean validate() {
11     return DB.exists(address) &&  !
           name.isEmpty() ;
12   }
13 }
```

```
1  public class Firm extends Customer {
2    private final String firm ;
3
4    public Firm(String name, Address
         address, String firm) {
5      super(name, address) ;
6      this.firm = firm ;
7    }
8    @Override
9    public boolean validate() {
10     return super.validate() &&  !firm.
         isEmpty() ;
11   }
12 }
```

- On délègue à la super-classe la validation des champs de Customer.

Héritage

- Bien en 1960, discutable de nos jours.
  - Ré-utiliser un algo : utiliser une méthode `static` dans une classe public...
  - Hériter implique aussi :
    - Le sous-typage.
    - L'héritage de **toutes** les méthodes...
    - Ne peut pas changer l'implémentation de la sous-classe si on ne peut pas toucher celle de la super-classe.

Héritage

- Doit redéfinir **toutes** les méthodes...

```
1  public class EmployeesList extends ArrayList<Employee> {
2    @Override
3    public boolean add(Employee e) {
4      Objects.requireNonNull(e) ; //pas de null dans ma liste
5      return super.add(e) ;
6    }
7    public static void main(String[] args) {
8      List<Employee> l = new ArrayList<Employee>() ;
9      l.add(null) ;
10     EmployeesList el = new EmployeesList() ;
11     el.addAll(l) ;
12     System.out.println(el) ; //[null]
13   }
14 }
```

Héritage

- Solution : déléguer.
- Pas le lien fort entre sous-classe et super-classe.

```
1 public class EmployeesList {
2   private final ArrayList<Employee> al = new ArrayList<Employee>() ;
3   private void add(Employee e) {
4     Objects.requireNonNull(e) ;
5     al.add(e) ; //delegation
6   }
7 }
```

Héritage et constructeurs

- Constructeurs non hérités ($\neq$ méthodes).
- Un constructeur doit faire appel au constructeur de sa super classe en **première** instruction.
- Fait implicitement si constructeur par défaut (sans argument).

```
1 public class Fruit {
2 }
```

```
1 public class Orange extends Fruit {
2   private final String s ;
3   public Orange(int a, String s) {
4     //ok
5     this.s=s ;
6   }
7 }
```

Héritage et constructeurs

- Constructeurs non hérités ($\neq$ méthodes).
- Un constructeur doit faire appel au constructeur de sa super classe en **première** instruction.
- Fait implicitement si constructeur par défaut (sans argument).
  - Attention, rappel, si définition d'un constructeur avec argument, plus de constructeur par défaut !

```
1 public class Fruit {
2   public Fruit(int a) {
3   }
4 }
```

```
1 public class Orange extends Fruit {
2   public Orange() {
3     //Compile pas
4     //Implicit super constructor Fruit() is
          undefined.
5     //Must explicitly invoke another
          constructor
6   }
7 }
```

Héritage et constructeurs

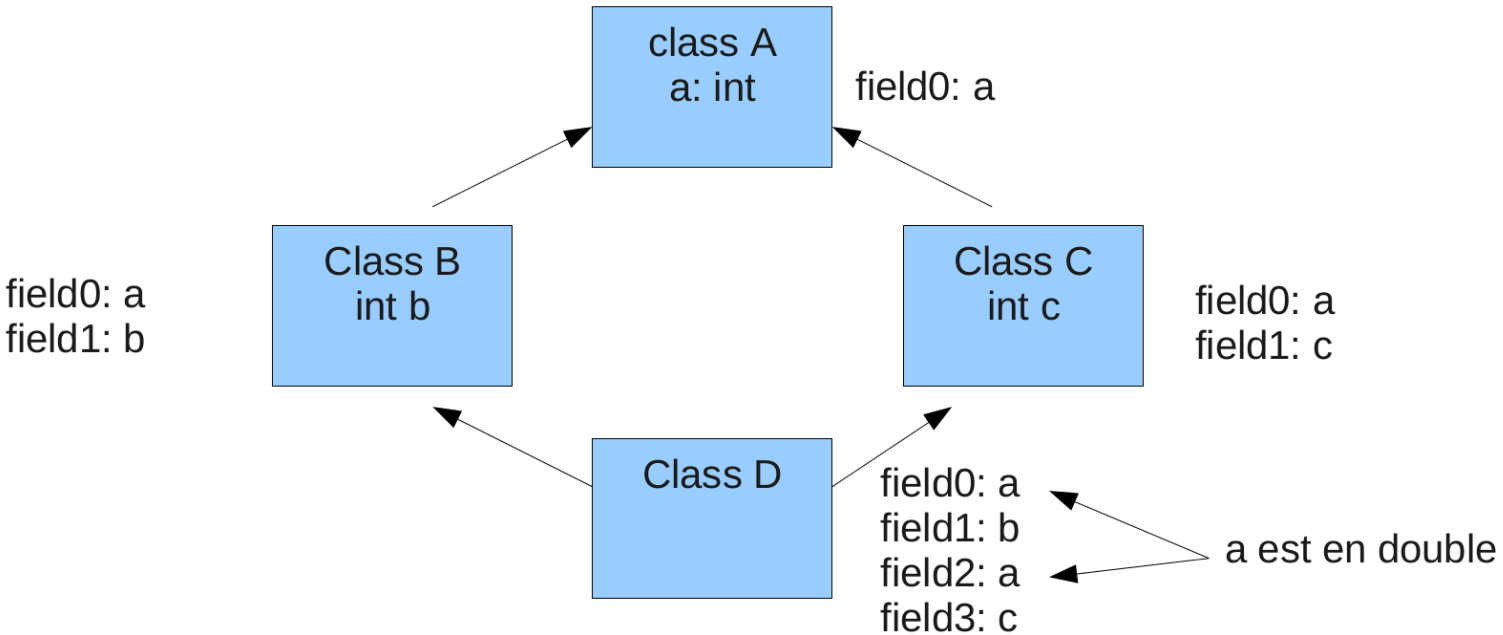
- Constructeurs non hérités ($\neq$ méthodes).
- Un constructeur doit faire appel au constructeur de sa super classe en **première** instruction.
- Fait implicitement si constructeur par défaut (sans argument).
  - Attention, rappel, si définition d'un constructeur avec argument, plus de constructeur par défaut !

```
1 public class Fruit {
2   public Fruit(int a) {
3   }
4 }
```

```
1 public class Orange extends Fruit {
2   private final String s ;
3   public Orange(int a, String s) {
4     super(a) ; //en premier !
5     this.s = s ;
6   }
7 }
```

Héritage multiple

- En Java, pas d'héritage multiple...
- Problème du diamant.



Interface

- Parfois besoin de voir un objet comme étant deux "trucs"...
- Utiliser des interfaces pour du sous-typage multiple (mot clef `interface`) :
  - Définition d'un type sans son implémentation (méthodes sans code).
  - Interdit d'instancier une interface (car pas de code !).
- Une classe peut fournir l'implémentation de plusieurs interfaces.

  - Pas de problème car pas de code hérité !

Interface

- Parfois besoin de voir un objet comme étant deux "trucs"...
- Utiliser des interfaces pour du sous-typage multiple (mot clef `interface`) :
  - Définition d'un type sans son implémentation (méthodes sans code).
  - Interdit d'instancier une interface (car pas de code !).
- Une classe peut fournir l'implémentation de plusieurs interfaces.
  - Pas de problème car pas de code hérité !
- Amélioration avec Java 8 et les traits (interface avec méthodes mais sans champs), choix à la main en cas de conflit ou méthode par défaut.

Interface

- Pourquoi les interfaces ?

Interface

- Pourquoi les interfaces ?
- Définir "l'interface" entre deux modules du projet.
- Définir une fonctionnalité transversale (ex : `Comparable`, `Cloneable`...).
- Définir un ensemble de fonctionnalités avec plusieurs implémentation possibles (ex : `ArrayList`, `LinkedList`...).

Interface

- Le compilateur vérifie que toutes les méthodes de l'interface sont implémentées par la classe.
- On peut décider de lever une exception si on ne veut pas implémenter cette opération.

```
1  public interface List {
2    public Object get(int index) ;
3    /** optional operation */
4    public void set(int index,Object o) ;
5  }
6
7  public class NullList implements List {
8    public void set(int index,Object o) {
9      throw new UnsupportedOperationException() ;
10   }
11   public Object get(int index) {
12     return null ;
13   }
14 }
```

Interface

- Rien interdit d'ajouter des méthodes dans l'implémentation d'une interface.
  - La variable devra être typée du nom de l'implémentation et non de l'interface pour les utiliser.

```
1  public interface ReadOnlyHolder<T> {
2    public T get() ;
3  }
4
5  public class Holder<T> implements ReadOnlyHolder<T> {
6    private T t ;
7    public T get() {
8      return t ;
9    }
10   public void set(T t) {
11     this.t=t ; //readOnly  ?   :(
12   }
13 }
```

Interface

- **Méthodes** d'une interface, obligatoirement et implicitement `abstract` et `public`.
- **Champs** d'une interface, obligatoirement et implicitement `final`, `public`, `static`.
- Impossible d'un définir une méthode statique.

Héritage

- En résumé, hériter c'est :
  1. Récupérer les membres visibles (champs, méthodes etc).
  2. Possibilité de redéfinir les méthodes.
  3. Sous-typage (Orange est un Fruit).

Cours 2 : Polymorphisme et compagnie

Sous-typage

- Substituer un type par un autre.
- A quoi ça sert ?

Sous-typage

- Substituer un type par un autre.
- A quoi ça sert ?
- Utiliser un algorithme écrit pour un certain type et l'utiliser avec un autre.

Sous-typage

- Substituer un type par un autre.
- A quoi ça sert ?
- Utiliser un algorithme écrit pour un certain type et l'utiliser avec un autre.
- Si `Orange` hérite de `Fruit`, alors `Orange` est un sous-type de `Fruit` :
  - Partout où un objet `Fruit` est attendu, on peut lui donner une `Orange`.
  - L'inverse est faux.

Sous-typage

- Existe pour (slides suivantes) :
  - Héritage de classe ou d'interface.
  - Implémentation d'interface.
  - Tableaux d'objets.
  - Types paramétrés.

Sous-typage et héritage

```
1 class A {
2 }
3 class B extends A {
4 }
5 public static void main(String[] args) {
6   A a = new B() ;
7 }
```

- B hérite de A.
- B est un sous-type de A.
- B récupère tous les membres de A.

Sous-typage et interface

```
1  interface I {
2    void m() ;
3  }
4  class A implements I {
5    public void m() {
6    }
7  }
8  public static void mani(String[] args) {
9    I i = new A() ;
10 }
```

- A est un sous-type de I.
- A possède un code pour toutes les méthodes de I.

Sous-typage et tableaux

- Tableaux java sont des sous-types d'Object (et Serializable et Cloneable).
- U[] est un sous type de T[] si U est un sous-type de T, et si U et T ne sont pas primitifs.

```
1 double[] t = new int[2] ;  //ko
2 Number[] tab = new Integer[2] ; //ok
```

Sous-typage et tableaux

- Problème à l'exécution...

```
1 Number[] tab = new Integer[2] ; //ok
2
3 tab[0] = 2 ;
4 tab[1] = 2.2 ;  //java.lang.ArrayStoreException
```

Sous-typage et types paramétrés

- Pas de check à l'exécution pour les types paramétrés.
- `List<String>` n'est pas un sous-type d'une `List<Object>`.
- Voir cours sur les wildcards pour gérer ce problème.

```
1  ArrayList<String> al1 = new ArrayList<>() ;
2  ArrayList<Object> al2 = al1 ; //compile pas
3
4  //si compilait :
5  al2.add(new Integer(3)) ; //compile  ! !  :(
6  String s = al1.get(0) ; //Aie, ClassCastException...
```

Cours 2 : Polymorphisme et compagnie

Polymorphisme

- Idée du **polymorphisme** : considérer les fonctionnalités suivant le type réel d'un objet et non le type de la variable où il est référencé.
- **Sous-typage** : stocker un objet comme une variable d'un super-type.

Polymorphisme

- Intérêt ?

Polymorphisme

- Intérêt ?
- Avoir certaines parties d'un algorithme spécialisé en fonction du type réel de l'objet.
- Pas besoin de dispatcheur dans l'algorithme
- PAS DE INSTANCEOF !!

Polymorphisme

```java
public class Fruit {
  public void name() {
    System.out.println("Je suis un
        fruit") ;
  }
}
```

```java
public class Orange extends Fruit {
  @Override
  public void name() {
    System.out.println("Je suis une
        orange") ;
  }
}
```

```java
public class Pamplemousse extends
    Fruit {
  @Override
  public void name() {
    System.out.println("Je suis un
        pamplemousse") ;
  }
}
```

```java
public class Polym {
  public static void main(String[]
      args) {
    Fruit[] fruits = new Fruit[]{new
        Orange(), new Pamplemousse()
        } ;

    for (Fruit f  : fruits) {
      f.name() ;
    }
  }
}
```

- Affiche ?

Polymorphisme

```java
public class Fruit {
  public void name() {
    System.out.println("Je suis un
        fruit") ;
  }
}
```

```java
public class Orange extends Fruit {
  @Override
  public void name() {
    System.out.println("Je suis une
        orange") ;
  }
}
```

```java
public class Pamplemousse extends
    Fruit {
  @Override
  public void name() {
    System.out.println("Je suis un
        pamplemousse") ;
  }
}
```

```java
public class Polym {
  public static void main(String[]
      args) {
    Fruit[] fruits = new Fruit[]{new
        Orange(), new Pamplemousse()
        } ;

    for (Fruit f  : fruits) {
      f.name() ;
    }
  }
}
```

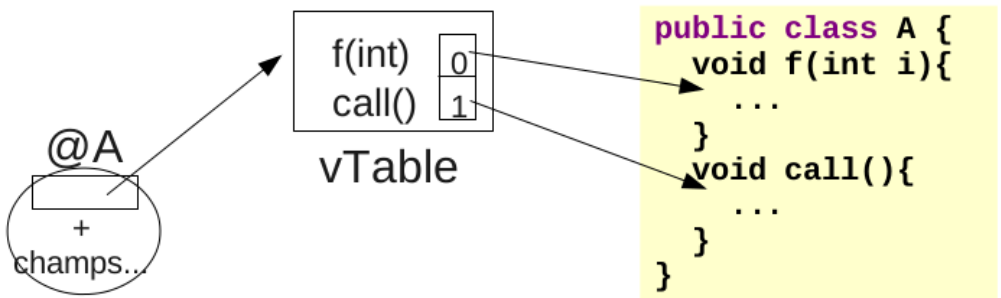
- Affiche ?
  - Je suis une orange
  - Je suis un pamplemousse

Polymorphisme

- Il y a **redéfinition** de méthode si le site d'appel (ici dans la boucle) peut appeler la méthode redéfinie à la place de celle choisie à la compilation.
- Rédéfinition d'une méthode selon :
  - Son nom.
  - Sa visibilité (au moins aussi grande que l'originale).
  - Sa signature (depuis 1.5, type de retour peut être un sous-type).
  - Ses exceptions (peut être un sous-type ou pas d'exception).
  - Son paramétrage (soit toutes les 2, soit aucune).
- Vérifié par l'annotation `@Override` à la compilation.
  - Compile pas si non respecté.
  - Pas indispensable au polymorphisme.

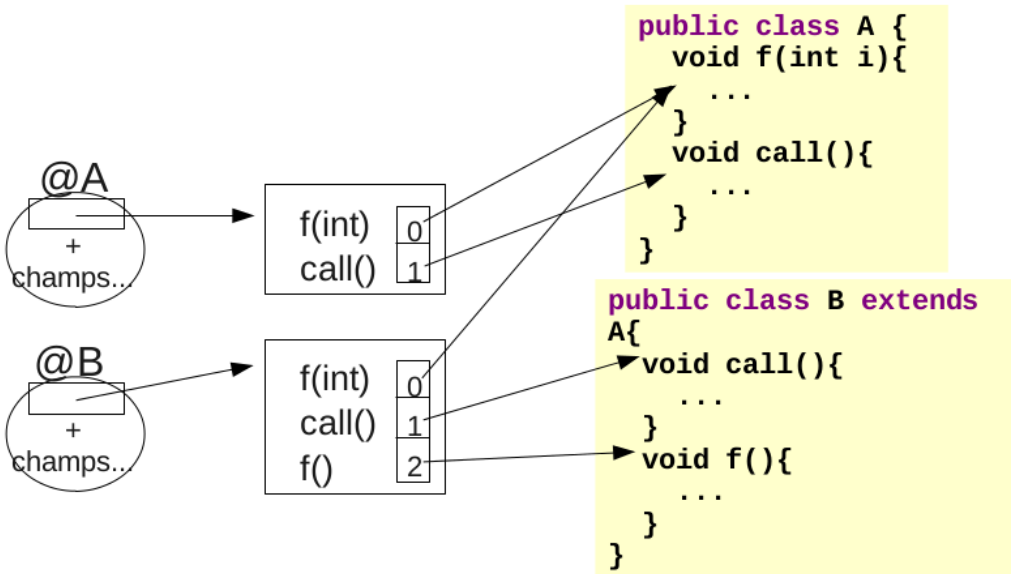
Polymorphisme - Comment ça marche ?

- Chaque objet possède en plus de ses champs un pointeur sur une table de pointeurs de fonctions (vTable) (une par classe).
- Pour chaque méthode : un index dans la table, en commençant par les super-classes.



Polymorphisme - Comment ça marche ?

- Deux méthodes redéfinies ont le **même index**.
- L'appel polymorphe est : `object.vtable[index](args)`.



Rédéfinition vs Surcharge

- Quelle différence ?

Rédéfinition vs Surcharge

- Quelle différence ?
- **Surcharge** : **des** méthodes de **même noms** mais de **profils différents** dans une même classe.
  - Choisi à la **compilation** selon les arguments.
- **Redéfinition** : **deux** méthodes de **même noms** et de **même profil** dans 2 classes dont l'une hérite de l'autre.
  - Choisi à l'exécution selon le type réel.

Rédéfinition vs Surcharge

```
1 public class A {
2   public void m(CharSequence a) {...} // surcharge
3   public void m(List<Character> a) {...} // surcharge
4 }
5 public class B extends A {
6   public void m(Object a) {...} // surcharge
7   public void m(CharSequence a) {...} // redéfinition
8 }
```

Exemple

- On désire dessiner un tableau avec des rectangles et des ellipses sur une surface graphique.

```
1 public class Ellipse {
2   private final int x,y,w,h ;
3 }
```

```
1 public class Rectangle {
2   private final int x,y,w,h ;
3 }
```

```
1 public class DrawingArea {
2   public void drawRect(x,y,w,h){...}
3   public void drawEllipse(x,y,w,h) {...}
4 }
```

Exemple - 1ère tentative

```
1  public static void drawAll(DrawingArea area, Object[] array) {
2    for(Object o : array) {
3      if (o instanceof Rectangle) {
4        Rectangle r = (Rectangle)o ;
5        area.drawRect(r.x, r.y, r.width, r.height) ;
6      } else
7      if (o instanceof Ellipse) {
8        Ellipse e = (Ellipse)o ;
9        area.drawEllipse(e.x, e.y, e.width, e.height) ;
10     } else {
11       throw new AssertionError() ;
12     }
13   }
14 }
```

Exemple - 1ère tentative

```
1  public static void drawAll(DrawingArea area, Object[] array) {
2    for(Object o : array) {
3      if (o instanceof Rectangle) {
4        Rectangle r = (Rectangle)o ;
5        area.drawRect(r.x, r.y, r.width, r.height) ;
6      } else
7      if (o instanceof Ellipse) {
8        Ellipse e = (Ellipse)o ;
9        area.drawEllipse(e.x, e.y, e.width, e.height) ;
10     } else {
11       throw new AssertionError() ;
12     }
13   }
14 }
```

- Si on rajoute une forme, il faut penser à aller dans cette static...
- Ajout de forme ne peut être fait que par le développeur de drawAll()

Exemple - 2ème tentative - héritage

```java
public class Rectangle {
  private final int x,y,w,h ;
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse extends Rectangle{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

```java
public static void drawAll(DrawingArea area, Rectangle[] array) { //astuce sous-
    typage
  for(Rectangle r : array) {
    r.draw(area) ; //astuce polymorphisme
  }
}
```

Exemple - 2ème tentative - héritage

```java
public class Rectangle {
  private final int x,y,w,h ;
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse extends Rectangle{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

```java
public static void drawAll(DrawingArea area, Rectangle[] array) { //astuce sous-
    typage
  for(Rectangle r : array) {
    r.draw(area) ; //astuce polymorphisme
  }
}
```

- Mais pourquoi une ellipse est un rectangle ?

Exemple - 3ème tentative - interface

- Besoin d'un "contrat" que nos forment doivent respecter (draw).
- Interface permet sous-typage et polymorphisme, mais sans l'héritage des champs et méthodes ("héritage simplifié").

Exemple - 3ème tentative - interface

```java
public interface Shape {
  public void draw(DrawingArea area) ;
}
```

```java
public static void drawAll(DrawingArea
      area, Shape[] array) { //sous-
    typage
  for(Shape s : array) {
    s.draw(area) ; //polymorphisme
  }
}
```

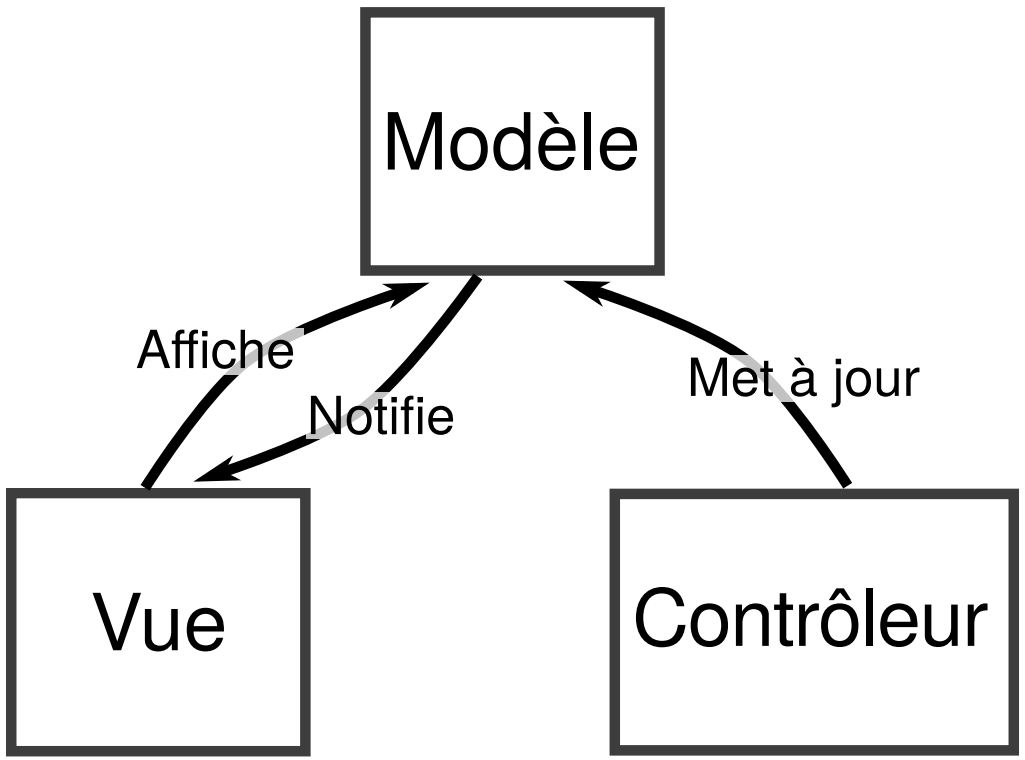
```java
public class Rectangle implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawRect(x,y,w,h) ;
  }
}
```

```java
public class Ellipse implements Shape{
  private final int x,y,w,h ;
  @Override
  public void draw(DrawingArea area){
    area.drawEllipse(x,y,w,h) ;
  }
}
```

Cours 2 : Polymorphisme et compagnie

Modèle de conception MVC

- Modèle → Représentation abstraite du monde
- Vue → Affiche le modèle
- Contrôleur → Met à jour le modèle



Modèle et vue

paquetage Modèle : représentation conceptuelle

- `abstract class Shape`
- `class Rectangle extends Shape`
- `class Ellipse extends Shape`
- `class World`
- ...

paquetage Vue : visualisation de la représentation conceptuelle

- `class Display`
- `interface DrawableShape`
- ...

Couplage Modèle – Vue

```java
// Model
import view.Display ; // couplage M/V

class World{
  private ArrayList<Shape> shapes ;
  private Display d ; // couplage M/V

  public void add(Shape s){
    shapes.add(s) ;
  }
  public void setDisplay(Display d){
    this.d = d ;    // couplage M/V
  }
  public void changeWorld(){
    doChangeWorld() ;
    d.draw() ; // couplage M/V
  }
}
```

```java
// View
class Display{
  private World w ;

  public void draw(){
      /* dessine toutes les formes de l'
           objet World monde */
  }

  public static void main(String []args)
       {
    World w = new World() ;
    Display d = new Display(world) ;
    w.setDisplay(d) ; // couplage M/V
  }
}
```

Couplage Modèle – Vue

```
1  // Model
2  import view.Display ; // couplage M/V
3
4  class World{
5    private ArrayList<Shape> shapes ;
6    private Display d ; // couplage M/V
7
8    public void add(Shape s){
9      shapes.add(s) ;
10   }
11   public void setDisplay(Display d){
12     this.d = d ;    // couplage M/V
13   }
14   public void changeWorld(){
15     doChangeWorld() ;
16     d.draw() ; // couplage M/V
17   }
18 }
```

```
1  // View
2  class Display{
3    private World w ;
4
5    public void draw(){
6        /* dessine toutes les formes de l'
               objet World monde */
7    }
8
9    public static void main(String []args)
         {
10     World w = new World() ;
11     Display d = new Display(world) ;
12     w.setDisplay(d) ; // couplage M/V
13   }
14 }
```

- Problème : fort couplage entre le modèle et la vue

Modèle de conception : Observateur

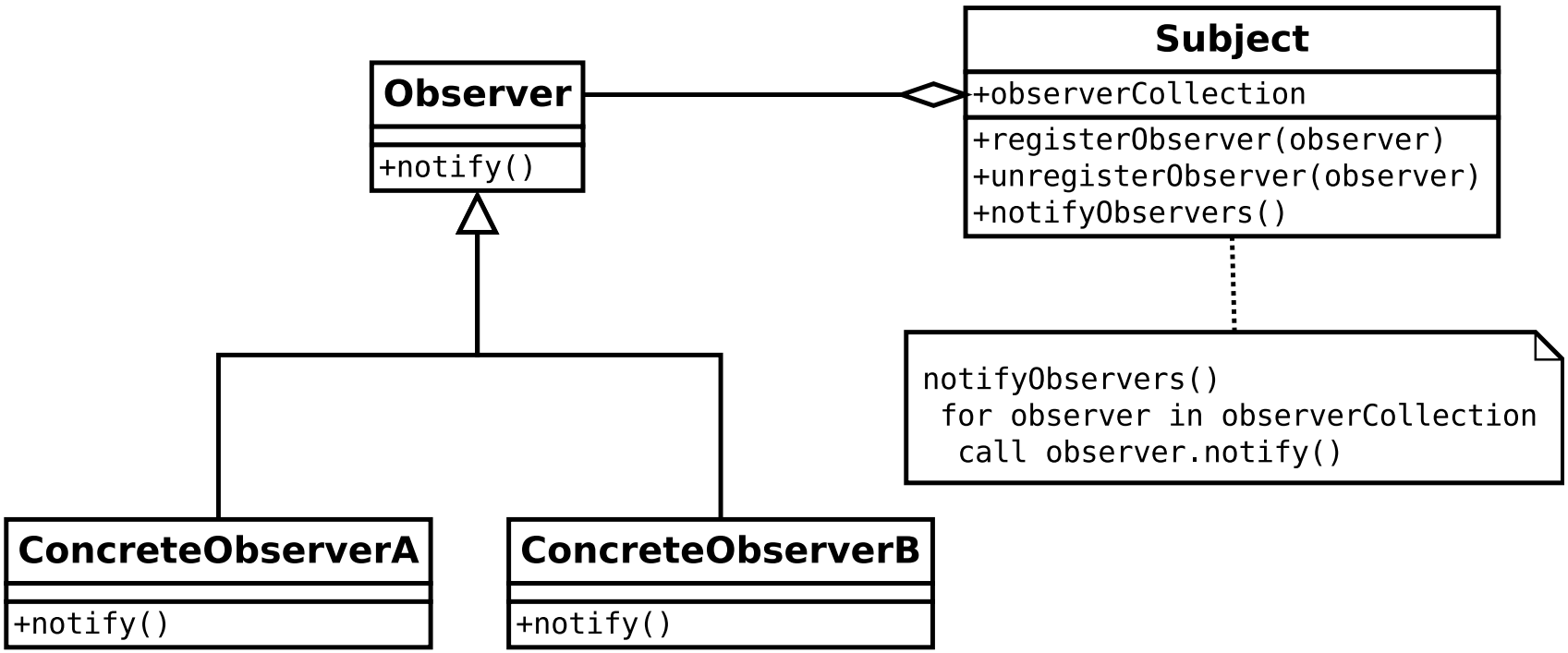


image : wikipedia

Couplage Modèle – Vue (avec Observateur)

```
1  // Model
2
3  class World extends Observable{
4    private ArrayList<Shape> shapes ;
5
6    public void add(Shape s){
7      shapes.add(s) ;
8    }
9    public void changeWorld(){
10     doChangeWorld() ;
11     notifyObservers() ;
12   }
13 }
```

```
1  // View
2  class Display implements Observer{
3    private World w ;
4
5    public void draw(){
6        /* dessine toutes les formes de l'
               objet World */
7    }
8    public void update(Observable o,
           Object arg){
9        draw() ; // Pas de couplage !
10   }
11
12   public static void main(String []args)
           {
13     World w = new World() ;
14     Display d = new Display(world) ;
15     world.addObserver(this) ;
16   }
17
18 }
```

Couplage Modèle – Vue (avec Observateur)

```java
// Model

class World extends Observable{
  private ArrayList<Shape> shapes ;

  public void add(Shape s){
    shapes.add(s) ;
  }
  public void changeWorld(){
    doChangeWorld() ;
    notifyObservers() ;
  }
}
```

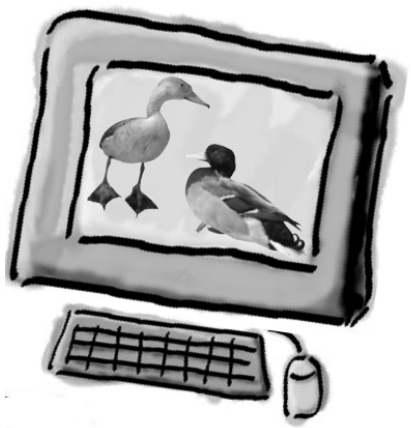
```java
// View
class Display implements Observer{
  private World w ;

  public void draw(){
      /* dessine toutes les formes de l'
           objet World */
  }
  public void update(Observable o,
       Object arg){
      draw() ; // Pas de couplage !
  }

  public static void main(String []args)
       {
    World w = new World() ;
    Display d = new Display(world) ;
    world.addObserver(this) ;
  }

}
```

- Couplage faible (le modèle est indépendant de la vue)

Cours 2 : Polymorphisme et compagnie

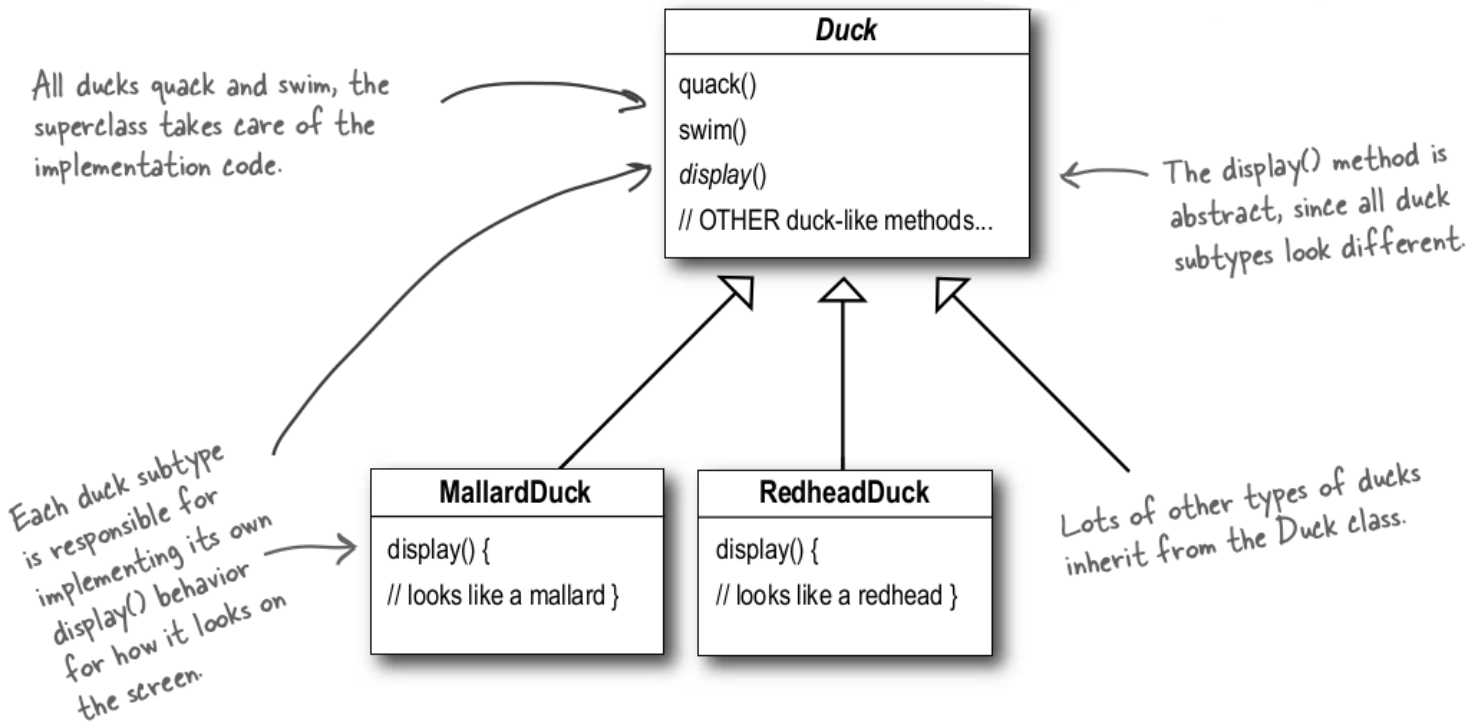
Canards

- Exemple illustré héritage / délégation / encapsulation / etc. (Livre Head first Design Patterns)
- Une entreprise fait un jeu de simulation de canards.



Implémentation initiale

- Une superclasse `Duck`, que tous les types de canards héritent.



Ajout d'une fonctionnalité

- L'entreprise veut ajouter la fonction **vol** aux canards.
- Comment faire ?

Ajout d'une fonctionnalité

- L'entreprise veut ajouter la fonction **vol** aux canards.
- Comment faire ?
- Facile ! Ajouter une méthode `fly()` dans `Duck` à côté des autres.

Ajout d'une fonctionnalité

- L'entreprise veut ajouter la fonction **vol** aux canards.
- Comment faire ?
- Facile ! Ajouter une méthode `fly()` dans `Duck` à côté des autres.
- Problème ?

Problème

Problème

- Tous les canards ne volent pas !

Problème

- Tous les canards ne volent pas !
- L'héritage donne la propriété à toutes les sous-classes, même celles qui ne doivent pas l'avoir !
- Bonne idée de réutiliser le code mais crée un problème !

Problème

- Tous les canards ne volent pas !
- L'héritage donne la propriété à toutes les sous-classes, même celles qui ne doivent pas l'avoir !
- Bonne idée de réutiliser le code mais crée un problème !
- Solution ?

Problème du vol d'un canard en plastique

- Solution ? Override la méthode `fly()` pour les canards en plastique pour ne rien faire ?

Problème du vol d'un canard en plastique

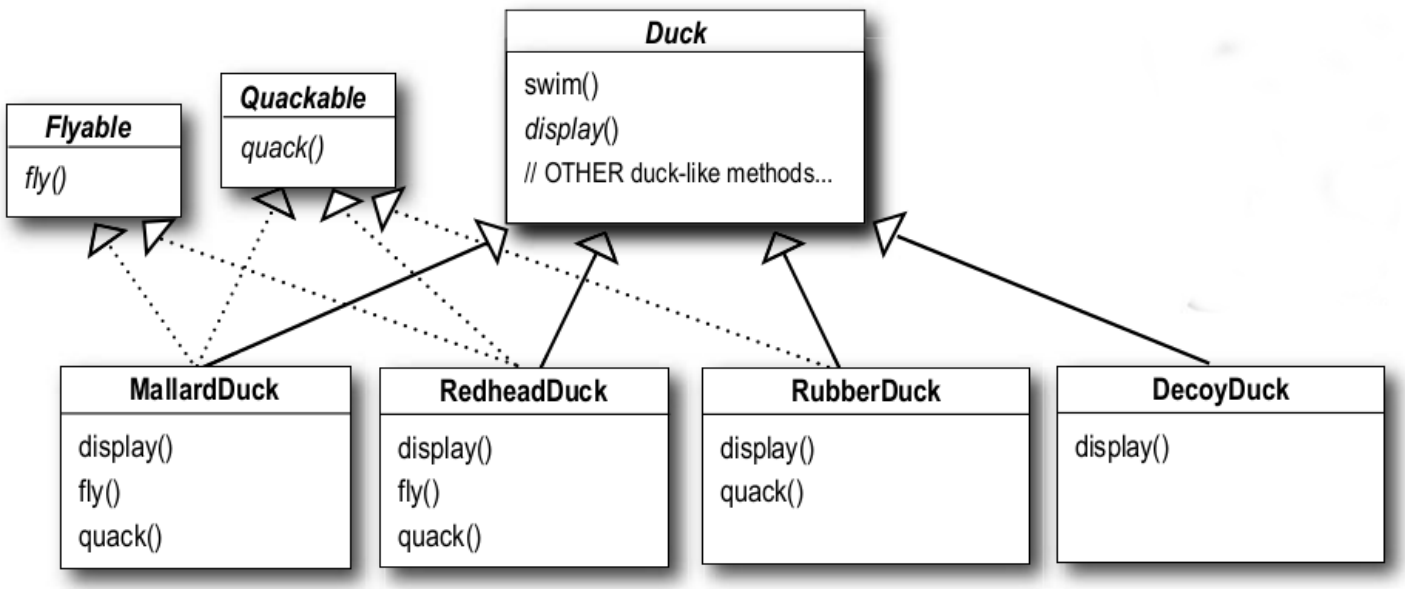
- Solution ? Override la méthode `fly()` pour les canards en plastique pour ne rien faire ?
- Mais si on ajoute maintenant le canard en bois (qui ne cancane même pas !) ?

Problème du vol d'un canard en plastique

- Solution ? Override la méthode `fly()` pour les canards en plastique pour ne rien faire ?
- Mais si on ajoute maintenant le canard en bois (qui ne cancane même pas !) ?
- Oublions l'héritage ! Interfaces ?

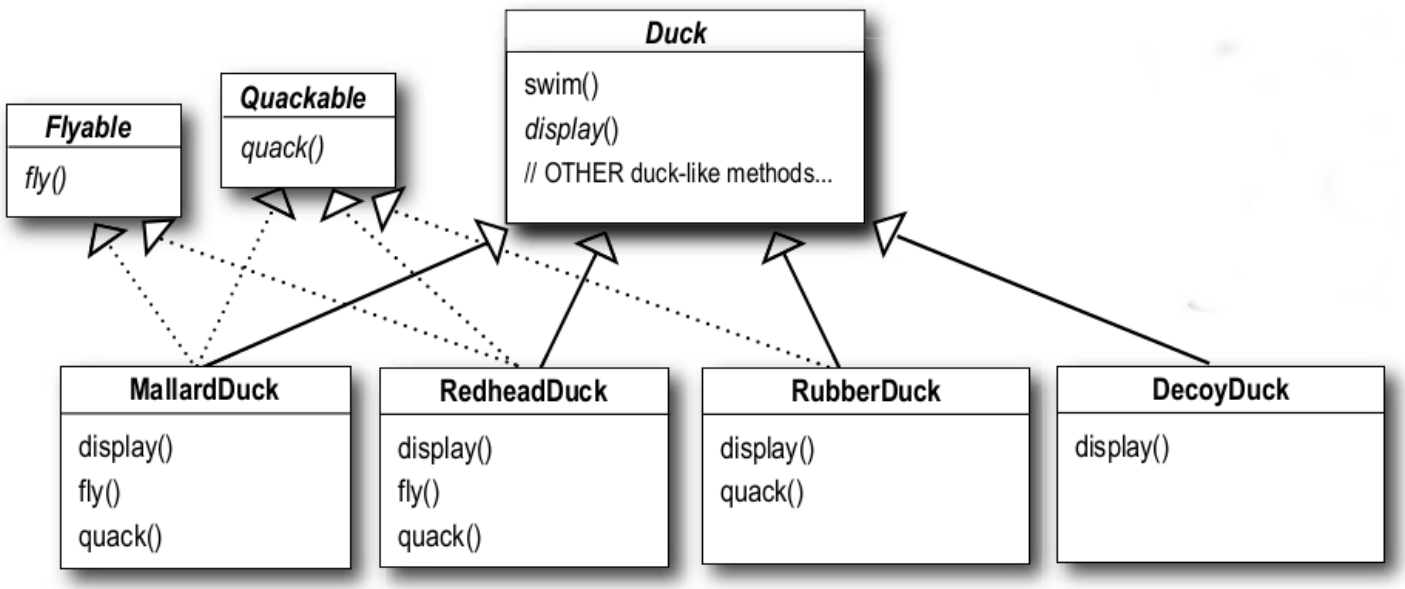
Interfaces

- Sortir `fly()` de `Duck` et faire une interface `Flyable` ? Seuls les canards pouvant voler implémentent l'interface ! Idem pour le cancanage !



Interfaces

- Sortir `fly()` de `Duck` et faire une interface `Flyable` ? Seuls les canards pouvant voler implémentent l'interface ! Idem pour le cancanage !



- Bien ?

Interfaces

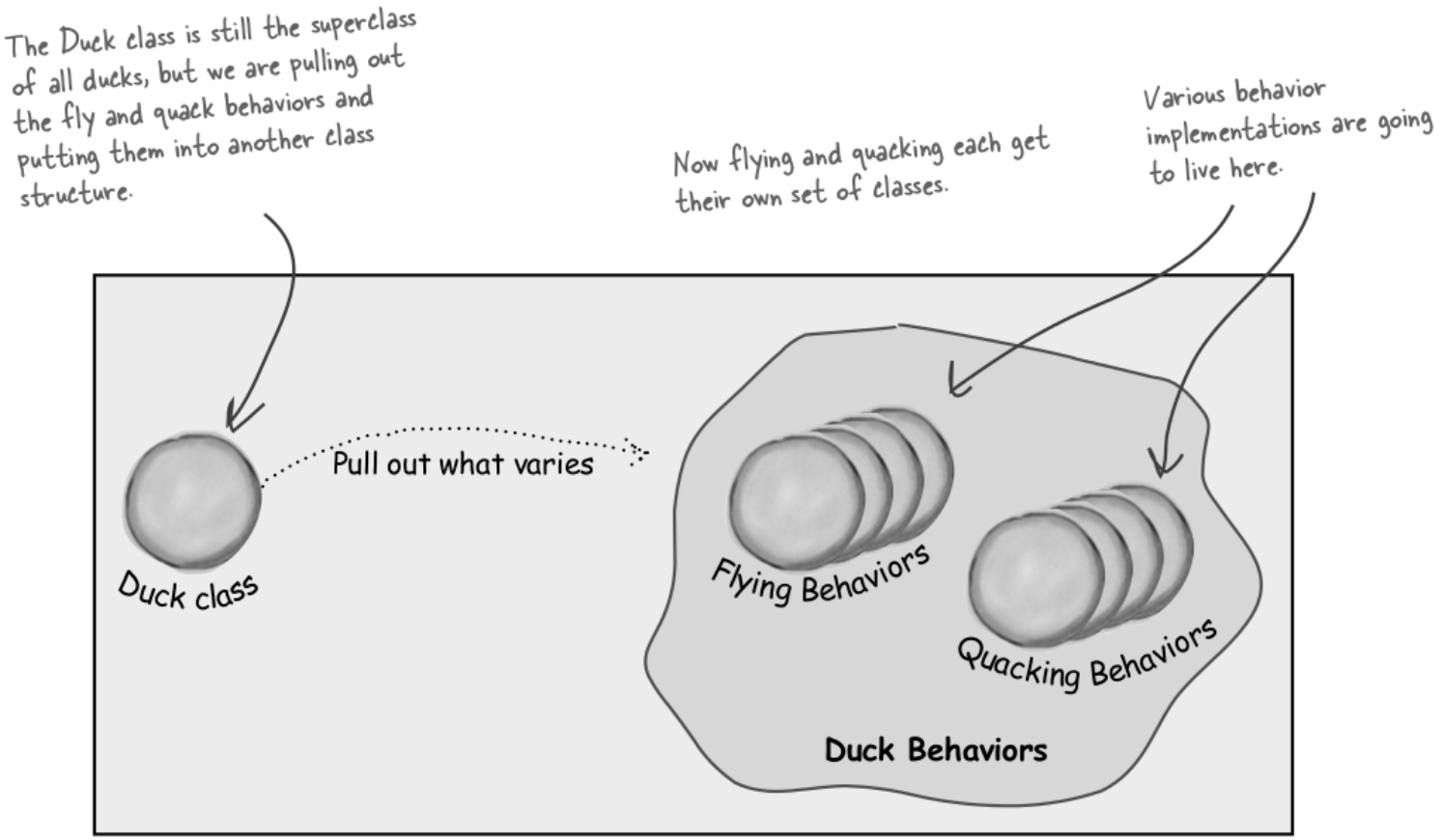
- Héritage était un problème car ajoutait des fonctionnalités non voulues selon les sous-classes.
- Mais ici on perd la réutilisation du code !!
- Doit implémenter/changer le code de voler 42 fois ! (pas de code dans les interfaces)

Interfaces

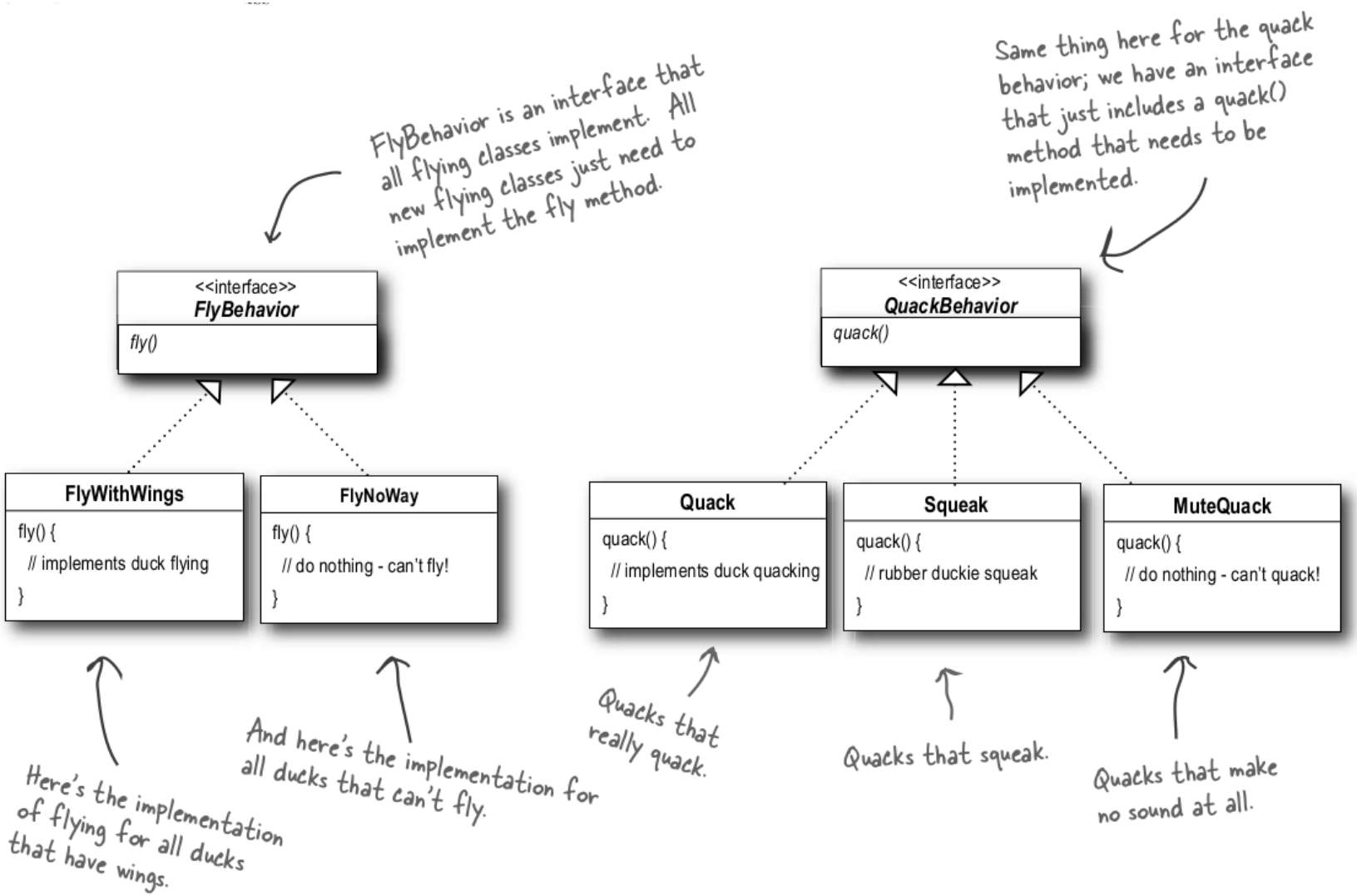
- Héritage était un problème car ajoutait des fonctionnalités non voulues selon les sous-classes.
- Mais ici on perd la réutilisation du code !!
- Doit implémenter/changer le code de voler 42 fois ! (pas de code dans les interfaces)
- Encapsulons ce qui change pour ne pas que ça impacte le reste du code !
  - Moins de conséquences en cas de changements !!

Interfaces

- On crée un ensemble de classe pour la fonctionnalité voler, un autre ensemble pour la fonctionnalité cancaner (ce qui peut varier).
- Duck garde ce qui reste identique.



Interfaces


FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly method.
Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.
<<interface>>
FlyBehavior
fly()
<<interface>>
QuackBehavior
quack()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
Here's the implementation of flying for all ducks that have wings.
And here's the implementation for all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.

Interfaces

- Les Behavior peuvent être réutilisés par d'autres classes sans la lourdeur de l'héritage : plus caché dans Duck !
- Existe ailleurs ?

Interfaces

- Les Behavior peuvent être réutilisés par d'autres classes sans la lourdeur de l'héritage : plus caché dans Duck !
- Existe ailleurs ?
- Sortable, Comparable, etc.
- Une sonnerie de téléphone qui utilise le son d'un canard...

Interfaces

- Les Behavior peuvent être réutilisés par d'autres classes sans la lourdeur de l'héritage : plus caché dans Duck !
- Existe ailleurs ?
- Sortable, Comparable, etc.
- Une sonnerie de téléphone qui utilise le son d'un canard...
- Si on veut avoir des canards qui volent avec une fusée ?

Interfaces

- Les Behavior peuvent être réutilisés par d'autres classes sans la lourdeur de l'héritage : plus caché dans Duck !
- Existe ailleurs ?
- Sortable, Comparable, etc.
- Une sonnerie de téléphone qui utilise le son d'un canard...

- Si on veut avoir des canards qui volent avec une fusée ?
- Ajoute une classe héritant de `FlyBehavior`.

Délégation

- Les canards **délèguent** leur comportement de vol et de cacanage au lieu d'utiliser une méthode définie sur eux.

```
1 public class Duck {
2   QuackBehavior quackbehavior ; //reference a ce qui implémente le comportement
3   FlyBehavior flyBehavior ;
4
5   public void performQuack() {
6     quackbehavior.quack() ; //delegation
7   }
8 }
```

- Pas besoin de savoir quel type d'objet va cancaner, juste de savoir comment le faire.

Délégation

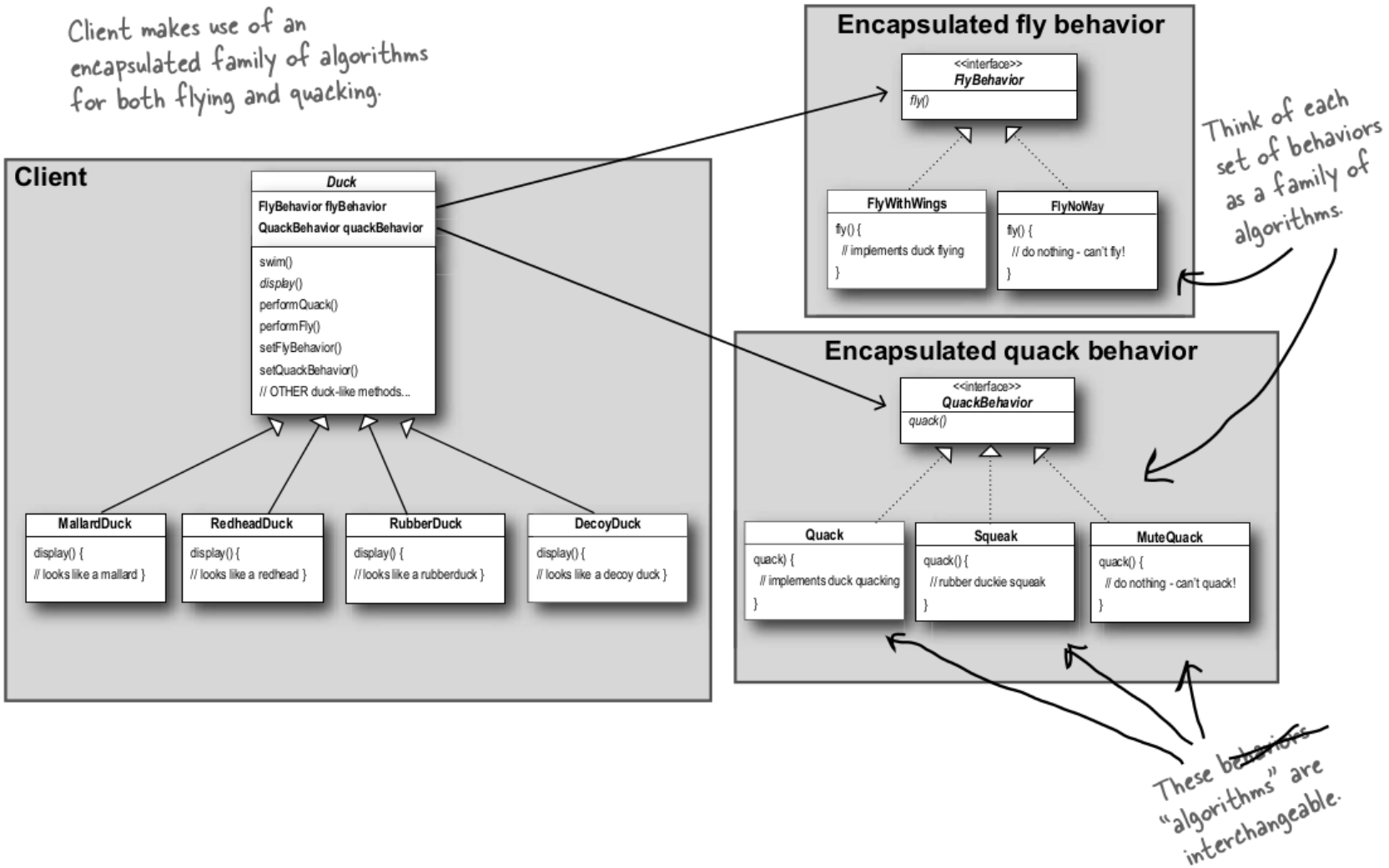
- Les canards **délèguent** leur comportement de vol et de cacanage au lieu d'utiliser une méthode définie sur eux.

```
1 public class Duck {
2   QuackBehavior quackbehavior ; //reference a ce qui implémente le comportement
3   FlyBehavior flyBehavior ;
4
5   public void performQuack() {
6     quackbehavior.quack() ; //delegation
7   }
8 }
```

- Pas besoin de savoir quel type d'objet va cancaner, juste de savoir comment le faire.

```
1 public class MallardDuck extends Duck {
2   //hérite des champs de Duck
3
4   public MallardDuck() {
5     quackBehavior = new Quack() ; //le performQuack utilisera cette implémentation
          de QuackBehavior
6     flyBehavior = new FlyWithWings() ;
7   }
8 }
```

Au final...



- (Au passage) c'est le pattern stratégie...

Cours 2 : Polymorphisme et compagnie

Test d'égalité

- Que testent les opérateurs == et != ?

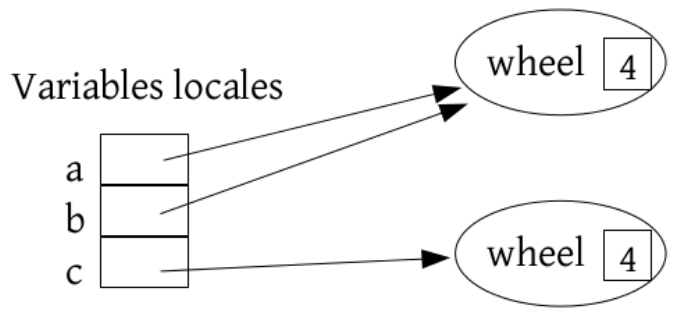
Test d'égalité

- Que testent les opérateurs == et != ?
- Ca dépend du type !
  - Types primitifs : leurs valeurs.
  - Types objets : la valeur des **références** !

Test d'égalité

- Que testent les opérateurs == et != ?
- Ca dépend du type !
  - Types primitifs : leurs valeurs.
  - Types objets : la valeur des **références** !

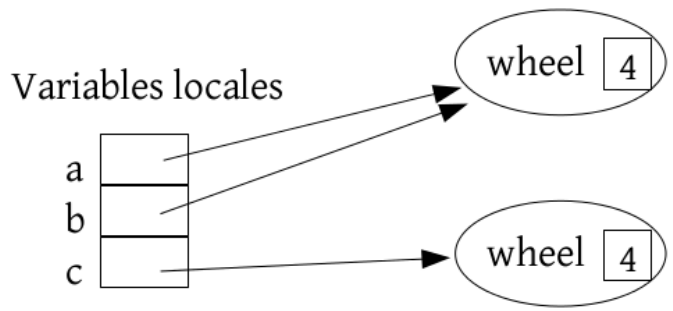
```
1 Truck a = new Truck(4) ;
2 Truck b = a ;
3 Truck c = new Truck(4) ;
4
5 syso(a==b, b==c, a==c) ;
```



Test d'égalité

- Que testent les opérateurs == et != ?
- Ca dépend du type !
  - Types primitifs : leurs valeurs.
  - Types objets : la valeur des **références** !

```
1 Truck a = new Truck(4) ;
2 Truck b = a ;
3 Truck c = new Truck(4) ;
4
5 syso(a==b, b==c, a==c) ;
```



- Vrai, Faux, Faux.

Test d'égalité

- Pour comparer champ à champ deux objets, redéfinir `Object.equals(Object)`.
- La méthode `equals` d'`Object` teste les références ! (juste un ==).
- La plupart des classes de l'API redéfinissent `equals`.

```
1    String s1 = new String("MIAGE") ;
2    String s2 = new String("MIAGE") ;
3    System.out.println(s1==s2) ; //false
4    System.out.println(s1.equals(s2)) ; //true
```

Redéfinir equals

- Comme définie sur `Object`, la méthode redéfinie doit prendre un `Object` en argument !
- Doit renvoyer `false` si l'objet en argument n'est pas de la même classe que l'objet courant.
  - (Un des) seul cas où on peut utiliser `instanceof`.

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.
- Une sous-classe est un type d'une super-classe, donc :

```
1 String s = "Miage" ;
2 if(s instanceof Object) { ...} //vrai
```

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.
- Une sous-classe est un type d'une super-classe, donc :

```
1 String s = "Miage" ;
2 if(s instanceof Object) { ...} //vrai
```

```
1 String s = "Miage" ;
2 if(s instanceof Integer) { ...} //compile pas
```

instanceof

- Format : `objectReference instanceof type`, renvoie un boolean.

```
1 String s = "Miage" ;
2 if(s instanceof String) { ...}
3 Object o = new String("egaiM") ;
4 if(o instanceof String) {...}
5 if(o instanceof Integer) {...}
```

- Vrai/Faux ?
  - vrai, vrai, faux.
- Une sous-classe est un type d'une super-classe, donc :

```
1 String s = "Miage" ;
2 if(s instanceof Object) { ...} //vrai
```

```
1 String s = "Miage" ;
2 if(s instanceof Integer) { ...} //compile pas
```

- Attention :

```
1 String s = null ;
2 if(s instanceof String) { ...} //faux
```

instanceof et getClass

- `getClass()` permet d'obtenir la classe d'un objet à l'exécution.

```
1 Object o = new Integer(3) ;
2 o.getClass() == Integer.class ; //true
```

instanceof et getClass

- `getClass()` permet d'obtenir la classe d'un objet à l'exécution.

```
1 Object o = new Integer(3) ;
2 o.getClass() == Integer.class ; //true
```

- Différence entre ces if ?
- Indice : `Number` est une classe abstraite...

```
1 Object o = new Integer(3) ;
2 if(o.getClass() == Number.class) {...
3 if(o instanceof Number) { ...
```

instanceof et getClass

- `getClass()` permet d'obtenir la classe d'un objet à l'exécution.

```
1 Object o = new Integer(3) ;
2 o.getClass() == Integer.class ; //true
```

- Différence entre ces if ?
- Indice : `Number` est une classe abstraite...

```
1 Object o = new Integer(3) ;
2 if(o.getClass() == Number.class) {...
3 if(o instanceof Number) { ...
```

- `instanceof` teste si la classe de l'instance est la classe demandée ou une sous-classe.
- `.getClass() ==` est une égalité entre classes.
- `.getclass() == Toto` toujours faux si Toto est une interface ou une classe abstraite !

Pourquoi redéfinir equals

- L'API des collections utilise `equals` (déterminer si l'objet est présent dans la collection).
- Si `equals` non redéfini, test, ajout etc de l'objet dans une collection risque de bugger.

Comment redéfinir equals

```
1  public class Car {
2    private final String brand ;
3    private final int nbDoors ;
4
5    @Override
6    public boolean equals(Object o) { //Object sinon pas de redef.
7      if( !(o instanceof Car)) return false ;
8      Car c = (Car)o ; //cast safe
9      return ... ;
10   }
11 }
```

Comment redéfinir equals

```
1  public class Car {
2    private final String brand ; //object
3    private final int nbDoors ; //primitf
4
5    @Override
6    public boolean equals(Object o) {
7      if( !(o instanceof Car)) return false ;
8      Car c = (Car)o ;
9      return nbDoors==c.nbDoors && brand.equals(c.brand) ;
10     //== pour primitifs, equals pour objets
11     //les champs privés de c sont accessibles
12   }
13 }
```

Comment redéfinir equals – astuces

```
1  public class Car {
2    private final String brand ;
3    private final int nbDoors ;
4
5    @Override
6    public boolean equals(Object o) {
7      if(this==o) return true ; //raccourci
8      if( !(o instanceof Car)) return false ;
9      Car c = (Car)o ;
10     return nbDoors==c.nbDoors && brand.equals(c.brand) ;
11     //&& paresseux, primitifs d'abord
12   }
13 }
```

Hashcode

- `o.hashCode()` renvoie un entier qui "résume" l'objet `o`.
- Utilisé par l'API des collections basés sur les tables de hachage (HashMap, HashSet...).
- Doit bien couvrir l'ensemble des entiers possibles, sinon perd l'intérêt de la table de hachage.
- Rappel table de hash.

Hashcode et equals

- Si `o1.equals(o2) == true`, alors on doit avoir `o1.hashCode() == o2.hashCode()`.
- Inverse pas vrai !
- Si equals est redéfini, hashCode doit l'être aussi !

Equals sans Hashcode

```
1  public class Phone {
2    private final int areaCode ;
3    private final int number ;
4
5    public Phone(int ac, int n) {
6      areaCode = ac ;
7      number=n ;
8    }
9
10   @Override public boolean equals(Object o) {
11     if(o==this)return true ;
12     if( !(o instanceof Phone)) return false ;
13     Phone pn = (Phone)o ;
14     return areaCode==pn.areaCode && number==pn.number ;
15   }
16
17   public static void main(String[] args) {
18     Map<Phone, String> m = new HashMap<Phone, String>() ;
19     m.put(new Phone(0033, 123456), "Cornaz") ;
20
21     System.out.println(m.get(new Phone(0033, 123456))) ;
22   }
23 }
```

Equals sans Hashcode

- Affiche quoi ?

Equals sans Hashcode

- Affiche quoi ?
- `null`...
- Deux instances de Phone sont utilisées (ajout puis recherche) : `hashCode` d'objet est différent : pas dans le même sac.

Hashcode

- Le but est d'éviter au maximum les collisions et de bien répartir (pour les objets utilisés dans le programme).
- Doit être rapide a calculer ! (possibilité de le stocker dans le cas d'un objet non mutable !)
- On peut utiliser un XOR entre des valeurs de hash déjà bien calculées.
- Éventuellement un `Integer.rotateLeft` pour décaler des bits de façon circulaire.

Hashcode

```
1  public class Car {
2    private final String brand ;
3    private final int nbDoors ;
4
5    /* très mauvasis hashCode
6    @Override
7    public int hashCode() {
8      return 42 ;
9    }
10   */
11
12   /* mauvais hashcode
13   @Override
14   public int hashCode() {
15     return nbDoors+brand.hashCode() ;
16   }
17   */
18
19   @Override
20   public int hashCode() {
21     return nbDoors^Integer.rotateLeft(brand.hashCode(), 16) ;
22   }
23 }
```

Switch de strings

- En java, `switch` sur les `String` possible.
- Le compilateur calcule le hashCode pour chaque chaîne.
- Puis equals car possible d'avoir un même hashCode pour 2 chaines différentes.

Switch de strings

```
1  String s = ...
2  switch(s){
3    case "Chimon" :
4      ...
5    case "Roland" :
6      ...
7  }
```

```
1        switch(s.hashCode()) {
2        case "Chimon".hashCode() :
3          if(s.equals("Chimon")) {
4
5          }
6        case "Roland".hashCode() :
7          if(s.equals("Roland")) {
8
9          }
10       }
```