

Héritage (2)

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1



Université
Lille1
Sciences et Technologies

UFR IEEA
Formations en
Informatique de
Lille 1

Exemple : Dessin de Formes

- On souhaite réaliser une application de dessin permettant, entre autres, la manipulation de différentes formes graphiques : cercles, triangles, rectangles, polygônes, carrés, etc.
- Un objet `Feuille` gère l'état courant du dessin,
- L'application dispose d'outils permettant de déplacer la forme sélectionnée, d'en changer la taille, de lui appliquer une rotation, etc.

`Feuille` ou chacune des classes décrivant les outils doivent pouvoir être implémentées **indépendamment** des formes existantes

↪ on doit pouvoir **facilement** ajouter des formes à l'application.

(Open Close Principle)

```

public class Feuille {
    protected List<Shape> lesFormes;
    public void repaint() {
        for(Shape s : this.lesFormes) {
            s.draw();
        }
    }
}
public class MoveTool {
    public void move(Shape selectedShape) {
        selectedShape.move(...);
    }
}
public class RotateTool {
    public void rotate(Shape selectedShape) {
        selectedShape.rotate(...);
    }
}
public class ResizeTool {
    public void resize(Shape selectedShape) {
        selectedShape.resize(...);
    }
}

```

Il est nécessaire de disposer d'un abstraction `Shape`

Interface ou héritage ?

interface il n'est pas possible de définir un comportement pour les méthodes `draw`, `rotate`, `resize` ou d'autres de `Shape`.

héritage Un certain nombre de comportements et d'attributs peuvent être définis de manière commune pour toutes les formes : couleur du trait, motif du remplissage, épaisseur du trait, gestion du plan de profondeur d'affichage, etc.

Autre possibilité, utiliser une approche hybride :

une **classe abstraite**

Classes abstraites

Une classe abstraite

- est une classe (comporte donc des attributs et des méthodes),
- déclare de méthodes sans comportement attaché, ces méthodes sont dites **méthodes abstraites**,
- ne peut pas créer d'instance (même si elle peut déclarer un constructeur),
- les classes qui en héritent doivent concrétiser le comportement de **toutes** les méthodes abstraites
↪ sous peine d'être elles aussi abstraites

abstract

- déclaration classe abstraite : ajouter le qualificatif `abstract` avant `class` dans l'entête de déclaration.

```
public abstract class ClasseAbstraite ... {  
    ...  
}
```

abstract

- déclaration classe abstraite : ajouter le qualificatif **abstract** avant `class` dans l'entête de déclaration.
- déclaration méthode abstraite : ajouter le qualificatif **abstract** avant le type de retour dans la signature de la méthode et ne donner aucun corps.

```
public abstract class ClasseAbstraite ... {  
    ...  
    public abstract Type methodeAbstraite(...);  
}
```

abstract

- déclaration classe abstraite : ajouter le qualificatif **abstract** avant `class` dans l'entête de déclaration.
- déclaration méthode abstraite : ajouter le qualificatif **abstract** avant le type de retour dans la signature de la méthode et ne donner aucun corps.

```
public abstract class ClasseAbstraite ... {
    ...
    public abstract Type methodeAbstraite(...);
}
```

<< abstract >> <i>ClasseAbstraite</i>
...
... + <i>methodeAbstraite</i>

Retour en Forme

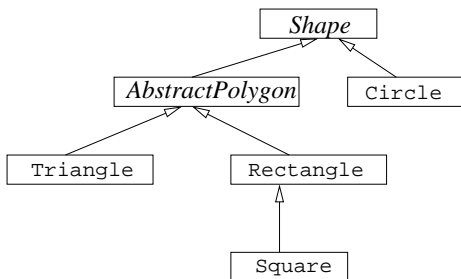
```

public abstract class Shape {
    protected int lineWidth;
    protected FillStyle fillStyle;
    protected byte depth;
    public void setLineWidth(int width) { this.lineWidth = width }
    public void toFront() { depth = 0; }
    public FillStyle getFillStyle() { return fillStyle; }

    public abstract void draw();
    public abstract void rotate(double angle);
    public abstract void move(...);
    public abstract void resize(...);
}

public class Circle extends Shape {
    protected Point center;
    protected int radius;
    public void draw() { ... code ... }
    public void rotate(double angle) { ... code ... }
    public void move(...) { ... code ... }
    public void resize(...) { ... code ... }
    public Point getCenter() { ... code ... }
}

```

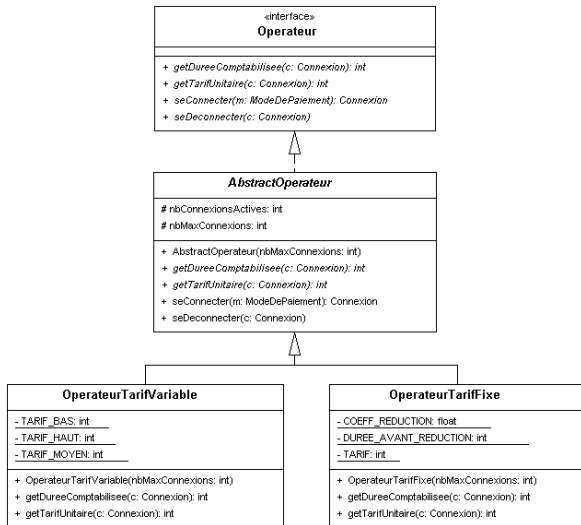


```

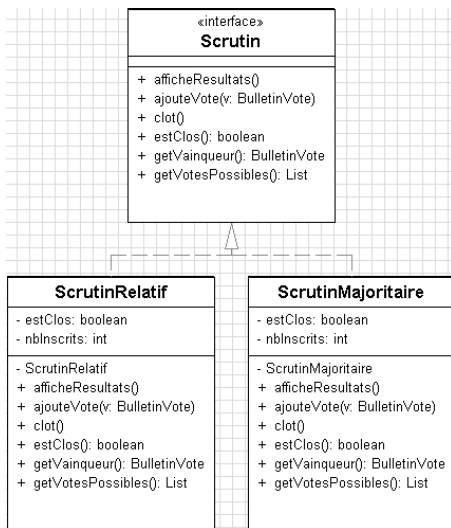
public abstract class Shape { ... }

public class Circle extends Shape { ... }
public abstract class AbstractPolygon extends Shape {
    protected List<Edge> edges;
    public abstract int getNumberOfEdges();
    ...
}
// les deux classes suivantes fournissent une définition pour getNumberOfEdges()
// (et les autres méthodes abstraites de Shape)
public class Triangle extends AbstractPolygon { ... }
public class Rectangle extends AbstractPolygon { ... }
// réutilise le getNumberOfEdges() de Rectangle
public class Square extends Rectangle { ... }
  
```

Opérateurs



Scrutins



Scrutins

```

...
public abstract class AbstractScrutin implements Scrutin {
    protected boolean estClos;
    public boolean estClos() {
        return this.estClos;
    }
    ...
    public abstract BulletinVote getVainqueur();
}

public class ScrutinMajoritaire extends AbstractScrutin {
    public BulletinVote getVainqueur() {
        ... TRAITEMENT Majoritaire...
    }
}

public class ScrutinRelatif extends AbstractScrutin {
    public BulletinVote getVainqueur() {
        ... TRAITEMENT Relatif...
    }
}

```

Clients bancaires

```

...
public abstract class Client {
    protected String adresse;
    protected List<Compte> comptes;
    public Client(String adresse) {
        this.adresse = adresse; this.comptes = new ArrayList<Compte>();
    }
    public abstract Identite getIdentite();
    public String getAdresse() { return this.adresse; }
    public List<Compte> getComptes() { return this.comptes; }
    public void ajouteCompte(Compte nouveau) { this.comptes.add(nouveau); }
    public void addEcriture(...) throws UnsupportedOperationException { ... }
}
public class ClientPhysique extends Client {
    protected PersonneId pid;
    public ClientPhysique(String adresse, PersonneId id) {
        super(adresse);
        this.id = id;
    }
    public Identite getIdentite() { return this.pid; }
    ...
}
public class ClientEntreprise extends Client {
    protected EntrepriseId eid;
    protected String SIRET;
    public Identite getIdentite() { return this.eid; }
    ...
}

```

Retour sur les enum

Méthodes différenciées pour chaque constante

```

package operation;
import java.util.*;
public enum Operation {
    plus {
        public double eval(double x, double y) { return x + y; }
    },
    minus {
        public double eval(double x, double y) { return x - y; }
    },
    times {
        public double eval(double x, double y) { return x * y; } },
    divided_by {
        public double eval(double x, double y) { return x / y; } };

    public abstract double eval(double x, double y);
    //-----
    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values()) {
            System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
        }
    }
}

```

Retour sur les enum

Méthodes différenciées pour chaque constante

```

package operation;
import java.util.*;
public enum Operation {
    plus {
        public double eval(double x, double y) { return x + y; }
    },
    minus {
        public double eval(double x, double y) { return x - y; }
    },
    times {
        public double eval(double x, double y) { return x * y; } },
    divided_by {
        public double eval(double x, double y) { return x / y; } };

    public abstract double eval(double x, double y);
    //-----
    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values()) {
            System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
        }
    }
}

```

java operation.Operation 6 3
 6.0 plus 3.0 = 9.0
 6.0 minus 3.0 = 3.0
 6.0 times 3.0 = 18.0
 6.0 divided by 3.0 = 2.0

Dans l'API Java

- dans `java.util`

```
public interface Collection<E> { ... }
public abstract class AbstractCollection<E> implements Collection<E> { ... }
public abstract class AbstractSet<E> extends AbstractCollection<E> { ... }
public abstract class AbstractList<E> extends AbstractCollection<E> { ... }
public class ArrayList<E> extends AbstractList<E> { ... }
public class LinkedList<E> extends AbstractList<E> { ... }
```

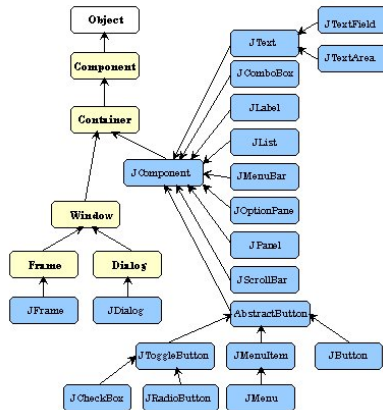
- dans `javax.swing`

```
public abstract class JComponent extends java.awt.Container { ... }
public class JPanel extends JComponent { ... }
public abstract class AbstractButton extends JComponent { ... }
public class JButton extends AbstractButton { ... }
public class JMenuItem extends AbstractButton { ... }
```

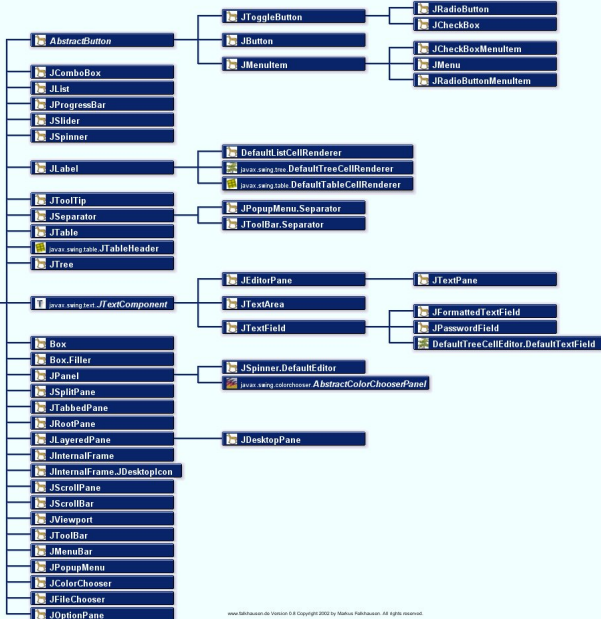
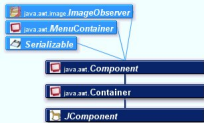
Hiéarchies de composants graphiques

java.awt et javax.swing

- **factorisation** des comportements et propriétés des différents éléments
 ↪ position, taille, visibilité, gestion des événements, etc.
- **polymorphisme**
 ↪ tout les composants swing sont du type "JComponent"



javax.swing.* JComponents



Héritage d'interfaces

- il est possible de définir une interface héritant d'une autre interface.
- comme pour les classes, on peut *étendre* par héritage.

```
public interface Aquatique {
    public void nage();
}
public interface Marin extends Aquatique {
}
public interface Predateur {
    public void chasse();
}
public interface PredateurMarin extends Marin, Predateur {
    public float kiloPoissonsMangesParJour();
}
```

- intérêt : typage mutiple = intersection de types.

```

public interface Animal { }
public interface Aquatique { }
public interface AnimalAquatique extends Animal, Aquatique { }
public class Zoo {
    public void addAnimal(Animal animal) { ... }
}
public void Aquarium {
    public void add(Aquatique aquatique) { ... }
}
public class ZooAvecAquarium extends Zoo {
    protected Aquarium aquarium;
    public void addAnimalAquatique(AnimalAquatique animalAqua) {
        this.addAnimal(animalAqua);
        this.aquarium.add(animalAqua);
    }
}
public class Mammifere implements Animal { ...}
public class Cetace extends Mammifere implements AnimalAquatique { ... }

public class Nenuphar implements Aquatique { ... }
    // ... utilisation ...
ZooAvecAquarium zoo = new ZooAvecAquarium();
Cetace cet = new Cetace();
zoo.addAnimalAquatique(cet);
zoo.getAquarium().add(new Nenuphar());

```

final

- il est possible d'interdire l'héritage d'une classe
- il suffit de mentionner le modificateur `final` dans l'entête de déclaration de classe.

```
public final class ClassePasHeritable {  
    ...  
}
```

`java.lang.String`, `java.lang.Boolean`, `java.net.InetAddress`, etc.

- il est possible d'interdire la surcharge (redéfinition) d'une méthode
- il suffit de mentionner le modificateur `final` dans la signature de la méthode.

```
public class UneClasse {
    public final void uneMethode() { ... }
}
public class UneSousClasse extends UneClasse {
    // ne peut redéfinir public void uneMethode()
}
```

```
    dans classe java.applet.Applet
        public final void setStub(AppletStub stub)
    public static final AudioClip newAudioClip(URL url)
```

`final` ~ “constant”

Application

Garantir le code exécuté indépendamment de tout héritage possible.

Dans une classe `Scrutin`.

Garantir l'anonymat du votant quelque soit le type de `Scrutin` :

```
public final void vote(Votant votant)
    throws VoteImpossibleException {
    if (this.peutVoter(votant)) {
        Vote vote = votant.vote(this.lesVotesPossibles);
        this.enregistreVote(vote);
        this.aVote(votant);
    }
    else {
        throw new VoteImpossibleException ();
    }
}
private final void enregistreVote(Vote vote) {
    this.lesVotes.add(vote);
}
```

Multi-héritage ?

- faire hériter une classe **simultanément** de classes et récupérer ainsi les comportements de chacune

```
public class Mammifere extends Animal {
    public String organeDeRespiration() {
        return "poumons";
    }
}
public class Nageur {
    public void nage() { ... }
}
public class Cetace extends Mammifere, Nageur { // interdit en Java
}
// ...utilisation...
Cetace cet = new Cetace();
System.out.println(cet.organeDeRespiration());
cet.nage();
Mammifere mam = cet;           // upcast possible vers l'une ou
Nageur nageur = cet;          // l'autre des classes
```

Attention

L'héritage multiple de classes **n'existe pas** en JAVA

Problème

- Quelle définition choisir en cas d'une déclaration **multiple** dans les super-classes ?

```
public class Mammifere extends Animal {
    public int getSize() { ... }
}
public class Nageur {
    public int getSize() { ... }
}
public class Cetace extends Mammifere, Nageur {
}
// ...utilisation...
Cetace cet = new Cetace();
System.out.println(cet.getSize()); // Quelle définition de méthode ?
```

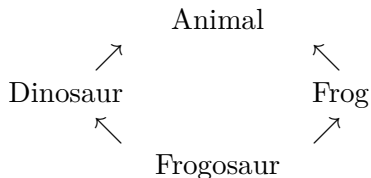
- une solution (?) : imposer le “cast”

```
Cetace cet = new Cetace();
System.out.println((Mammifere) cet.getSize());
ou
System.out.println((Nageur) cet.getSize());
```

Le problème du diamant

ou “*le losange de la mort*”

(d'après B. Venners)



Jurassic Park (M. Crichton)

```

public abstract class Animal {
    public abstract void talk();
}
public class Frog extends Animal {
    public void talk() {
        System.out.println("Ribit, Ribit");
    }
    public void jump() { ... }
}
public class Dinosaur extends Animal {
    public void talk() {
        System.out.println("I'm a poor lonesome dinosaur");
    }
    public boolean isCarnivorous() { ... }
}
public class Frogosaur extends Frog,Dinosaur {
    // impossible en Java
}
// ... utilisation ...
Animal animal = new Frogosaur();
animal.talk(); //!!! ambiguïté!!!

```

Le problème du diamant est lié au multi-héritage de 2 classes qui descendent de la même super-classe, lors de l'upcast vers la super-classe commune (ici “super-super-classe”)

Quelle définition de `talk()` choisir ?

Le problème apparaît aussi pour d'éventuels attributs dans `Animal` :

Quelle copie de l'attribut considérer ? (y a-t-il 2 copies ?)

en Java

- héritage **simple** de classe
- implémentation **multiple** des interfaces.

Utiliser les interfaces

Choisir l'héritage de l'une des deux classes et définir une interface pour la partie spécifique à l'autre et que l'on souhaitait récupérer.

Utiliser les interfaces

Choisir l'héritage de l'une des deux classes et définir une interface pour la partie spécifique à l'autre et que l'on souhaitait récupérer.

```
public abstract class Animal {
    public abstract void talk();
}
class Frog extends Animal {
    public void talk() {
        System.out.println("Ribit, Ribit");
    }
    public void jump() { ... }
}
interface DinosaurInterface {
    public boolean isCarnivorous();
}
public class Frogosaur extends Frog
    implements DinosaurInterface {
    public boolean isCarnivorous() { ... }
}
// ... utilisation...
Frogosaur frogosaur = new Frogosaur();
Animal animal = frogosaur;
animal.talk();
Frog frog = frogosaur;
DinosaurInterface din = frogosaur;
```

Utiliser les interfaces

Choisir l'héritage de l'une des deux classes et définir une interface pour la partie spécifique à l'autre et que l'on souhaitait récupérer.

```
public abstract class Animal {
    public abstract void talk();
}
class Frog extends Animal {
    public void talk() {
        System.out.println("Ribit, Ribit");
    }
    public void jump() { ... }
}
interface DinosaurInterface {
    public boolean isCarnivorous();
}
public class Frogosaur extends Frog
    implements DinosaurInterface {
    public boolean isCarnivorous() { ... }
}
// ... utilisation...
Frogosaur frogosaur = new Frogosaur();
Animal animal = frogosaur;
animal.talk();
Frog frog = frogosaur;
DinosaurInterface din = frogosaur;
```

```
public abstract class Animal {
    public abstract void talk();
}
interface FrogInterface {
    public void jump();
}
class Dinosaur {
    public void talk() {
        System.out.println("Ribit, Ribit");
    }
    public boolean isCarnivorous() { ... }
}
public class Frogosaur extends Dinosaur
    implements FrogInterface {
    public void jump() { ... }
}
// ... utilisation ...
Frogosaur frogosaur = new Frogosaur();
Animal animal = frogosaur;
animal.talk();
FrogInterface frog = frogosaur;
Dinosaur din = frogosaur;
```

Combiner interfaces et héritage

Accroître le polymorphisme

tout en évitant les problèmes liés au multi-héritage.

Permettre de “multi-typer” les objets
et
d’accroître l’**abstraction** grâce à l’upcast

```

public interface Aquatique {
    public void nage();
}
public class Cetace extends Mammifere implements Aquatique {
    public void nage() { ... }
}
public class Zoo {
    public void addAnimal(Animal animal) { ... }
}
public class Aquarium {
    public void addAquatique(Aquatique aquatique) { ... }
}
public class ZooAvecAquarium extends Zoo {
    protected Aquarium aquarium;
    public void addCetace(Cetace cetace) {
        // à la fois un Animal et un Aquatique
        this.addAnimal(cetace);
        this.aquarium.addAquatique(cetace);
    } }

// ... utilisation ...
ZooAvecAquarium zoo = new ZooAvecAquarium();
Cetace cet = new Cetace();
zoo.addCetace(cet);

```

Exemple

```

public interface Talkative {
    public void talk();
}
public abstract class Animal implements Talkative { ... }
public class Frog extends Animal {
    public void talk() {
        System.out.println("Ribit, Ribit");
    }
}
public class Dog extends Animal {
    public void talk() {
        System.out.println("Ouah, Ouah");
    }
}
public class Interrogator {
    public void makeItTalk(Talkative subject) {
        subject.talk();
    }
} // Interrogator

```

```

public class CuckooClock extends Clock implements Talkative {
    public void talk() {
        System.out.println("Ah que coucou!!!");
    }
    public Time getTime() {
        ...
    }
    public static void main(String[] args){
        (new Interrogator()).makeItTalk(new CuckooClock());
    }
} // CuckooClock

```

- **N'importe quelle classe** peut potentiellement implémenter Talkative et ainsi être passée en argument de makeItTalk

Héritage d'interfaces

- Rappel : le multi-héritage est possible avec les interfaces. (dans ce cas pas de conflit en cas de définitions multiples)

```
public interface Aquatique {
    public void nage();
}
public interface Marin extends Aquatique {
}
public interface Predateur {
    public void chasse();
}
public interface PredateurMarin extends Marin, Predateur {
    public int kiloPoissonsMangesParjour();
}
```

Composition et Héritage

Deux manières de réutiliser

- Héritage relation *is-a*
- Composition relation *has-a*

Réutiliser : héritage

Avantages :

- liaison dynamique et polymorphisme (utilisation de upcast)

Intérêts :

- modification de code en remplaçant un objet de classe C par une instance d'une sous-classe de C

```
AbstractList<T> l = new ArrayList<T>();
```

peut devenir

```
AbstractList<T> l = new LinkedList<T>();
```

- Mais seul ce type de modification par introduction de nouvelles sous-classes est facilité.

fragilité l'interface de la super-classe

super-classe **fragile**

⇒ La plus légère des modifications dans la superclasse peut corrompre le comportement des sous-classes.

En fait c'est l'**interface** de la super-classe qui est sensible/fragile

- On dit que l'héritage réalise une *encapsulation faible* : un changement de l'interface de la super-classe corrompt le code qui utilise la sous-classe

Exemple

```
public class Fruit {
    public int peel() {
        System.out.println("peel done");
        return 1;
    }
}

public class Apple extends Fruit { } // héritage

public class Example1Fruit {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
} // Example1Fruit
```

Changement de l'interface

```

public class Peel {
    private int numberOfPeels;
    public Peel(int numberOfPeels) {
        this.numberOfPeels = numberOfPeels;
    }
    public int getNumberOfPeels() { return numberOfPeels; }
}

public class Fruit {
    public Peel peel() {
        System.out.println("peel done");
        return new Peel(1);
    }
}

public class Apple extends Fruit { }
-----
// corruption de Example1Fruit
public class Example1CorrompuFruit {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
} // Example1CorrompuFruit

```

// change interface Fruit
// change type de retour de peel()
// rien n'est à changer
// ERREUR : CODE CORROMPU

Utilisation de la composition

au lieu d'hériter d'une classe C , on peut :

- ① détenir une référence vers une instance de C
 - ② encapsuler les méthodes à récupérer de C dans de nouvelles méthodes
- il n'y a pas de récupération automatique de toute l'interface publique de C , elle doit être explicite \Rightarrow **délégation**
 - C devient une classe "back-end"
 - encapsulation *plus forte* : on peut ne pas changer l'**interface** de la classe englobante lorsque l'on change celle de C

Rappel : *interface = signatures des messages acceptés par instances*

Exemple revisité

```

public class Fruit {
    public int peel() {
        System.out.println("peel done");
        return 1;
    }
}

public class Apple {
    private Fruit fruit = new Fruit(); // composition

    public int peel() {
        return this.fruit.peel();
    }
}

public class Example2Fruit {
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    }
} // Example2Fruit

```

Changement de l'interface

```

public class Peel {
    private int numberOfPeels;
    public Peel(int numberOfPeels) { this.numberOfPeels = numberOfPeels; }
    public int getNumberOfPeels() { return numberOfPeels; }
}
public class Fruit {                                // change interface Fruit
    public Peel peel() {                            // change du type de retour de peel()
        System.out.println("peel done");
        return new Peel(1);
    }
}
public class Apple {
    private Fruit fruit = new Fruit();
    public int peel() {                             // il faut changer le code de peel()
        return (this.fruit.peel()).getNumberOfPeels();
    }
}
public class Example2Fruit {                       // pas de corruption de Example2Fruit
    public static void main(String[] args) {
        Apple apple = new Apple();
        int pieces = apple.peel();
    } // Example2CorrompuFruit
}

```

Composition > Héritage

- **C** plus facile de changer l'interface d'une classe "back-end" que d'une super-classe.
 ↪ permet de conserver l'**interface** de la classe "front-end" (même si chgmt code)
- **C** plus facile de changer l'interface d'une classe "front-end" que d'une sous-classe.
 ↪ il n'est pas toujours possible de changer l'interface d'une méthode héritée (ex : type de retour) ; avec l'agrégation, on a le choix.
- **C** avec agrégation, on peut retarder la création des objets agrégés avec l'héritage, sous-couche créée dès la création de l'objet
- **=** le changement de l'implémentation (pas de l'interface) est aussi facile dans les 2 cas

Héritage > composition

- **H** plus facile d'ajouter des sous-classes par héritage que d'ajouter des classes "front-end" par composition
 ↪ grâce au polymorphisme, on peut facilement passer d'une (instance de) super-classe à sa sous-classe sans changer de code pas si simple avec composition...
- **H** la délégation d'invocation de méthode de la composition a un coût, surtout si elle est systématique
- **H** lien par héritage plus facile à appréhender que lien par composition
- = le changement de l'implémentation (pas de l'interface) est aussi facile dans les 2 cas

Utiliser les interfaces

- **composition** plus de flexibilité, risque de corruption de code limité
- **héritage** plus facile d'hériter des sous-classes (grâce à l'utilisation de l'upcast)

Penser à combiner composition et interfaces

Exemple revisité (2)

```

public interface Peelable {
    public Peel peel();
}
public class Fruit {
    public int peel() {
        System.out.println("peel done");
        return 1;
    }
}
public class Apple implements Peelable {
    private Fruit fruit = new Fruit();
    public Peel peel() {
        return new Peel(this.fruit.peel());
    }
}
public class PeelProcessor() {
    public void peelSomething(Peelable something) {
        something.peel();
    }
    public static void main(String[] args) {
        PeelProcessor pp = new PeelProcessor();
        pp.peelSomething(new Apple());
    }
} // PeelProcessor

```

- le problème de la modification de la classe “back-end” est pris en charge par la composition
- le problème de l’ajout de sous-classe est pris en charge par l’interface `Peelable`

Héritage=tentation

- héritage **SSI** relation *is-a* entre les classes
exemples :
 - Apple *is-a* Fruit semble naturel
 - StudentList *is-a* ArrayList<Student> (me?) paraît plus discutable
- la relation *is-a* est elle durable/fiable ?
pour la durée l'application mais aussi la durée du code en général (API)
on peut avoir une Person *is-a* Employee mais quid si la Person devient sans emploi... ou devient un Boss...
- ne pas utiliser l'héritage juste pour "récupérer" du code, préférer la composition

The Liskov Substitution Principle (LSP)

*Les sous-classes doivent pouvoir remplacer leur classe de base
Les méthodes qui utilisent des objets d'une classe doivent pouvoir
utiliser "inconsciemment" des objets dérivés de cette classe*

On doit pouvoir Upcaster sémantiquement

Le dilemme cercle/ellipse

un *Cercle* est une *Ellipse*

Ellipse : 2 foyers

Circle : 1 seul centre

problème avec `setFoyers(Point p1, Point p2)`

ne pas utiliser l'héritage juste pour factoriser du code

héritage \implies extension/spécialisation

(ce n'est pas le cas *Cercle* pour *Ellipse*, dans ce cas utiliser la composition)