# Interfaces (suite)

Programmation Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille 1

- interaction avec un objet par envoi de messages (= appel de méthode, le plus svt)
- pour manipuler un objet **il faut et il suffit** de connaître les messages qu'il accepte

*type* = ensemble des messages acceptés par objet de ce type
$\Longrightarrow$ pour manipuler un objet, il faut connaître son "*type*"

2 notions :

- les messages acceptés (intervient à la programmation)
  $\hookrightarrow$ la *signature* de la méthode
- la réaction aux messages (traitement/comportement) (intervient à l'exécution)
  $\hookrightarrow$ le *code* de la méthode
- la compilation vérifie la "légalité" d'un envoi de message sur une référence en fonction de son type

# Classe

- En Java, pour définir une référence, il faut préciser son type :

  ```
  UnType uneReference;
  ```

  On peut alors invoquer sur la référence **tous** les envois de messages définis par le type `UnType` mais **aucun autre** !
- **Mais** la référence doit préalablement être **initialisée** par une instance **d'une classe**.
- Le traitement provoqué par l'invocation est alors défini par **la classe** de l'objet référencé.

**une classe est un type, une interface aussi**

- **mais en +** : une classe **impose** le traitement associé aux messages
- toutes les instances d'une même classe accomplissent le **même** traitement

# Question

Peut on, et comment, permettre des comportements différents pour une même manipulation ?
c-à-d

avoir des réactions différentes pour une même invocation

**ou** séparer l'envoi de messages du traitement associé

**ou** séparer la signature de la méthode du code associé

# Pourquoi ? (exemples)

- disposer d'une méthode **générique** de tri d'un tableau (ie. sans imposer la nature des éléments)
  il faut pouvoir :
  - typer les éléments du tableau
  - comparer deux éléments : le traitement associé à cette comparaison va donc dépendre de la classe des éléments
- décrire la **notion** de compteur
  un compteur est "quelque chose" :
  - qui a une valeur et qui peut passer à la valeur suivante ("incrémenter")

  il existe **différents** types de compteur
  - compteur cyclique, compteur borné, à "incrément" variable

  pour chacun de ces types de compteurs le traitement est différent, mais la manipulation est identique

- avoir un outil de manipulation d'images de **différents** formats (jpg, gif, bmp, etc.).
  Pour une image on veut pouvoir connaître :
  - sa dimension (largeur× hauteur)
  - la valeur du pixel à coordonnée donnée
  - convertir l'image vers un autre format

  les traitements vont dépendre du format de l'image (car de son codage), on veut pourtant manipuler les images d'une manière identique, et pouvoir ajouter d'autres formats d'images.
- jeu pour lequel l'ordinateur peut être adversaire, avec possibilité d'avoir **différentes** stratégies de jeu pour le programme
  Un joueur géré par un programme doit être capable de calculer son coup. Pour que tous les joueurs gérés par l'ordinateur ne jouent pas de la même manière, il faut pouvoir différencier le traitement associé à la demande de calcul du coup de jeu.

# Algorithme générique

- Les objets du tableau doivent être comparables 2 à 2
  C'est la seule condition pour le "typage" : $\Longrightarrow$ interface **`Comparable`**

```
public interface Comparable<T> {
   public int compareTo(T o);
}
public class String implements Comparable<String>, ...  { ...  }
public class TriBulle {
   public void triBulle(Comparable<?>[] aTrier) {
      Comparable tmp;
      for(int i = aTrier.length-1; i > 0; i--) {
         for(int j = 0; j < i; j++) {
            if (aTrier[j].compareTo(aTrier[j+1]) <0) {
               tmp = aTrier[j]; aTrier[j] = aTrier[j+1]; aTrier[j+1] = tmp;
            } } }
   }
   public void affiche(Comparable<?>[] t) { ...  }
}
```

```
// ...  utilisation
TriBulle tb = new TriBulle();
String[] t = new String[]{"bacd", "abcd", "aabc", "abda"};
tb.affiche(t);
tb.triBulle(t);
tb.affiche(t);
```

```
+--trace--------------
| bacd abcd aabc abda
|
| aabc abcd abda bacd
```

# Ce qu'il NE FAUT PAS écrire (1)

Avoir une seule classe `Image` et distinguer par un attribut "`type`" les différents formats d'image.

- nécessité de séparer les traitements particuliers à un format par des tests sur `type`
  ↪ pour chaque méthode "spécialisée" au format
  - `getWidth()`, `getHeight()`, `loadImage()`, `saveImage()`, etc.
- problème lors de l'ajout d'un nouveau type : chaque méthode doit être modifiée
  ↪ **difficulté d'extension** (cf. "*Open Close Principle*")
- + problème de l'attribut représentant les données de l'image : il faut choisir une représentation commune pour tous les formats.

# Ce qu'il NE FAUT PAS écrire (2)

```
public class Image {
   private String type;                          // ou avec une classe ImageType
   public Image(String type) {                   // constructeur
      this.type = type;
   ...  }
   public int getWidth() {
      if (this.type.equals("jpg")) {
         ...  // traitement pour déterminer la largeur d'une image jpg
      }
      else if (this.type.equals("gif")) {
         ...  // traitement pour déterminer la largeur d'une image gif
      }
      else if (this.type.equals("bitmap")) {
         ...  // traitement pour déterminer la largeur d'une image bitmap
      }
   }
   ...  }
public class ImageManipulator {
   ...
   public int getImageWidth(Image img) { return img.getWidth(); }
}
```

**NON !**

```
... utilisation
   ImageManipulator manipulator = new ImageManipulator();
   Image img1 = new Image("gif");   Image img2 = new Image("jpg");
   manipulator.getImageWidth(img1);                  // même manipulation
   manipulator.getImageWidth(img2);                  // pour les 2 formats
```

# Pour des comportements différents il faut des classes différentes

avoir 3 classes

`JpgImage` `GifImage` `BitmapImage`

avec chacune leur “version” des méthodes

`getWidth()`, `getHeight()`, `loadImage()`, `saveImage()`, etc.

chaque classe d’image définit la méthode `public int getWidth()`

- que devient la méthode `getImageWidth` de `ImageManipulator` ?

# Ce qu'il NE FAUT PAS écrire NON PLUS

```
public class ImageManipulator {
   ...
   public int getImageWidth(Object img) {
      if (img instanceof JpgImage) {
         return ((JpgImage) img).getWidth();
      }
      else if (img instanceof GifImage) {
         return ((GifImage) img).getWidth();
      }
      else if (img instanceof BitmapImage) {
         return ((BitmapImage) img).getWidth();
      }
      else throw new NotAnImageException();
   }
   ...
}
```

**NON !**

```
... utilisation
ImageManipulator manipulator = new ImageManipulator();
JpgImage img1 = new JpgImage(...);
GifImage img2 = new GifImage(...);
manipulator.getImageWidth(img1);                    // même manipulation
manipulator.getImageWidth(img2);                    // pour les 2 formats
```

# Constat

- difficulté de l'ajout d'une nouvelle classe d'images (OCP ?) :
  - modification de chaque méthode de `ImageManipulator`
- l'argument de `getImageWidth` est de type `Object`
  ↪ pas de détection à la compilation si l'argument de `getImageWidth` n'est pas une instance d'une "classe image"

```
ImageManipulator manipulator = new ImageManipulator();
// accepté à la compilation:
manipulator.getImageWidth(new String("timoleon"));
```

$\Longrightarrow$ on perd le typage

Il faut **mixer** les deux approches :

- Il faut un type `Image` **commun**
- Il faut des classes **différentes** pour chaque format d'image

# La solution

Pour obtenir des traitements différents avec une même invocation de méthode, il faut avoir des objets :

- de **même** type
- de classes **différentes**

La solution ?

les **INTERFACES**

- fixent les messages acceptés/autorisés
- le comportement doit être implémenté **séparément** par des classes

# Polymorphisme

Un objet est toujours instance d'une classe

- Si une classe implémente une interface une instance de cette classe peut être vue comme du type de l'interface et manipulée comme telle

**Polymorphisme**

un objet est du type :

- de sa classe
- des interfaces implémentées par sa classe

- On peut déclarer une référence comme étant du type d'une interface
  $\Rightarrow$ on n'autorise sur cette référence que les envois de messages définis par l'interface

**MAIS**

- on initialise la référence par un objet qui est **instance d'une classe**
- cette classe doit implémenter l'interface.
- on **ne peut pas** invoquer sur la référence les méthodes définies par la classe mais **pas** par l'interface.

La référence ne donne accès qu'à la partie de l'objet **restreinte** à ce qui est défini par l'interface.

# Ce qu'il FAUT écrire

```
public interface Image {
   public int getWidth(); ...
}
public class JpgImage implements Image {
   public int getWidth() {
      ...  // traitement pour déterminer la largeur d'une image jpg
   }
}
public class GifImage implements Image {
   public int getWidth() {
      ...  // traitement pour déterminer la largeur d'une image gif
   }
}

public class ImageManipulator {
   public int getImageWidth(Image img) { return img.getWidth(); }
}
```

**OUI !**

```
... utilisation
ImageManipulator manipulator = new ImageManipulator();
Image img1 = new JpgImage();
Image img2 = new GifImage();
manipulator.getImageWidth(img1);                    // même manipulation
manipulator.getImageWidth(img2);                    // pour les 2 formats
```

# Ajout d'une nouvelle classe d'images ?

Il suffit de définir une classe implémentant l'interface `Image`

```
public class BitmapImage implements Image {
   public int getWidth() {
      ...  // traitement pour déterminer la largeur d'une image bitmap
   }
}
```

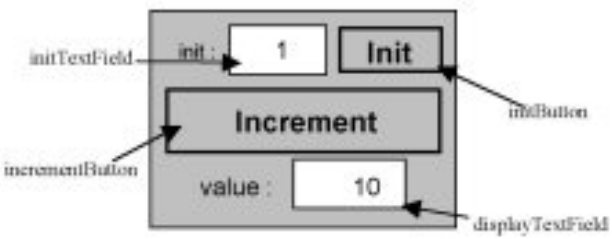
```
...  utilisation
ImageManipulator manipulator = new ImageManipulator();
manipulator.getImageWidth(new BitmapImage());     // même manipulation
```

"*Open Close Principle*" respecté
grâce à l'**abstraction** de la notion "`Image`"

# Compteur

- on souhaite disposer d'une classe pour gérer l'interface graphique de compteurs à valeurs entières

```
public class CounterGraphiqueIHM {    // très approximatif !!!
   private Counter counter;
   ...
   public CounterGraphiqueIHM(Counter counter) {
      ...  // initialisation de la partie graphique
      this.counter = counter;
   }
   public void initButtonAction() {
      int value = Integer.parseInt(initTextField.getText());
      counter.initValue(value);
      displayCounter();
   }
   public void incrementButtonAction() {
      counter.increment();
      displayCounter();
   }
   public void displayCounter() {
      displayTextField.setText(""+counter.getCurrentValue());
   }
}
```




- Il faut avoir des compteurs de **même** type (ie. présentant une interface commune) et ayant des comportements **différents**

# Abstraction de la notion de Compteur

```
public interface Counter {
   public int getCurrentValue();
   public void increment();
   public void initValue(int init);
}


public class SimpleCounter implements Counter {
   private int value;
   public SimpleCounter(int value) { this.value = value; }
   public int getCurrentValue() { return this.value:  }
   public void increment() { this.value++; }
   public void initValue(int init) { this.value = init; }
}

public class ModularCounter implements Counter {
   private int value;
   private int modulo;
   public SimpleCounter(int value, int modulo) {
      this.value = value; this.modulo = modulo;
   }
   public int getCurrentValue() { return this.value:  }
   public void increment() { this.value = (this.value+1) % modulo; }
   public void initValue(int init) { this.value = init; }
}
```

- l'ajout d'une nouvelle classe de compteur est immédiat :

```
public class AnotherCounter implements Counter {
   private int value;
   public SimpleCounter(int value) { this.value = value; }
   public int getCurrentValue() { return this.value:  }
   public void increment() { this.value = 2*this.value + 1; }
   public void initValue(int init) { this.value = init; }
}


// ...  utilisation

Counter simpleCounter = new SimpleCounter(0);              // upcast
Counter modularCounter = new ModularCounter(0,7);          // vers
Counter anotherCounter = new AnotherCounter(0);            // Counter


new CounterGraphiqueIHM(simpleCounter);
new CounterGraphiqueIHM(modularCounter);
new CounterGraphiqueIHM(anotherCounter);
```

# Autre version : abstraction de la notion d'incrément

- Répétition du code des méthodes `getCurrentValue` et `initValue`
  En fait l'abstraction se situe au niveau de la manière d'incrémenter...

```
public interface IncrementFunction {
   public int increment(int value);
}

public class SimpleIncrement implements IncrementFunction {
   public int increment(int value) { return value++; }
}

public class ModularIncrement implements IncrementFunction {
   private int modulo;
   public ModularIncrement (int modulo) { this.modulo = modulo; }
   public int increment(int value) { return (value+1) % modulo; }
}

public class AnotherIncrement implements IncrementFunction {
   public int increment(int value) { return 2*value+1; }
}
```

## Autre version : abstraction de la notion d'incrément

```
public class Counter {
   private int value;
   private IncrementFunction incrementF;
   public SimpleCounter(int value, IncrementFunction incrementF) {
      this.value = value;
      this.incrementF = incrementF;
   }
   public int getCurrentValue() { return this.value:  }
   public void increment() { this.value = incrementF.increment(this.value); }
   public void initValue(int init) { this.value = init; }
}


// ...  utilisation

Counter simpleCounter = new Counter(0, new SimpleIncrement());      // upcast
Counter modularCounter = new Counter(0, new ModularIncrement(7));   // vers
Counter anotherCounter = new Counter(0, new AnotherIncrement());    //IncrementFunction

new CounterGraphiqueIHM(simpleCounter);
new CounterGraphiqueIHM(modularCounter);
new CounterGraphiqueIHM(anotherCounter);
```

# Des joueurs avec des stratégies différentes

```
public class Jeu {
   Joueur joueur1;
   Joueur joueur2;
   public Jeu(Joueur joueur1, Joueur joueur2) {
      this.joueur1 = joueur1; this.joueur2 = joueur2;
   }
   public void unTourDeJeu {
      Coup coupJoueur1 = this.joueur1.joue();
      Coup coupJoueur2 = this.joueur2.joue();
      ...  // comparaison des coups et résolution du tour de jeu
   }
```

- Faire en sorte que les joueurs puissent jouer de manières différentes :
  $\Longrightarrow$ traitements différents
- l'**abstraction** se situe au niveau de la manière de jouer :
  **notion de stratégie de jeu**
- Les joueurs sont tous instances d'une même classe
- Ils sont **paramétrés** par leur stratégie de jeu qui permet de différencier leur comportement.

## Des joueurs avec des stratégies différentes

```
public interface Strategie {
   public Coup calculDuCoupAJouer(...);     // classe Coup définie "ailleurs"
}

public class Joueur {
   private Strategie maStrategie;
   public Joueur(Strategie strategie, ....)  {
      this.maStrategie = strategie; ...
   }
   public Coup joue() {
      return this.maStrategie.calculDuCoupAJouer(...);
   }
}

public class StrategieAleatoire implements Strategie {
   public Coup calculDuCoupAJouer(...)  { ...  }; // choix aléatoire du coup
}
public class StrategieImpl implements Strategie {
   public Coup calculDuCoupAJouer(...)  { ...  }; // autre manière de jouer
}

// ...  utilisation
   Joueur joueur1 = new Joueur(new StrategieAleatoire());
   Joueur joueur2 = new Joueur(new StrategieImpl());
   new Jeu(joueur1, joueur2).unTourDeJeu();
```

# Interface = Abstraction

Les interfaces sont des **types**

- **fixent** des signatures des méthodes **sans imposer** le comportement associé,
- permettent une vision **polymorphe** sur les objets et facilitent la **généricité** (notion de "*template*"),
- permettent d'**offrir** aux autres un cadre de programmation
  ↪ concepteur de *framework*
- permettent de **réutiliser** des classes et de les **adapter** à un contexte
  ↪ utilisateur de *framework*
- facilitent **l'extension** d'un programme (l'ajout de comportements) **sans modification** de l'existant.

# Open Close Principle

## Open Close Principle

"Un module doit être ouvert aux extensions mais fermé aux modifications"

- À l'extrême toujours commencer par définir des interfaces et seulement ensuite les classes les implémentant
- "*manipuler des abstractions et concrétiser le plus tard possible*"

# Faire varier le comportement

1. identifier l'abstraction à manipuler,
2. définir une interface caractérisant cette abstraction, c-à-d définir les signatures des méthodes qui y sont liées,
3. effectuer les manipulations de l'abstraction sur des références ayant pour type cette interface,
4. concrétiser cette interface par différentes classes définissant les différents comportements souhaités,
5. utiliser le polymorphisme pour initialiser les références du type de l'interface par des instances des classes l'implémentant

L'interface peut intervenir au niveau d'un attribut utilisé plutôt que directement sur la classe dont on veut faire varier le comportement (cf. `IncrementFunction` et `Strategie`).