# conception et prog objet avancés

## DEV.3.4– CPOO

Florent Madelaine

IUT de Sénart Fontainebleau
Département Informatique

Année 2022-2023
Cours 3

# Les notions pour l'évaluation

1. Usage : DCU, synopsis.
2. Structure : DC sans les méthodes, DO.
3. Comportement : DC avec méthodes, Diagramme de Séquence.
4. Abstraction : DC avec classes abstraites et interfaces, API, couplage faible.

# Couplage faible

## Intérêt

Lorsque un projet devient complexe avec beaucoup de classes qui dépendent directement les unes des autres, on ne peut plus rien changer facilement.
Dans le monde réel, un logiciel connaît des évolutions dans le temps avec parfois plusieurs équipes voir plusieurs entreprises qui y travaille, si les choses sont fortement couplées on ne peut plus rien changer.

## Solution

Il faut pouvoir avoir une approche modulaire en limitant les interactions et en rendant indépendant les couches.
L'abstraction est un mécanisme naturel en OO qui permet de définir exactement la manière dont les différentes parties communiqueront (c'est ce qu'on appelle une API).

# Révisions Classes abstraites

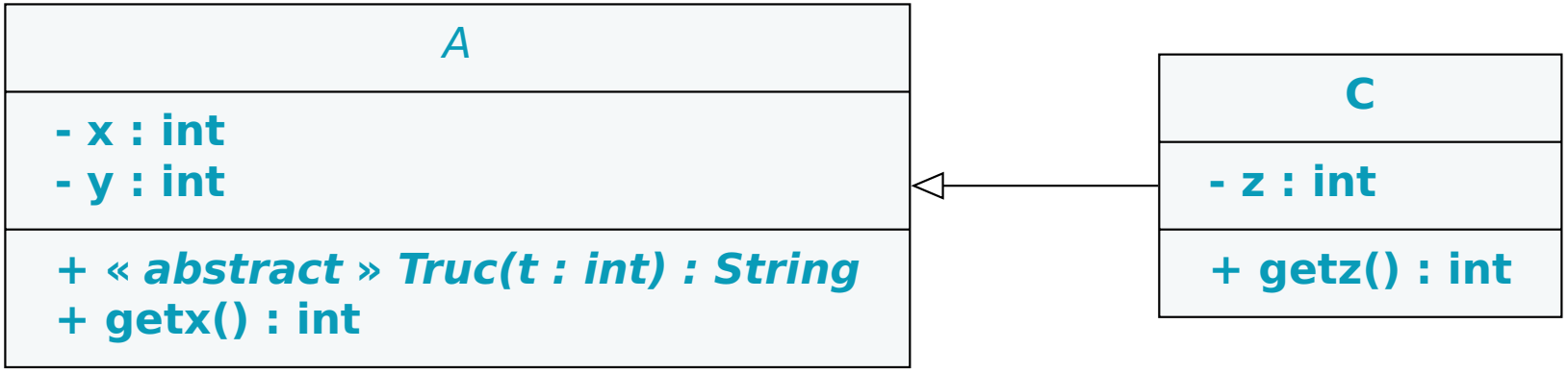
## Comment ça marche ?

- Une classe abstraite ne peut pas être instanciée directement.
- Il faut instancier une classe concrète qui en hérite et donc implémente toutes les méthodes de cette classe abstraite pour avoir un objet qu'on peut considérer comme de ce type abstrait par héritage.
- L'intérêt c'est de proposer un traitement homogène de tous les objets de ce type abstrait tout en interdisant d'instancier directement un objet de ce type.
- Notez qu'une classe abstraite peut tout à fait avoir des méthodes qui ne sont pas abstraites (par exemple les get/set).

# Révisions Classes abstraites

## Notation UML

- Le nom de la classe est en italique.
- Les méthodes abstraites sont en italique.
- On ajoute aussi parfois le stéréotype «abstract» en plus de l'italique (nécessaire quand vous rendez un diagramme fait à la main, car ce n'est pas toujours clair sinon que la classe est abstraite).
- L'héritage est noté comme d'habitude.



# Où puis-je trouver des exemples ?

Mhh je voudrais des exemples.
Je voudrais des exemples en java.

J'ai les sources et je vais voir dedans ! Commençons par regarder la doc de java.util à tout hasard, au cas où ça pourrait servir, être utile quoi.

# Où puis-je trouver des exemples ?

Mhh je voudrais des exemples.
Je voudrais des exemples en java.

J'ai les sources et je vais voir dedans ! Commençons par regarder la doc de java.util à tout hasard, au cas où ça pourrait servir, être utile quoi.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/package-tree.html

```
Class Hierarchy
  java.lang.Object
     java.util.AbstractCollection<E>
       java.util.AbstractList<E>
         java.util.AbstractSequentialList<E>
           java.util.LinkedList<E>
        java.util.ArrayList<E>
        java.util.Vector<E>
          java.util.Stack<E>
      java.util.AbstractQueue<E>
        java.util.PriorityQueue<E>
      java.util.AbstractSet<E>
        java.util.EnumSet<E>
        java.util.HashSet<E>
          java.util.LinkedHashSet<E>
        java.util.TreeSet<E>
      java.util.ArrayDeque<E>
    java.util.AbstractMap<K,V>
      java.util.EnumMap<K,V>
      java.util.HashMap<K,V>
        java.util.LinkedHashMap<K,V>
      java.util.IdentityHashMap<K,V>
      java.util.TreeMap<K,V>
      java.util.WeakHashMap<K,V>
...
```

**Exo**

Faite un dessin rapide des classes héritant de `AbstractCollection` (NB : si il y a Abstract dans le nom c'est abstrait, sinon c'est concret).

# Extrait source AbstractCollection

```
public abstract class AbstractCollection<E> implements Collection<E>
{
  // ... snip ...
  // Query Operations
  /**
   * Returns an iterator over the elements contained in this collection.
   * @return an iterator over the elements contained in this collection
   */
  public abstract Iterator<E> iterator();

  public abstract int size();
  /**
   * {@inheritDoc}
   * <p>This implementation returns <tt>size() == 0</tt>.
   */
  public boolean isEmpty() {
      return size() == 0;
  }

  /**
   * {@inheritDoc}
   *
   * <p>This implementation iterates over the elements in the collection,
   * checking each element in turn for equality with the specified element.
   *
   * @throws ClassCastException   {@inheritDoc}
   * @throws NullPointerException {@inheritDoc}
   */
  public boolean contains(Object o) {
      Iterator<E> it = iterator();
      if (o==null) {
          while (it.hasNext())
              if (it.next()==null)
                  return true;
      } else {
          while (it.hasNext())
              if (o.equals(it.next()))
                  return true;
      }
      return false;
    }
```

# Suite extrait source AbstractCollection

```
     // ... snip ...
    /**
     * Returns a string representation of this collection.  The string
     * representation consists of a list of the collection's elements in the
     * order they are returned by its iterator, enclosed in square brackets
     * (<tt>"[]"</tt>).  Adjacent elements are separated by the characters
     * <tt>", "</tt> (comma and space).  Elements are converted to strings as
     * by {@link String#valueOf(Object)}.
     *
     * @return a string representation of this collection
     */
    public String toString() {
        Iterator<E> it = iterator();
        if (! it.hasNext())
            return "[]";

        StringBuilder sb = new StringBuilder();
        sb.append('[');
        for (;;) {
            E e = it.next();
            sb.append(e == this ? "(this Collection)" : e);
            if (! it.hasNext())
                return sb.append(']').toString();
            sb.append(',').append(' ');
        }
    }

}
```

# Extrait source AbstractList

```
public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {
 // ... snip ...
/**
 * {@inheritDoc}
 *
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
abstract public E get(int index);

 // ... snip ...
// Iterators

/**
 * Returns an iterator over the elements in this list in proper sequence.
 *
 * <p>This implementation returns a straightforward implementation of the
 * iterator interface, relying on the backing list's {@code size()},
 * {@code get(int)}, and {@code remove(int)} methods.
 *
 * <p>Note that the iterator returned by this method will throw an
 * {@link UnsupportedOperationException} in response to its
 * {@code remove} method unless the list's {@code remove(int)} method is
 * overridden.
 *
 * <p>This implementation can be made to throw runtime exceptions in the
 * face of concurrent modification, as described in the specification
 * for the (protected) {@link #modCount} field.
 *
 * @return an iterator over the elements in this list in proper sequence
 */
public Iterator<E> iterator() {
    return new Itr();
  }
  // ... snip ...
```

# Extrait source ArrayList

```java
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * Default initial capacity.
     */
    private static final int DEFAULT_CAPACITY = 10;

    // ... snip ...

    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer. Any
     * empty ArrayList with elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA
     * will be expanded to DEFAULT_CAPACITY when the first element is added.
     */
    transient Object[] elementData; // non-private to simplify nested class access

    /**
     * The size of the ArrayList (the number of elements it contains).
     *
     * @serial
     */
    private int size;

    // ... snip ...


    /**
     * Returns the number of elements in this list.
     *
     * @return the number of elements in this list
     */
    public int size() {
        return size;
      }

      // ... snip ...
```

# Révisions interfaces

## Comment ça marche ?

- Une interface est essentiellement une classe abstraite sans attribut.
- Notez qu'une interface en java peut avoir des méthodes qui ne sont pas abstraites (*default method*).
- Une classe implémente une interface en offrant toutes les méthodes de l'interface.
- L'interface est donc un contrat que la classe doit remplir en terme de comportement.

## Différence notable avec les classes abstraites

On peut implémenter plusieurs interfaces ! Solution de l'héritage multiple pas pratique.

# Révisions interfaces

## Comment ça marche ?

- Une interface est essentiellement une classe abstraite sans attribut.
- Notez qu'une interface en java peut avoir des méthodes qui ne sont pas abstraites (*default method*).
- Une classe implémente une interface en offrant toutes les méthodes de l'interface.
- L'interface est donc un contrat que la classe doit remplir en terme de comportement.

## Différence notable avec les classes abstraites

On peut implémenter plusieurs interfaces ! Solution de l'héritage multiple pas pratique.

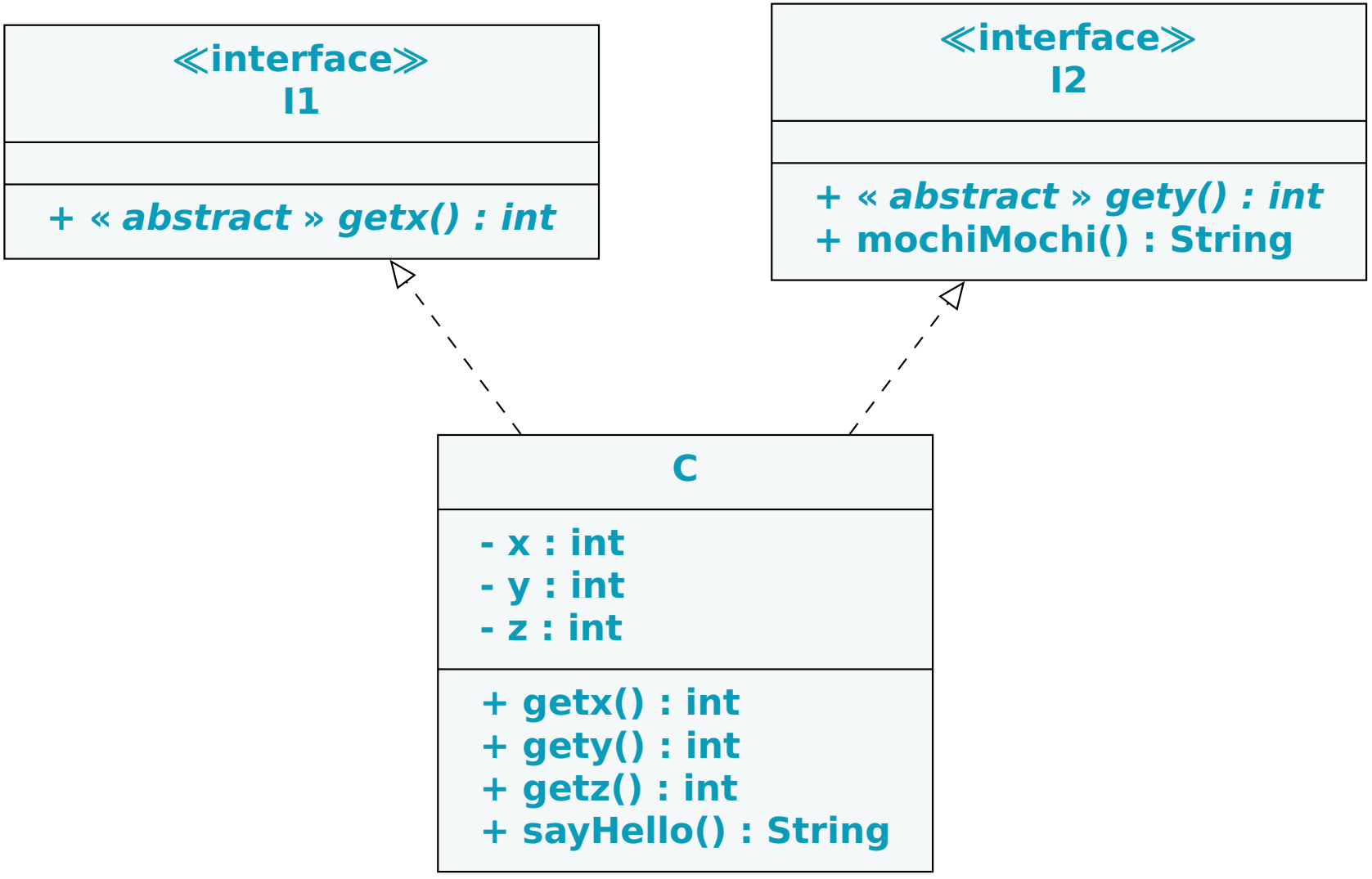
## Notation UML

- On ajoute le stéréotype «`interface`»
- Comme il y a maintenant les *default method* je suggère de les mettre en fonte normale et
- de mettre en italique les autres méthodes de l'interface avec le stéréotype « abstract ».
- La flèche de l'implémentation ressemble à celle de l'héritage mais en pointillés.

## Notation UML

- On ajoute le stéréotype «`interface`»
- Comme il y a maintenant les *default method* je suggère de les mettre en fonte normale et
- de mettre en italique les autres méthodes de l'interface avec le stéréotype « abstract ».
- La flèche de l'implémentation ressemble à celle de l'héritage mais en pointillés.

# Exemple Notation



# Intérêt des default Method

Si l'interface change (il ne faut pas c'est mal mais entre deux versions majeures on a le droit et puis des nouveaux besoins peuvent arriver), alors le vieux code n'est plus compatible en théorie.

Ceci permet de pouvoir espérer garder une rétro-compatibilité du vieux code avec la nouvelle interface.

Notez que si le vieux code contient une méthode du même nom que la nouvelle default method, on a quant même perdu.

https://stackoverflow.com/questions/31188231/do-java-8-default-methods-break-source-compatibility

# Exemple : java util

**https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/package-tree.html**

```
util.AbstractCollection<E> (implements util.Collection<E>)
  util.AbstractList<E> (implements util.List<E>)
    util.AbstractSequentialList<E>
      util.LinkedList<E>
     (implements lang.Cloneable, util.Deque<E>, util.List<E>, io.Serializable)
   util.ArrayList<E>
   (implements lang.Cloneable, util.List<E>, util.RandomAccess, io.Serializable)
   util.Vector<E>
   (implements lang.Cloneable, util.List<E>, util.RandomAccess, io.Serializable)
     util.Stack<E>
  util.AbstractQueue<E> (implements util.Queue<E>)
    util.PriorityQueue<E> (implements io.Serializable)
  util.AbstractSet<E> (implements util.Set<E>)
    util.EnumSet<E> (implements lang.Cloneable, io.Serializable)
    util.HashSet<E>
    (implements lang.Cloneable, io.Serializable, util.Set<E>)
      util.LinkedHashSet<E>
      (implements lang.Cloneable, io.Serializable, util.Set<E>)
    util.TreeSet<E>
    (implements lang.Cloneable, util.NavigableSet<E>, io.Serializable)
  util.ArrayDeque<E> (implements lang.Cloneable,util.Deque<E>, io.Serializable)
```

**Exo**

Faite un dessin rapide des interfaces implémentées par les classes héritant de `AbstractCollection`.

# Autres liens entre classes

L'héritage et les association introduisent un couplage fort entre les classes.

Il existe d'autres formes de couplage fort, par exemple quand une classe va créer une instance d'une autre classe, ou encore recevoir puis transmettre une telle instance ou encore utiliser une méthode statique d'une autre classe.

Dans tous ces cas on peut noter la dépendance sur le diagramme de classe avec une flèche avec des pointillés et un stéréotype expliquant de quoi il s'agit.

# Comment faire pour réduire sa dépendance ?

## Solution

Il faut dépendre de choses abstraites.

## Conséquences

- Les diagrammes deviennent plus compliqués.
- Mais on gagne en flexibilité.
- On sait ce qui ne peut pas changer sans risquer de casser quelque chose (les interfaces).
- Ici c'est la signature des méthodes et le comportement préconisé de chaque méthode qui ne peut pas changer.
- On peut changer l'implémentation sans soucis.

# Exemple

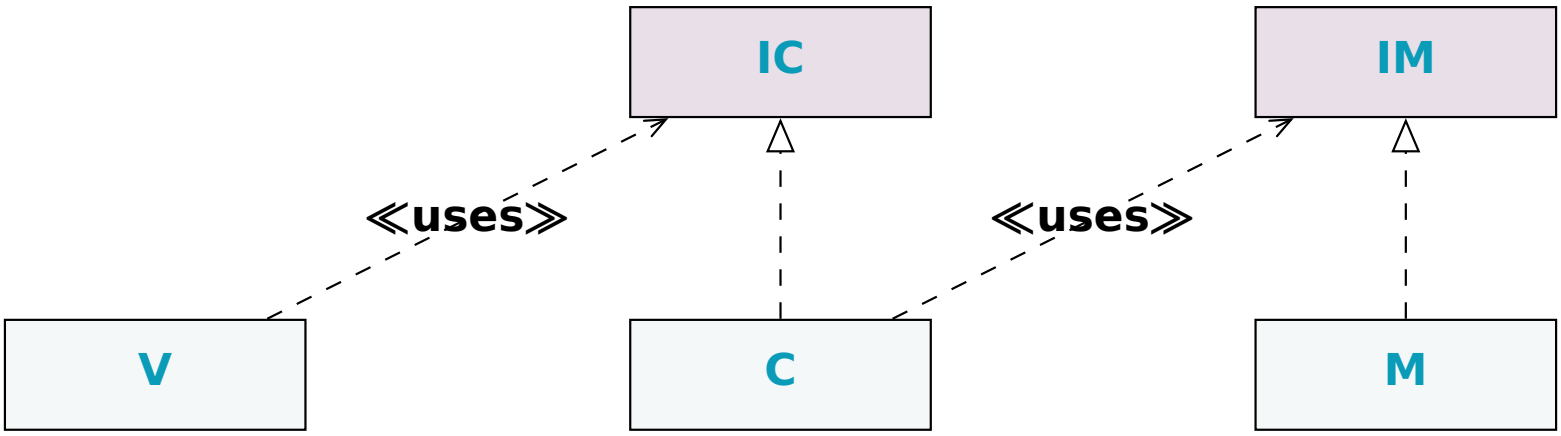
Imaginons trois couches réalisées par des personnes différentes. La présentation (V), la partie métier (C) et la partie DAO (M).


Si l'une d'entre elle change, alors je dois répercuter des changements dans les autres classes.

# Découpler

## Idée

Il faut découpler les classes en passant par des interfaces qui établissent le contrat de communication



# C’est du costaud, c’est SOLID

## Bilan

La solution proposée est connue sous le nom de *dependency inversion principle* (le D de SOLID). C’est-à-dire que :

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d’abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Avec une telle conception, nous sommes fermés à la modification mais ouvert à l’extension.
C’est le *Open Closed principle* (le O de SOLID dont vous avez entendu parlé probablement en APL avec Luc).

https://en.wikipedia.org/wiki/SOLID

# Exo final

**Exo**

Faite un dessin rapide de l'API proposée pour le projet IHM.