Interfaces

Modélisation et Programmation par objets 1 – cours 7

Une interface est au sens commun (définition du dictionnaire le Robert) une limite commune à deux systèmes, deux ensembles, deux appareils. En informatique, on rencontre souvent le terme interface quand on définit les interactions d'un système avec ses utilisateurs : on parle souvent d'interface utilisateur, pouvant être graphique ou textuelle (le terme plus à la mode est CLI, Command Line Interface). Les interfaces que nous abordons ici ne sont pas des interfaces utilisateurs, mais plutôt des interfaces entre deux systèmes informatiques.

1 Interfaces, premiers éléments

Une interface est par définition une zone de contact. Les interfaces qui nous intéressent ici sont celles qui se placent comme frontière entre des éléments logiciels.

Comme nous nous plaçons ici dans le cadre de la modélisation et programmation orientée Objets, une interface est comme une façade pour une classe, décrivant ce que la classe peut offrir comme services à un programme.

Les interfaces contiennent majoritairement des déclarations de méthodes publiques, qui décrivent donc des services offerts (et quelques autres éléments que nous verrons à la section 2).

Les interfaces permettent d'améliorer le code pour plusieurs raisons sur lesquelles nous reviendront par la suite :

— on décrit des types de manière plus abstraite qu'avec les classes et par conséquent ces types sont plus réutilisables ;
— c'est une technique pour masquer l'implémentation puisqu'on découple la partie publique d'un type de son implémentation ;
— on peut écrire du code plus générique (plus général), au sens où il est décrit sur ces types plus abstraits ;
— les relations de spécialisation entre les interfaces (d'une part) et entre les classes et les interfaces (d'autre part) relèvent de la spécialisation multiple, ce qui facilite l'organisation des types d'un programme.

1.1 Première interface en UML

Une interface se présente presque comme une classe, mais annotée par $<< interface >>$, comme illustré à la figure 1.

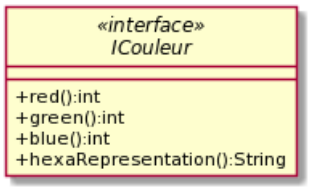

Figure 1 – Une interface ICouleur

Une interface UML contient principalement des opérations, comme illustré ici : l'interface couleur a ici 4 méthodes : trois pour obtenir les valeurs de rouge, de vert et de bleu, et une pour obtenir le code hexadécimal de la couleur. Ce sont les 4 opérations que l'on attend ici d'une couleur.

1.2 Première interface en Java

Une interface Java contient principalement des déclarations de méthodes. Une première interface ICouleur est donnée au code 1.

Programme 1 : *Une interface ICouleur*

```
public interface ICouleur {
        public abstract int red();
        public abstract int green();
        public abstract int blue();
        public abstract String hexaRepresentation();
}
```

Les méthodes d'une interface sont par défaut abstraites (`abstract`) et publiques (`public`). Ces mots-clefs sont d'ailleurs facultatifs, de sorte que l'interface présentée au code 1 peut être également être écrite comme au code 2.

Programme 2 : *Une interface ICouleur en omettant tous les mots-clefs facultatifs*

```
public interface ICouleur {
        int red();
        int green();
        int blue();
        String hexaRepresentation();
}
```

À première vue, les interfaces se présentent donc comme des classes abstraites ne contenant que des méthodes abstraites.

1.3 Réalisation, implémentation d'interface

Les interfaces sont globalement des éléments abstraits, dont le but est de contractualiser un ensemble de services rendus. Une interface n'est donc pas directement instanciable. Elle ne contient d'ailleurs pas de constructeur.

Une interface doit être réalisée, ou implémentée, par une classe. C'est alors que l'interface se pose en façade de la classe, en exposant son "contrat" fourni.

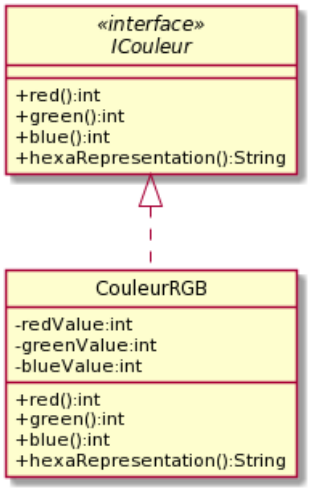

FIGURE 2 – Une interface ICouleur et une implémentation RGB

La relation d'implémentation (ou réalisation) se note avec une flèche pointillée terminée par une tête triangulaire creuse, comme illustré à la figure 2.

La classe CouleurRGB implémente l'interface ICouleur : elle a l'obligation (car elle est concrète) d'implémenter toutes les méthodes déclarées dans l'interface. La classe Couleur RGB choisit de coder les couleurs avec le système RGB (une valeur de rouge, une valeur de vert, une valeur de bleu, toutes entre 0 et 255). La classe CouleurRGB contient donc des éléments d'implémentation, alors que l'interface ICouleur n'en contient pas. On peut d'ailleurs envisager qu'une autre classe implémente ICouleur, en faisant des choix d'implémentation différents. Par exemple, on peut coder les couleurs avec le système HSL (Hue pour teinte donnée en général en degrés, S pour Saturation et L pour Luminosity, données en général en pourcentage), c'est ce qui est proposé dans la classe CouleurHSL (voir figure 3). C'est un choix d'implémentation qui est ici un peu tortueux au regard de l'interface ICouleur, mais qui peut se justifier si d'autres opérations étaient ajoutées.

En Java, une implémentation d'interface utilise le mot-clef `implements` pour déclarer l'interface implémentée, comme dans l'exemple du code 3

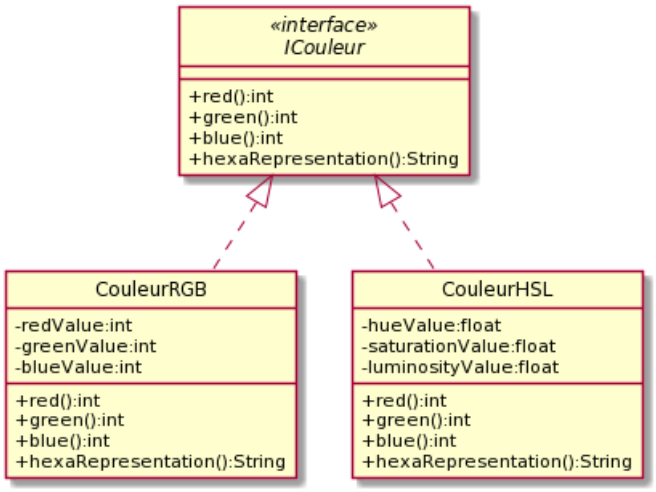

FIGURE 3 – Une interface ICouleur et deux implémentations

Programme 3 : *Une classe Couleur RGB qui implémente l'interface ICouleur*

```
public class CouleurRGB implements ICouleur{
  private int redValue;
  private int greenValue;
  private int blueValue;

  public CouleurRGB(int redValue, int greenValue, int blueValue) {
    this.redValue = redValue;
    this.greenValue = greenValue;
    this.blueValue = blueValue;
  }

  public int red() {
    return redValue;
  }

  public int green() {
    return greenValue;
  }

  public int blue() {
    return blueValue;
  }

  public String hexaRepresentation() {
    String red=Integer.toHexString(red()).length()==1?"0"+Integer.toHexString(red()):
    ↪  Integer.toHexString(red());
    String green=Integer.toHexString(green()).length()==1?"0"+Integer.toHexString(green()):
    ↪  Integer.toHexString(green());
    String blue=Integer.toHexString(blue()).length()==1?"0"+Integer.toHexString(blue()):
    ↪  Integer.toHexString(blue());
    return "#"+red+green+blue;
  }
}
```

Du point de vue des types, introduire une interface définit un nouveau type. Un élément typé par une interface peut référencer un objet dont le type est une implémentation de l'interface, ainsi qu'illustré dans l'exemple suivant (code 4.

Programme 4 : *Utilisation de l'interface ICouleur et de ses implémentations*

```
ICouleur blue= new CouleurRGB(0,0,255);
ICouleur couleur=new CouleurHSL(200f,90f,8f);
```

2 Les interfaces de Java, éléments complémentaires

2.1 Peut-on mettre des attributs dans une interface ?

2.1.1 Attributs "de classes" (statiques)

En UML comme en Java, on peut trouver dans une interface des déclarations d'attributs "de classe" (attributs statiques) constants et publiques. Ces attributs servent à déclarer des constantes.

Ainsi par exemple, on peut déclarer dans l'interface ICouleur une constante représentant le préfixe "#" à utiliser en tête du code hexadécimal d'une couleur. Cela est illustré dans la figure 4 et au code 5. Ce préfixe pourrait alors être utilisé par la méthode hexaRepresentation des implémentations de l'interface.

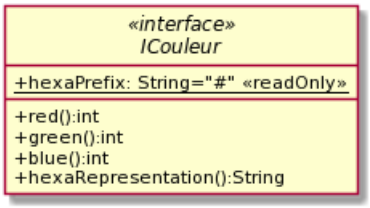

FIGURE 4 – Une interface ICouleur avec une constante

Programme 5 : *Une interface ICouleur avec une constante*

```
public interface ICouleur {
        public static final String hexaPrefix="#";
        public abstract int red();
        public abstract int green();
        public abstract int blue();
        public abstract String hexaRepresentation();
}
```

Il est à noter que pour déclarer en Java de telles constantes, les mot-clefs `static`, `public`, `final` sont facultatifs. Ainsi, on aurait pu juste écrire : `String hexaPrefix="#";`

2.1.2 Attributs d'instances, méfiance

En UML, il est possible de déclarer des attributs d'instance dans une interface. Toutefois, ceux-ci ne sont pas des éléments d'implémentation, ils servent à spécifier quels éléments devront être disponibles dans les implémentations (éventuellement via des méthodes). Nous recommandons ici de ne pas utiliser d'attributs d'instances dans les interfaces UML.

En Java, il n'est pas possible de déclarer des attributs d'instance dans une interface. Attention, du fait que le mot-clef `static` peut être omis, on peut facilement avoir l'impression d'avoir déclaré un attribut d'instance dans une interface ... Ce n'est qu'une impression, ce sera bien un attribut de classe.

2.2 Peut-on définir le corps des méthodes dans les interfaces Java ?

Avant Java 8, la réponse était non. Mais depuis Java 8, on peut dans une certaine mesure, avec les méthodes dites `default`.

On peut définir le corps d'une méthode dans une interface Java >=8 à condition que celui-ci n'ait pas besoin des détails d'implémentation (qui ne sont pas dans l'interface), c'est-à-dire n'utilise que : des constantes et des méthodes déclarées dans l'interface. Par exemple, la méthode hexaRepresentation() pourrait avoir son corps défini directement dans l'interface, avec l'usage du mot-clef `default` (et bien sûr sans le mot-clef `abstract`). Ceci est illustré au code 6.

Programme 6 : *Une interface ICouleur avec une méthode par défaut*

```
public interface ICouleur {
  public static final String hexaPrefix="#";
  public abstract int red() ;
```

```
  public abstract int green();
  public abstract int blue();
  public default  String hexaRepresentation() {
   String red=Integer.toHexString(red()).length()==1?
    ↪  "0"+Integer.toHexString(red()):Integer.toHexString(red());
   String green=Integer.toHexString(green()).length()==1?
    ↪  "0"+Integer.toHexString(green()):Integer.toHexString(green());
   String blue=Integer.toHexString(blue()).length()==1?
    ↪  "0"+Integer.toHexString(blue()):Integer.toHexString(blue());
        return hexaPrefix+red+green+blue;
  }
}
```

Notons que, galvanisé par un désir de placer du code dans des interfaces, un programmeur imprudent pourrait également définir les méthodes blue, red et green par un calcul à partir de la représentation hexadécimale donnée par la méthode hexaRepresentation. Cela crée un cycle d'appel. Que ce soit avec des méthodes par défaut ou avec des méthodes classiques dans des classes, ce cycle n'est pas détecté à la compilation mais provoque bien sûr un débordement de pile à l'exécution.

Comme le suggère le mot-clef **default**, il s'agit d'une implémentation "par défaut", qui sera disponible pour les classes d'implémentation, mais qui peut aussi être redéfinie (spécialisée ou masquée).

Rajoutons par exemple une méthode isGrey() dans l'interface. Son code peut être donné à partir des méthodes de l'interface. Par contre, pour les couleurs basées HSL, il est dommage de passer par les codes RGB pour déterminer si une couleur est un gris puisqu'il suffit de vérifier que la saturation est nulle. Cela est illustré à la figure 7.

Programme 7 : *Redéfinir (ou pas) une méthode default*

```
public interface ICouleur {
        public static final String hexaPrefix="#";
        public abstract int red() ;
        public abstract int green();
        public abstract int blue();
        public default  String hexaRepresentation() {...}
        public default boolean isGrey() {
                return red()==blue()&&blue()==green();
        }
}
public class CouleurRGB implements ICouleur{
        private int redValue;
        private int greenValue;
        private int blueValue;
        public CouleurRGB(int redValue, int greenValue, int blueValue) {...}
        public int red() {return redValue;}
        public int green() {return greenValue;}
        public int blue() {return blueValue;}
}
public class CouleurHSL implements ICouleur {
        private float hue;//teinte
        private float saturation;//saturation
        private float luminosity;// luminosité
        public CouleurHSL(float hue, float saturation, float luminosity) {
                this.hue = hue;
                this.saturation = saturation;
                this.luminosity = luminosity;
        }
        public int red() {return toRGB()[0];}
        public int green() {return toRGB()[1];        }
        public int blue() {        return toRGB()[2];        }
        private int[] toRGB() {// code compliqué !}
        public boolean isGrey() {return saturation==0;}
}
```

Les méthodes par défaut peuvent être utilisées de manière transparente en y faisant appel sur une instance d'une implémentation, ainsi qu'illustré ci-dessous :

Programme 8 : *Appel de méthodes par défaut*

```
ICouleur blue= new CouleurRGB(0,0,255);
System.out.println(blue.hexaRepresentation());
System.out.println(blue.isGrey());
ICouleur couleur=new CouleurHSL(0.99f,0.9f,0.08f);
System.out.println(couleur.hexaRepresentation());
System.out.println(couleur.isGrey());
```

2.3 Peut-on mettre des méthodes de classe (statiques) dans des interfaces ?

On peut depuis Java 8 implémenter une méthode statique dans une interface, ainsi qu'illustré ci-dessous (code 9).

Programme 9 : *Méthode de classe dans une interface*

```
public interface ICouleur {
        ...
        public static boolean areNear(ICouleur c1, ICouleur c2) {
                return Math.abs((c1.red()-c2.red())/c1.red())<0.1
                                && Math.abs((c1.green()-c2.green())/c1.green())<0.1
                                && Math.abs((c1.blue()-c2.blue())/c1.blue())<0.1;
        }
}
```

Ces méthodes de classe (statiques) peuvent être appelées comme le sont les méthodes de classe définies dans des classes, par exemple ici : `boolean b=ICouleur.areNear(coul1, coul2);`.

3 Implémentation et extension d'interfaces

3.1 Extension d'interfaces

De même qu'une classe peut étendre une autre classe (par une relation d'héritage, via le mot-clef `extends` en Java), une interface peut étendre une autre interface.

On peut par exemple envisager d'écrire une interface ICouleurNommée, qui étend ICouleur en lui ajoutant une méthode name() :String, ainsi qu'illustré à la figure 5 et au code 10.

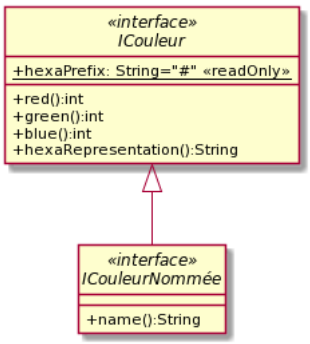

FIGURE 5 – Extension d'interface

Programme 10 : *Extension d'interface*

```
public interface ICouleurNommee extends ICouleur {
        public String nom();
}
```

Mais à la différence d'une classe, une interface peut en étendre plusieurs. Ainsi, dans l'exemple de la figure 6 et le code 11, ICouleurNommee étend 2 interfaces : ICouleur et IResource.

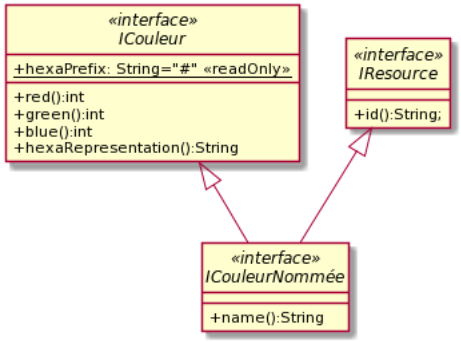

FIGURE 6 – Extension d'interfaces

Programme 11 : *Extension d'interface*

```
public interface ICouleurNommee extends ICouleur, IResource {
        public String nom();
}

public interface IResource{
        public String id();
}
```

Du point de vue des types, un objet typé par ICouleurNommee "est un" ICouleur et "est un" IResource. On pourra donc écrire (sans besoin de cast/transtypage explicite) :

Programme 12 : *Extensions mutiples et types*

```
ICouleurNommee cn;
... // initialisation de cn
ICouleur c=cn;
IResource r=cn;
```

Dans cet exemple, on peut invoquer les méthodes :
— name, red, green, blue, hexaRepresentation, et id sur cn,
— red, green, blue, et hexaRepresentation sur c, et
— id sur r.

3.2 Implémentation multiple

Une même classe peut implémenter plusieurs interfaces. Elle doit alors si elle est concrète implémenter toutes les méthodes de toutes les interfaces implémentées. On dit souvent que Java est à héritage simple et implémentation multiple. Du point de vue des types, une instance de la classe peut être affectée à un élément typé par n'importe laquelle des interfaces implémentées.

L'implémentation multiple d'interfaces, couplée à l'extension multiple des interfaces entre elles, permet une meilleure structuration hiérarchique des types. L'implémentation multiple ne permet pas de pallier totalement l'absence d'héritage multiple en Java, mais participe largement à la construction de hiérarchies de types, comme en atteste par exemple la façon dont sont structurées les collections de l'API Java.

3.3 Extension et implémentation

Une même classe peut étendre une (seule) autre classe et implémenter plusieurs interfaces. Dans ce cas, on doit déclarer en Java la classe étendue avant les interfaces implémentées : `public class A extends B implements I1, I2`.

3.4 La fameuse configuration du diamant

Depuis l'introduction des méthodes par défaut dans les interfaces Java, on peut rencontrer en Java la fameuse configuration en diamant (en losange serait plus juste). Elle apparaît par exemple dans les configurations de la figure 7, dans lesquelles on a indiqué avec $<< default >>$ les méthodes ayant une implémentation par défaut (ce qui est une notation "maison", car la notation n'existe pas en UML).

Dans cet exemple, on a trois niveaux hiérarchiques : celui des grands-mères (IGM1 et IGM2), celui des mères (IM1, IM1bis, IM2 et M2bis) et celui des filles (F1 et F2). Dans la première comme dans la deuxième configuration, la question qui se pose est : quelle est la méthode m() dont les filles disposent ?

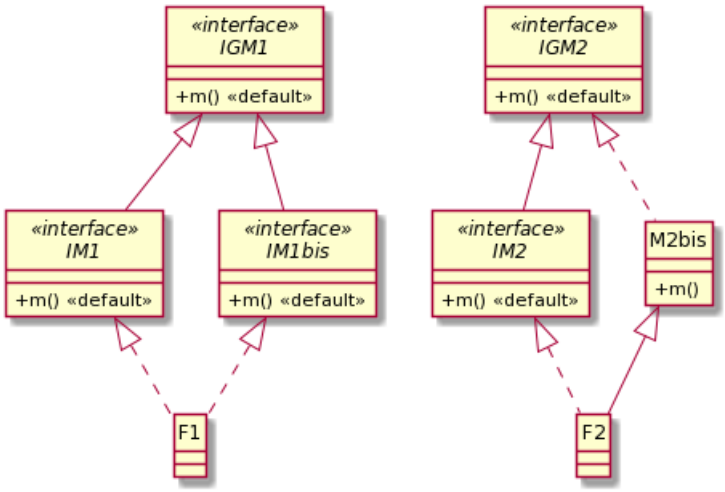

FIGURE 7 – Deux configurations en diamant

En Java, dans le premier cas, F1 doit absolument choisir quelle implémentation elle veut, en plaçant dans F1 une méthode m(), qui appellera la méthode héritée souhaitée : `IM1.m()` ou `IM1bis.m()`.

En Java, dans le deuxième cas, c'est ce qui est hérité de la classe qui prime sur ce qui est hérité de l'interface. Donc la méthode m de F2 est celle de M2bis.

3.5 Implémentation partielle avec des classes abstraites

Nous avons vu que quand une classe concrète implémente une interface, elle a l'obligation de donner une implémentation à toutes les méthodes de l'interface. En revanche, une classe abstraite qui implémente une interface peut en donner une implémentation partielle : les méthodes non implémentées sont alors laissées abstraites et devront être implémentées par une sous-classe. Cette classe abstraite peut permettre de factoriser pour plusieurs implémentations différentes des comportements communs, ou faire des premiers choix d'implémentation qui devront être complétés par la suite.

4 Utilisation des interfaces

Du point de vue de l'utilisateur, les interfaces permettent de s'abstraire de la façon dont un type est implémenté, en n'utilisant que des méthodes prévues dans l'interface, et sans faire de suppositions sur la façon dont les choix d'implémentation. On peut ainsi plus facilement changer d'implémentation.

Imaginons par exemple une palette de couleurs. Nous n'avons pas besoin de connaître les différents implémentations envisageables de couleurs : le code de la Palette peut uniquement se baser sur l'interface comme dans le code 13.

Programme 13 : *Une classe pour les palettes de couleurs*

```
public class Palette {
        private List<ICouleur> couleurs =new ArrayList<>();

        public void ajoutCouleur(ICouleur c) {
                if (!c.isGrey()) {// on ne veut pas de gris dans notre palette
                        couleurs.add(c);
                }
        }

        public void affichePalette() {
                for (ICouleur c:couleurs) {
                        System.out.println(c.hexaRepresentation());
                }
        }
}
```

On peut ainsi ensuite ajouter des couleurs RGB ou des couleurs HSL à notre palette comme dans le code 14.

Programme 14 : *Utilisation d'une palette de couleurs*

```
Palette rose=new Palette();
CouleurRGB dragee=new CouleurRGB(196, 110, 173);
CouleurRGB fuschia=new CouleurRGB(204, 9, 130);
CouleurHSL rosenormal=new CouleurHSL(316, 42, 60);
rose.ajoutCouleur(dragee);
rose.ajoutCouleur(fuschia);
rose.ajoutCouleur(rosenormal);
rose.affichePalette();
```

Attention, deux implémentations (correctes) différentes d'une même interface sont équivalentes d'un point de vue fonctionnel, mais pas d'un point de vue "performances". Par exemple dans notre exemple de couleurs, calculer un camaïeu de couleurs sera très simple (d'un point de vue calculatoire) en HSL, et beaucoup moins en RGB, et bien sûr calculer les composantes RGB d'une couleur sera trivial avec le codage RGB et calculatoirement plus complexe avec une représentation HSL.

5 Interfaces et types abstraits

L'une des utilisations les plus classiques des interfaces est la représentation des types abstraits de données. Nous l'illustrons avec l'exemple du type `Pile` (un tel type existe bien sûr dans l'API Java mais nous en faisons un autre).

Au code 15 se trouve une interface décrivant les opérations disponibles sur les piles. Le <T> exprime ce que l'on appelle le paramètre de généricité, ici c'est un type appelé `T` qui sera passé en paramètre à un autre type, en l'occurrence le type `Pile`. En d'autres termes, `Pile` est paramétrée par un type appelé `T`.

Programme 15 : *Une interface Pile*

```
public interface Pile<T>
{
  void empile(T t);
  void depile();
  T sommet();
  boolean estVide();
}
```

`PileAL` est une classe (également générique) implémentant l'interface `Pile` à l'aide d'une ArrayList pour stocker les éléments.

Programme 16 : *Une classe PileAL*

```
public class PileAL<T> implements Pile<T>
{
  ArrayList<T> v=new ArrayList<T>();
  public PileAL(){}
  public void empile(T t){v.add(t);}
  public void depile(){v.remove(v.size()-1);}
  public T sommet(){return v.get(v.size()-1);}
  public boolean estVide(){return v.isEmpty();}
  public String toString(){return "Pile "+v.toString();}
  }
```

Le programme `main` montre comment on crée une pile en passant un paramètre de type réel (`Integer`).

Programme 17 : *Une utilisations des piles*

```
        Pile<Integer> p1 = new PileAL<Integer>();
p1.empile(7);
p1.empile(5);
p1.empile(4);
System.out.println(p1);
p1.depile();
System.out.println(p1);
```

Quel est l'intérêt d'avoir fait une interface dans ce cas ? Imaginons un programme utilisateur qui ait besoin d'une pile pour effectuer une descente en profondeur dans un graphe, l'évaluation d'une expression arithmétique, etc. Ce programme peut être

écrit en utilisant l'interface `Pile` comme elle est utilisée pour déclarer la pile `p1` dans le programme précédent. Il n'y a aucun moyen de tricher et de s'appuyer sur une quelconque hypothèse d'implémentation de la pile pour écrire ce programme utilisateur. Seule l'instruction de création de la pile concrète (comme `new PileAL<Integer>()`) devra être modifiée si on décide pour des raisons d'efficacité pour certaines opérations (comme l'ajout ou le retrait à une position interne) de changer de mode d'implémentation de la pile, et que l'on préfère une `PileListe`, implémentée à l'aide d'une liste chaînée.