**Notes de cours**
Programmation II
INF2120

Version 1.0
20 septembre 2013

# Les classes abstraites et les interfaces en Java

Mélanie Lord
UQAM

## Table des matières

# 1. Méthodes abstraites versus concrètes

### Méthode concrète

Une méthode concrète est une méthode complètement définie qui possède un entête et un corps (des instructions, une implémentation). Ce sont les méthodes qu'on a vues jusqu'à maintenant.

Exemple 1.1

```
public int getAge() {
    return age;
}
```

: entête de la méthode
: corps de la méthode

### Méthode abstraite

Une méthode abstraite est une méthode qui ne possède pas de corps (pas d'implémentation). On déclare une méthode abstraite avec le modificateur `abstract`, comme ceci :

```
public abstract int getAge();    //se termine par un point-virgule
```

***Note :*** *Une méthode abstraite devra éventuellement être concrétisée (implémentée) par une classe enfant.*

# 2. Les classes abstraites

## 2.1 Notions de base

Une classe abstraite est une **classe qui ne peut pas être instanciée** (on ne peut pas construire d'objets de cette classe avec l'opérateur `new`). Une telle classe ne peut que servir de classe de base pour des classes dérivées. On peut cependant déclarer une variable d'une classe abstraite.

Pour déclarer une classe abstraite, on utilise encore le modificateur `abstract` qu'on ajoute devant le mot `class` lors de la définition d'une classe, comme ceci :

```
public abstract class A {}

A a1;  //OK on peut déclarer une variable d'une classe abstraite
A a2 = new A(); //NON, instanciation interdite
                //(erreur de compilation)
```

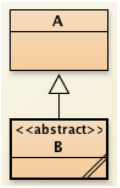
Une classe abstraite :

- **Peut contenir tout ce qu'une classe concrète** (sans le mot abstract) **peut contenir** (variables d'instance ou de classe, constantes d'instance ou de classe, méthodes concrètes, constructeurs… ) PLUS des **méthodes abstraites**.
- **Peut ne contenir aucune méthode abstraite** (mais la plupart du temps, elle en contient).
- **Peut hériter d'une classe** (comme une classe concrète).
- **Peut implémenter des interfaces** (comme une classe concrète).

Mélanie Lord
Lord.melanie@uqam.ca

Lorsqu'une classe **contient** au moins une **méthode abstraite**, cette classe **doit être déclarée abstraite**.

Une classe dérivée d'une classe non abstraite peut être déclarée abstraite : une classe abstraite n'est pas toujours la classe la plus haute dans la hiérarchie.




Les méthodes abstraites d'une classe abstraite ne peuvent pas être déclarées `private`.

Les méthodes abstraites d'une classe abstraite ne peuvent pas être déclarées `final`.

Les méthodes abstraites d'une classe abstraite **peuvent être concrétisées par ses sous-classes**.

**Lorsqu'une sous-classe ne concrétise (n'implémente, ne redéfinit) pas toutes les méthodes abstraites**, cette sous-classe **doit alors être déclarée abstraite**.

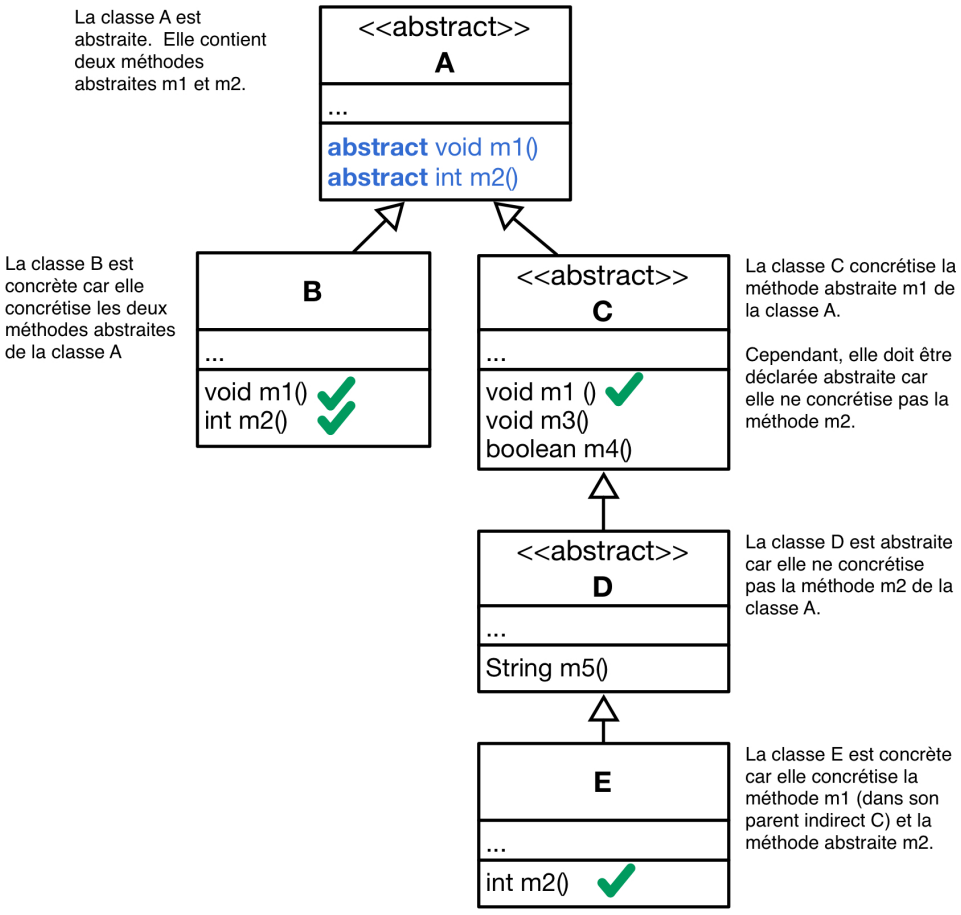



**Figure 2.1 Hiérarchie de classes avec classes et méthodes abstraites.**

Dans la hiérarchie de la figure 2.1, les déclarations et instanciations de variables permises sont (les instructions mises en commentaire ne compilent pas) :

```
public class Test {
  public static void main (String [] args) {

    A a;
    //a = new A(); //NON - A est abstraite
    a = new B();
    //a = new C(); //NON - C est abstraite
    //a = new D(); //NON - D est abstraite
    a = new E();

    B b;
    b = new B ();

    C c;
    //c = new C(); //NON - C est abstraite
    //c = new D(); //NON - D est abstraite
    c = new E();

    D d;
    //d = new D();
    d = new E ();

    E e;
    e = new E ();
  }
}
```

Exemple 2.1 : concrétisation de la méthode `affiche` de la classe abstraite `Affichable` dans les trois sous-classes `Entier`, `Flottant` et `Point` (tiré de [1] et [2]).

```
public abstract class Affichable {
  //methode abstraite (sans instructions)
  public abstract void affiche ();
}

public class Entier extends Affichable {
  private int valeur;

  public Entier ( int n ) {
    valeur = n;
  }

  //Concretisation de la methode abstraite affiche
  public void affiche () {
    System.out.println ( "Entier de valeur " + valeur );
  }
}
```

```
public class Flottant extends Affichable {
  private float valeur;

  public Flottant ( float x ) {
    valeur = x;
  }

  //Concretisation de la methode abstraite affiche
  public void affiche () {
    System.out.println ( "Flottant de valeur " + valeur );
  }
}

public class Point extends Affichable {
  private int x; //coordonnee x
  private int y; //coordonnee y

  public Point ( int x, int y ) {
    this.x = x;
    this.y = y;
  }

  //Concretisation de la methode abstraite affiche
  public void affiche () {
    System.out.println ( "Un point de coordonnées " +
                         "(" + x + ", " + y + ")" );
  }
}


public class TestsAffichable {

  public static void main (String[] params) {

    //Tout objet d'une classe concrete derivant
    //de Affichable pourra etre place dans tab
    Affichable[] tab = new Affichable[5];

    tab[0] = new Entier (23);
    tab[1] = new Flottant (3);
    tab[2] = new Point (23, 99);
    tab[3] = new Entier (233);
    tab[4] = new Point (-22, 9999);

    for ( int i = 0; i < tab.length; ++i ) {
      tab[i].affiche();
    }
  }
}
```

**Note** : Lorsqu'on concrétise une méthode, on doit enlever le mot abstract dans l'entête de la méthode.

La méthode `main` de la classe `TestsAffichable` affiche :

```
Entier de valeur 23
Flottant de valeur 3.0
Un point de coordonnées (23, 99)
Entier de valeur 233
Un point de coordonnées (-22, 9999)
```

Exemple 2.2 : Classe abstraite sans aucune méthode abstraite ( tiré de [1] ).

```
public abstract class FigureGeometrique {
  private int couleur;
  private boolean estPleine;

  /**
   * Une figure geometrique de couleur 0, vide
   */
  public FigureGeometrique () {
    this (0, false);
  }

  /**
   * Une figure geometrique
   * @param couleur sa couleur
   * @param pleine true si pleine, false sinon
   */
  public FigureGeometrique (int couleur, boolean pleine) {
    setCouleur (couleur);
    setEstPleine (pleine);
  }

  //GETTERS ET SETTERS
  /**
   * Determine la couleur de la figure
   * @return sa couleur
   */
  public int getCouleur () {
    return couleur;
  }

  /**
   * Determine si vide ou non
   * @return true si vide, false sinon
   */
  public boolean estVide () {
    return !estPleine;
  }
```

```
  /**
   * Changer la couleur
   * @param couleur la nouvelle couleur
   */
  public void setCouleur (int couleur) {
    this.couleur = couleur;
  }

  /**
   * Changer la valeur de pleine
   * @param pleine true si pleine, false sinon
   */
  public void setEstPleine (boolean pleine) {
    this.estPleine = pleine;
  }
}

public abstract class Figure2D extends FigureGeometrique {
  //coordonnees de la figure dans un plan cartesien
  private int x; //abcisse
  private int y; //ordonne
  /**
   * Figure2D en (0,0) de couleur 0 et vide
   */
  public Figure2D () {
    this ( 0, 0 );
  }

  /**
   * Figure2D de couleur 0, vide
   * et de coordonnees x et y
   * @param x position en x
   * @param y position en y
   */
  public Figure2D (int x, int y) {
    this ( x, y, 0, false );
  }

  /**
   * Figure2D initialisee avec les valeurs des parametres
   */
  public Figure2D (int x, int y, int couleur, boolean pleine) {
    super (couleur, pleine);
    changerPosition (x, y);
  }

  /**
   * Donne la position x de la figure2D
   * @return position x
   */
  public int getX () {
    return this.x;
  }
```

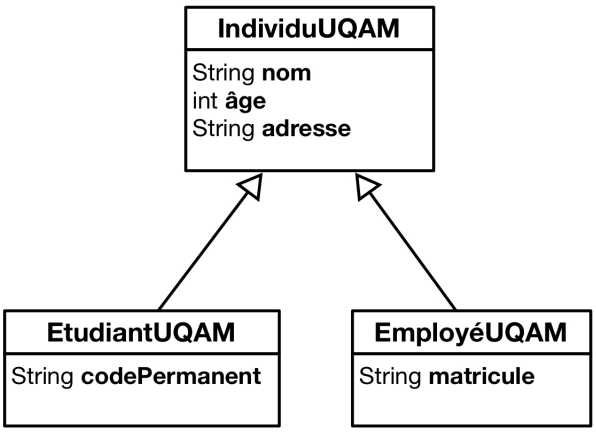
```
  /**
   * Donne la position y de la figure2D
   * @return position y
   */
  public int getY () {
    return this.y;
  }

  /**
   * Changer la position de la figure2D
   * @param x position en x
   */
  public void setX (int x) {
    this.x = x;
    this.y = y;
  }

  /**
   * Changer la position de la figure2D
   * @param y position en y
   */
  public void setY (int y) {
    this.y = y;
  }

  /**
   * Changer la position de la figure2D
   * @param x position en x
   * @param y position en y
   */
  public void changerPosition (int x, int y) {
    this.x = x;
    this.y = y;
  }

  //METHODES ABSTRAITES

  /**
   * Determine la surface de la figure2D
   * @return la surface de la figure2D
   */
  public abstract double surface ();

  /**
   * Representation de la figure 2D sous forme de String
   */
  public abstract String toString();
}
```

## 2.2 Intérêt des classes abstraites

On utilise une classe abstraite :

1. Lorsqu'on a besoin d'une classe pour regrouper des caractéristiques (attributs et méthodes) communes à plusieurs sous-classes, mais dont les instances de cette classe ne font pas de sens dans le contexte de l'application, soit parce que la définition de cette classe est incomplète en soi, soit parce qu'elle est trop générale (trop abstraite).

   Par exemple, dans la hiérarchie de classes ci-dessous, la classe IndividuUQAM pourrait très bien être abstraite parce que le concept "Individu UQAM" est vague et ne signifie pas grand-chose en soi : pour s'imaginer un individu à l'UQAM, on pense à un étudiant, un professeur, un employé de bureau, un technicien... qui sont des concepts concrets.



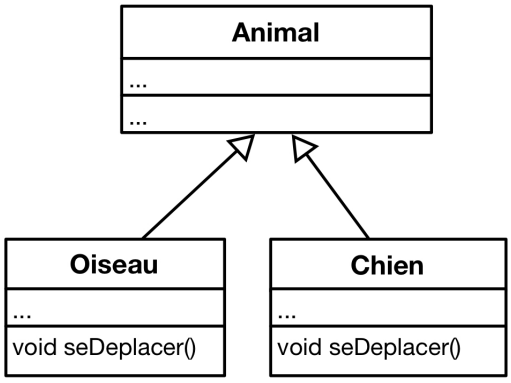

   Dans le contexte d'une application de gestion des étudiants et des employés de l'UQAM, par exemple, il est fort probable qu'on n'ait pas besoin de créer et gérer des objets de type IndividuUQAM, que des objets concrets EtudiantUQAM et EmployéUQAM.

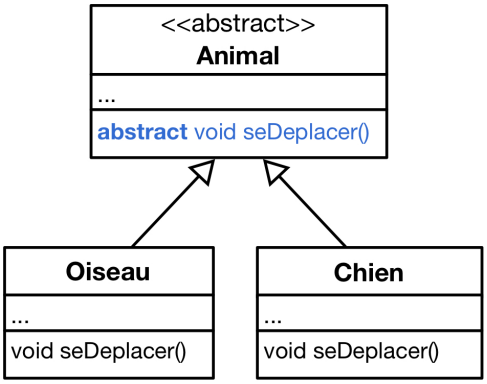
   Cependant, la classe IndividuUQAM demeure utile pour le regroupement des caractéristiques communes des sous-classes EtudiantUQAM et EmployéUQAM.

2. Pour définir des méthodes abstraites dans le but de forcer l'implémentation de ces méthodes dans des sous-classes concrètes : assure l'existence de ces méthodes dans les sous-classes. L'implémentation de ces méthodes pourra varier d'une sous-classe à l'autre.

   Par exemple, regardons les deux hiérarchies de classes suivantes :

   - Dans les deux hiérarchies, **on ne veut pas définir la méthode seDeplacer dans la classe Animal** car celle-ci ne fait pas sens pour un animal "en général" étant donné qu'un mode de déplacement est particulier à un animal concret (un chat, un poisson, un oiseau…).




```
public class Animal {}
```

```
public class Oiseau extends Animal {
  public void seDeplacer() {
    System.out.println("Je vole");
  }
}

public class Chien extends Animal {
  public void seDeplacer() {
    System.out.println("Je cours");
  }
}
```

```
public abstract class Animal {
  public abstract void seDeplacer();
}
```

```
public class Oiseau extends Animal {
  public void seDeplacer() {
    System.out.println("Je vole");
  }
}

public class Chien extends Animal {
  public void seDeplacer() {
    System.out.println("Je cours");
  }
}
```

Supposons qu'on veuille implémenter une méthode statique qui affiche le mode de déplacement des animaux d'un tableau reçu en paramètre.

Dans la hiérarchie de gauche, il faut tester la classe de l'objet Animal (`Oiseau` ou `Chien`) et effectuer conversion de type selon la classe de l'objet, pour que le code compile et ne plante pas.

```
public static void afficherDeplacement(Animal [] lesAnimaux) {
  for (Animal a : lesAnimaux) {
    //tester si a est une instance de Oiseau
    if (a instanceof Oiseau) {
      ((Oiseau)a).seDeplacer();          //besoin d'un cast
    //tester si a est une instance de Chien
    } else if (a instanceof Chien) {
      ((Chien)a).seDeplacer();           //besoin d'un cast
    }
  }
}
```

**Note** :

- l'instruction a instanceof Oiseau pourrait être remplacée par `a.getClass() == new Oiseau().getClass()`.
- **Attention**, lorsqu'on se voit obligé de tester la classe d'un objet de cette manière, ceci est parfois une indication d'une mauvaise conception en POO (voir une meilleure solution dans l'exemple suivant – hiérarchie de droite – qui tire parti du polymorphisme).

Dans la hiérarchie de droite, on est **certain** qu'il y a une implémentation de la méthode `seDeplacer` dans les sous-classes `Chien` et `Oiseau`.

**Cette certitude permet d'exploiter le polymorphisme** : comme on l'a déjà vu, à l'exécution, le choix de la méthode `seDeplacer` se fera selon le type dynamique de l'objet sur lequel on appelle la méthode. Si c'est un `Oiseau`, c'est la méthode `seDeplacer` de la classe `Oiseau` qui sera exécutée, si c'est un `Chien`, c'est la méthode `seDeplacer` de la classe `Chien` qui sera exécutée.

```
public static void afficherDeplacement(Animal [] lesAnimaux) {

  for (Animal a : lesAnimaux) {
    a.seDeplacer();    //compile et pas besoin d'un cast
  }
}
```

**Note :**
Dans la méthode précédente, l'instruction `a.seDeplacer()` compile, car la méthode `seDeplacer()` existe dans la classe `Animal` (même si celle-ci est abstraite).

# 3. Les interfaces

## 3.1 Défnition d'une interface

Une interface est un **type**, comme toute classe en Java, mais c'est un type qu'on dit **abstrait**. Cela signifie qu'**on peut déclarer des variables d'un type interface**, mais qu'**on ne peut pas en instancier avec l'opérateur `new`** (comme c'est le cas des classes abstraites).

On définit une interface en utilisant le **mot réservé `interface`** (à la place du mot réservé `class`, lorsqu'on définit une classe) :

```
public interface A {...}

A a1;  //OK on peut déclarer une variable d'un type interface
A a2 = new A(); //NON, instanciation interdite d'un type
                //interface (erreur de compilation)
```

Une interface peut seulement contenir :

- des méthodes d'instance publiques **abstraites**
- des **constantes** de classe publiques (`public static final`)

Une interface ne peut donc pas contenir :

- de variables
- de constructeurs
- de méthodes concrètes

Exemple 3.1: Définition de l'interface `IOperateurMath` avec deux méthodes

```
public interface IOperateurMath {
  //retourne la somme de n1 et n2
  public abstract int addition (int n1, int n2);

  //retourne le resultat de la multiplication de n1 par n2
  public abstract int multiplication (int n1, int n2);
}
```

***Note** : Les méthodes d'une interface sont toujours abstraites et publiques et l'on peut omettre les mots-clés `public` et `abstract`.*

## 3.2 Implémentation d'une interface

Les méthodes d'une interface doivent être implémentées (concrétisées) par toutes les classes qui *implémentent* cette interface. Lorsqu'une classe implémente une interface, celle-ci (ou l'un de ces descendants) **s'engage à redéfinir** (concrétiser, implémenter) **toutes les méthodes** de cette interface.

Pour signifier qu'une classe implémente une interface (et donc toutes les méthodes abstraites de cette interface), on utilise le **mot réservé `implements`**, comme ceci :

```
//Signifie que la classe B implémente l'interface A
public class B implements A {...}
```

Exemple 3.2 :
Implémentation de l'interface `IOperateurMath` par la classe `OperateurMath`

```
//La classe OperateurMath s'engage à définir toutes les méthodes
//listées dans l'interface IOperateurMath.
public class OperateurMath implements IOperateurMath {

  //retourne la somme de n1 et n2
  public int addition (int n1, int n2) {
    return n1 + n2;
  }

  //retourne le resultat de la multiplication de n1 par n2
  public int multiplication (int n1, int n2) {
    return n1 * n2;
  }
}
```

Une classe peut implémenter plusieurs interfaces.

Exemple 3.3 : Implémentation de deux interfaces par la classe `OperateurMath2`

Voici deux interfaces :

```
public interface IOperateurEntier {
  //retourne la somme n1 + n2
  public abstract int somme (int n1, int n2);
  //retourne le resultat de n1 * n2
  public abstract int multiplication (int n1, int n2);
}

public interface IOperateurFlottant {
  //retourne la somme n1 + n2
  public abstract double somme (double n1, double n2);
  //retourne le resultat de n1 * n2
  public abstract double multiplication (double n1, double n2);
}
```

Voici la classe OperateurMath2 qui implémente ces deux interfaces :

```
public abstract class OperateurMath2
        implements IOperateurEntier, IOperateurFlottant {

  //IMPLEMENTATION DE L'INTERFACE IOperateurEntier

  //retourne la somme n1 + n2
  public int somme (int n1, int n2) {
    return n1 + n2;
  }

  //retourne le resultat de n1 * n2
  public int multiplication (int n1, int n2) {
    return n1 * n2;
  }

  //IMPLEMENTATION DE L'INTERFACE IOperateurFlottant

  //retourne la somme n1 + n2
  public double somme (double n1, double n2) {
    return n1 + n2;
  }

  //retourne le resultat de n1 * n2
  public double multiplication (double n1, double n2) {
    return n1 * n2;
  }
}
```

***Note*** *: Lorsqu'il y a plus d'une interface à implémenter, on les énumère après le mot* `implements`*, en les séparant par des virgules.*

Une classe qui ne définit pas toutes les méthodes de toutes les interfaces qu'elle implémente doit être déclarée abstraite. Dans ce cas, c'est à un ou des descendants de cette classe que reviendra la responsabilité de définir le reste des méthodes des interfaces implémentées.

Exemple 3.4 : Implémentation partielle d'une interface

```
public abstract class OperateurMath3
        implements IOperateurEntier {

  //IMPLEMENTATION PARTIELLE DE L'INTERFACE
  //IOperateurEntier

  //retourne la somme n1 + n2
  public int somme (int n1, int n2) {
    return n1 + n2;
  }
}
```

```
public class OperateurMath4 extends OperateurMath3 {

  //IMPLEMENTATION PARTIELLE (DU RESTE DE)
  //L'INTERFACE IOperateurEntier

  //retourne le resultat de n1 * n2
  public int multiplication (int n1, int n2) {
    return n1 * n2;
  }
}
```

Une interface peut hériter (**extends**) d'une ou plusieurs interfaces. Dans ce dernier cas, on parle d'héritage multiple d'interfaces. (Rappelons cependant que l'héritage multiple pour les classes n'est pas permis en Java.)

Une interface qui hérite d'autres interfaces comprend alors ses propres méthodes abstraites PLUS toutes les autres méthodes abstraites de toutes les interfaces dont elle hérite.

Exemple 3.5 : héritage multiple d'interfaces.

```
public interface IOperateurMath
        extends IOperateurEntier, IOperateurFlottant {

  //retourne le resultat de la division reelle de n par 2
  public abstract double division (double n1, double n2);

  //Cette interface contient la methode division PLUS
  //toutes les methodes de l'interface IOperateurEntier PLUS
  //toutes les methodes de l'interface IOperateurFlottant.

}
```

***Note :***
*Une classe qui implémente l'interface `IOperateurMath` ci-dessus devra implémenter toutes les méthodes (abstraites) contenues dans l'interface `IOperateurMath`, `IOperateurEntier` et `IOperateurFlottant` (sinon elle devra être déclarée abstraite).*

Une classe peut à la fois hériter d'une autre classe et implémenter une ou plusieurs interfaces. Par exemple, la classe `B` suivante hérite de la classe `A` et implémente les interfaces `IA` et `IB`

```
public class B extends A implements IA, IB {...}
```

Exemple 3.6 : Implémentation de l'interface IAnimal

Voici l'interface IAnimal :

```
public interface IAnimal {
  public abstract void seDeplacer();
  public abstract void sIdentifier();
  public abstract void communiquer();
}
```

Implémentation de IAnimal par la classe Chien :

```
public class Chien implements IAnimal {
  private String race;
  private int age;
  private String jappement;

  public Chien (String race, int age, String jappement) {
    this.race = race;
    this.age = age;
    this.jappement = jappement;
  }

  public void seDeplacer() {
    System.out.println("Je marche et je cours");
  }

  public void sIdentifier() {
    System.out.println("Je suis un " + race +
                       " et j'ai " + age + " an(s)");
    }

  public void communiquer() {
    System.out.println(jappement);
  }
}
```

Implémentation de IAnimal par la classe Oiseau :

```
public class Oiseau implements IAnimal {
  private String chant;
  private String sorte;
  private String typeDeBec;

  public Oiseau (String chant, String sorte, String typeBec) {
    this.chant = chant;
    this.sorte = sorte;
    this.typeDeBec = typeBec;
  }

  public void seDeplacer() {
    System.out.println("Je vole !");
  }
```

```
  public void sIdentifier() {
    System.out.println("Je suis un(e) " + sorte +
                       " et mon bec est " + typeDeBec);
  }

  public void communiquer() {
    System.out.println(chant);
  }
}
```

Implémentation de IAnimal par la classe Poisson :

```
public class Poisson implements IAnimal {
  private String couleur;

  public Poisson (String couleur) {
    this.couleur = couleur;
  }

  public void seDeplacer() {
    System.out.println("Je nage");
  }

  public void sIdentifier() {
    System.out.println("Je suis un poisson " + couleur);
  }

  public void communiquer() {
    System.out.println("...");
  }
}
```

## 3.3 Utilisation de variables d'un type interface

Comme une interface est un type, on peut utiliser une interface :

- Pour déclarer une variable (ou un tableau) - mais pas l'instancier !

```
IOperateurMath op; //OK declaration

IOperateurMath op = new IOperateurMath; //NON – instanciation

IOperateurMath [] tab = new IOperateurMath [3];//OK déclaration
tab[0] = new IOperateurMath() //NON – instanciation
```

- Comme type de paramètre d'une méthode

```
public setOperateurMath (IOperateurMath op) {...}
```

- Comme type de la valeur de retour d'une méthode

```
public IOperateurMath getOperateur() {...}
```

**On peut affecter à une variable de type interface toute instance d'une classe qui implémente cette interface.**

Exemple 3.7 : Utilisation de variables/paramètres/type de retour de type interface

En considérant les classes `Poisson`, `Chien` et `Oiseau` ainsi que l'Interface `IAnimal`, on pourrait écrire la classe `Test` suivante :

```
public class Test {

  //Affiche les animaux de ce tableau
  public static void afficher (IAnimal [] lesAnimaux) {
    for (IAnimal a : lesAnimaux) {
      if (a != null) {
        a.sIdentifier();
        a.seDeplacer();
        a.communiquer();
        System.out.println("-----------------");
      }
    }
  }

  //Retourne un tableau qui contient seulement les instances
  //de Poisson
  public static IAnimal[] trierLesPoissons
                              (IAnimal [] lesAnimaux) {

    IAnimal [] lesPoissons = new IAnimal [lesAnimaux.length];
    int j = 0;
    for (IAnimal a : lesAnimaux) {
      if (a != null && a instanceof Poisson) {
        lesPoissons[j] = a;
        j++;
      }
    }
    return lesPoissons;
  }

  public static void main () {

    //Declaration de variables de type IAnimal et instanciation
    //avec une instance d'une classe qui implemente IAnimal
    IAnimal unPoisson = new Poisson ("rouge");
    IAnimal unChien = new Chien ("chiwawa", 2, "wif wif");
    IAnimal unOiseau = new Oiseau
             ("chique di di di", "mesange", "court et mince");

    //Declaration d'un tableau de type IAnimal
    IAnimal [] lesAnimaux = new IAnimal [4];
```

```
    //Instanciation des cases du tableau avec une instance
    //d'une classe qui implémente IAnimal
    lesAnimaux[0] = unPoisson;
    lesAnimaux[1] = unChien;
    lesAnimaux[2] = unOiseau;
    lesAnimaux[3] = new Poisson("doré");

    System.out.println("AFFICHER LES ANIMAUX\n");
    afficher(lesAnimaux);

    IAnimal [] lesPoissons = trierLesPoissons(lesAnimaux);

    System.out.println("\nAFFICHER LES POISSONS\n");
    afficher(lesPoissons);
  }
}
```

La méthode main ci-dessus affiche :

```
AFFICHER LES ANIMAUX

Je suis un poisson rouge
Je nage
...
----------------
Je suis un chiwawa et j'ai 2 an(s)
Je marche et je cours
wif wif
----------------
Je suis un(e) mesange et mon bec est court et mince
Je vole !
chique di di di
----------------
Je suis un poisson doré
Je nage
...
----------------

AFFICHER LES POISSONS

Je suis un poisson rouge
Je nage
...
----------------
Je suis un poisson doré
Je nage
...
----------------
```

***Note** : Dans la méthode afficher, grâce au polymorphisme, les méthodes exécutées seront choisies en fonction du type dynamique de l'objet a.*

## 3.4 Rappel : contrat versus implémentation

Comme on l'a déjà vu avec le principe d'encapsulation des données, il est important de faire la distinction entre

- le quoi **et** le comment
- les services offerts (contrat) **et** la manière dont ceux-ci sont implémentés (implémentation),

Une interface donne une liste de méthodes à implémenter cependant, celle-ci ne dit rien sur la manière dont ces méthodes devront être implémentées. La responsabilité de l'implémentation de ces méthodes appartient aux classes qui implémenteront cette interface et des classes différentes peuvent implémenter différemment la même interface.

Le contrat : Une interface constitue un contrat (ensembles de déclarations de méthodes publiques, sans leur implémentation). C'est la partie publique **nécessaire à l'utilisation**.

L'implémentation : Une classe (concrète) qui implémente une interface s'engage à respecter son contrat : implémenter les méthodes de cette interface. **Cette partie est privée, cachée à l'utilisateur**.

*Pour utiliser une classe qui implémente une interface, l'utilisateur n'a pas à connaître l'implémentation des méthodes de l'interface (le comment), seulement son contrat (le quoi).*

En séparant le contrat de l'implémentation, cela permet de modifier l'implémentation sans toucher au contrat (favorise la maintenance).

# 4. Exercices

## 4.1 Concrétisation d'une classe abstraite

Créer une classe (concrète) `Rectangle` dérivée de la classe `Figure2D` décrite à l'exemple 2.2, selon les spécifications suivantes :

Attributs d'instance (pour modéliser un rectangle) :

- hauteur
- largeur

Constructeurs :

- Un constructeur d'initialisation qui prend 2 paramètres : hauteur et largeur.
- Un constructeur d'initialisation qui prend 6 paramètres : couleur, pleine, x, y, hauteur et largeur.

Méthodes :

- Les getters et setters pour les attributs hauteur et largeur
- La concrétisaiton des méthodes abstraites de la classe Figure2D : surface() et toString()

## 4.2 Définition et implémentation d'une interface

Prenez la hiérarchie de classes de l'exemple 2.1 et modifier-la de telle sorte que la classe `Affichable` devienne une interface contenant la méthode (abstraite) `affiche()` et que les classes `Entier`, `Flottant` et `Point` implémente cette interface (au lieu d'hériter de Affichable).

Exécuter ensuite la méthode main de la classe TestsAffichable (exemple 2.1) pour tester vos classes et votre interface.

# 5. Solution des exercices

## 5.1 Solution exercice 4.1

```
public class Rectangle extends Figure2D {
  private int hauteur = 0;
  private int largeur = 0;

  public Rectangle (int hauteur, int largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }

  public Rectangle (int couleur, boolean pleine,
        int x, int y, int hauteur, int largeur) {

    super (x, y, couleur, pleine);
    this.hauteur = hauteur;
    this.largeur = largeur;
  }

  public int getHauteur () {
   return this.hauteur;
  }

  public int getLargeur () {
    return this.largeur;
  }

  public void setHauteur (int hauteur) {
    this.hauteur = hauteur;
  }
  public void setLargeur (int largeur) {
    this.largeur = largeur;
  }

  public double surface () {
    return hauteur * largeur;
  }

  public String toString () {
    return "Couleur  : " + getCouleur() + "\n" +
           "Pleine   : " + !estVide() + "\n" +
           "Position : (" + getX() + ", " + getY() + ")" + "\n" +
           "hauteur  : " + hauteur + "\n" +
           "largeur  : " + largeur + "\n";
  }
}
```

## 5.2 Solution exercice 4.2

```
public interface Affichable {
  //methode abstraite
  public abstract void affiche ();
}

public class Entier implements Affichable {
  private int valeur;

  public Entier (int n) {
    valeur = n;
  }

  //Concretisation de la methode abstraite affiche
  public void affiche () {
     System.out.println ( "Entier de valeur " + valeur );
  }
}

public class Flottant implements Affichable {
  private float valeur;

  public Flottant ( float x ) {
    valeur = x;
  }

  //Concretisation de la methode abstraite affiche
  public void affiche () {
    System.out.println ( "Flottant de valeur " + valeur );
  }
}

public class Point implements Affichable {
  private int x; //coordonnee x
  private int y; //coordonnee y

  public Point ( int x, int y ) {
    this.x = x;
    this.y = y;
  }

  //Concretisation de la methode abstraite affiche
  public void affiche () {
    System.out.println ("Un point de coordonnées " +
                        "(" + x + ", " + y + ")" );
  }
}
```

# 6. Références

[1] Laforest, Louise. *INF2120 – Programmation II (version alpha)*, Notes cours.
[2] Delannoy, Claude (2004). *Programmer en Java.* Paris : Eyrolles, 698 p.