Rappels sur les classes abstraites et interfaces

1 Classe abstraite

Une classe **abstraite** est une classe que l'on ne peut pas instancier, dont certaines méthodes sont abstraites et ne seront définies que dans les classes filles, comme par exemple la classe *Figure* vue au 1er semestre.

```
public abstract class Figure {
    protected Point origine;
    public Figure() {
   origine=new Point(0,0);
    }
   public Figure(Point p) {
        origine=p; //on stocke les adresses des objets
    }
     public Point getOrigine() {
   return this.origine;
    }
    public void setOrigine(Point p) {
   this.origine=p;
    }
    public abstract double p\'erimètre();
    public abstract double surface();
    public abstract void dessiner(PaD.PlancheADessin p);
    public abstract void dessiner(PaD.PlancheADessin p, java.awt.Color c);
}
```

Dans cet exemple, on a une variable d'instance *origine* les constructeurs et les méthodes set/get qui sont implémentées, mais les méthodes qui ne sont pas les mêmes selon la figure sont abstraites (e.g. surface() n'est pas la même pour un rectangle ou une ellipse).

2 Interface

La notion **d'interface** va un peu plus loin dans l'abstraction: tout est abstrait. Alors que la classe abstraite définit toujours une relation d'appartenance, l'interface se contente d'énoncer un comportement attendu. Par exemple, on pourrait dire qu'une figure doit être "mesurable" en terme de surface et de périmetre:

```
public interface Mesurable {
  double périmètre();
  double surface();
}
```

La classe figure *implémente* cette interface (dans le cas d'une interface, on utilise le mot *implements* au lieu de *extends*).

```
public abstract class Figure implements Mesurable {
    protected Point origine;
    public Figure() {
      origine=new Point(0,0);
    }
   public Figure(Point p) {
      origine=p; //on stocke les adresses des objets
    }
   public Point getOrigine() {
     return this.origine;
    }
   public void setOrigine(Point p) {
     this.origine=p;
    }
    public abstract void dessiner(PaD.PlancheADessin p);
    public abstract void dessiner(PaD.PlancheADessin p, java.awt.Color c);
}
```

L'avantage de l'interface, c'est qu'elle permet d'énoncer des comportements communs pour des objets n'ayant aucun lien d'appartenance. Par exemple, une maison peut tout aussi bien être mesurable:

```
public class Maison implements Mesurable {
  private double largeur;
  private double longueur;
  private String style;
  public Maison(double largeur,double longueur,String style) {
    this.largeur=largeur;
    this.longueur=longueur;
    this.style=style;
  }
  public double périmètre(){
    return (longueur+largeur)*2;
    }
  public double surface(){
    return longueur*largeur;
  }
}
```

En java, l'héritage est simple (un seul *extends*) mais on peut *implémenter* **plusieurs** *interfaces*.
On pourrait aussi avoir une interface *Dessinable* pour les figures:

```
  // import nécessaires...
  public interface Dessinable {
    void dessiner(PaD.PlancheADessin p);
    void dessiner(PaD.PlancheADessin p, java.awt.Color c);
  }
```

et avoir :

```
public abstract class Figure implements Mesurable, Dessinable {
```

3 Interface Comparable

De nombreuses interfaces existent en java, comme par exemple l'interface *Collection* qui est implémentée par la classe *ArrayList* (voir *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html)*.

Cette interface contient par exemple les méthodes *boolean add(E e)* ou *boolean isEmpty()* qui décrit un comportement commun à toutes les structures où l'on peut collectionner des éléments: on doit pouvoir ajouter un élément et savoir si la collection est vide.

Vous avez déjà utilisé l'interface *Comparable* pour écrire le tri par sélection. Implémenter cette interface revient à définir la méthode *int compareTo(T o)*. *x.compareTo(y)* doit renvoyer 0 si x et y sont égaux, un entier négatif si $x < y$ et un entier positif si $x > y$.

L'interface *Comparable* est implémentée par la plupart des classes prédéfinies de java. Par exemple, on peut comparer deux chaînes de caractères entre elles en utilisant *compareTo*.