Les interfaces

1. Rappels sur les types

Un type représente un ensemble de valeurs et les opérations qui sont possibles sur ces valeurs.

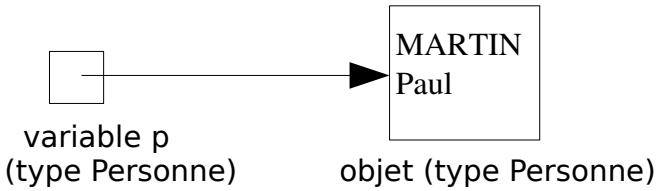
Type statique d'une variable : le type statique d'une variable (ou d'un paramètre, d'un attribut) est le type donné à une variable au moment de sa déclaration.

Remarque : le type statique est donné une fois pour toute ; il est visible dans le code source.

Type dynamique : le type dynamique d'une variable est le type de l'objet qu'elle référence pendant l'exécution du programme.

Exemple :

```
Personne p = new Personne("MARTIN", "Paul") ;
```



Dans cet exemple, le type de la variable (le type statique) et le type de l'objet (le type dynamique) sont identiques.

Remarque : une variable de type statique X peut référencer un objet de type X. Mais elle peut aussi référencer tout objet dont le type est un sous-type de X (c'est-à-dire tout objet qui est instance d'une-sous classe de X).

Java est un langage à typage statique : les types des variables sont fixés dans le code source et vérifiés à la compilation. Par contre, PHP est un langage à typage dynamique car les variables n'ont pas de type.

2. Les interfaces

Il ne faut pas confondre le terme *interface* au sens Java (qui est une construction du langage) avec le mot interface au sens interface home-machine : interface graphique, interface web.

Par ailleurs, en programmation objet, on appelle l'interface d'une classe tous les éléments publiques de la classe.

Les interfaces Java ressemblent aux classes abstraites mais s'en distinguent de plusieurs manières. On peut dire qu'une interface est une sorte de classe abstraite qui n'aurait pas d'attributs et qui n'aurait que des méthodes abstraites publiques. Une

interface est une sorte de classe abstraite « pure ».

Ainsi, les interfaces ne contiennent que des déclarations de méthodes publiques (on ne donne pas de corps) et de constantes (« variables » définies comme étant « static » et « final »).

L'héritage multiple n'existe pas en Java contrairement à d'autres langages comme le C++ : en Java, une classe ne peut pas hériter de plusieurs classes. Mais les interfaces sont un mécanisme très proche de l'héritage multiple qui permet d'éviter le problème de l'héritage en diamant.

Le grand intérêt des interfaces est qu'elles permettent un mécanisme proche de l'héritage multiple:
- une classe peut implémenter une ou plusieurs interfaces (mot-clé *implements),*
- une interface peut hériter d'une ou plusieurs interfaces (mot-clé *extends*),
- une interface ne peut pas hériter d'une classe.

Il maintenant possible d'affiner la définition du type donnée dans le cours sur les types simples :

Pour définir le type d'une variable, on dispose de deux possibilités :
- les types simples (appelés aussi types primitifs) ;
- les types d'objet définis grâce à des classes plus précisément grâce à des classes (concrètes et abstraites) et des interfaces.

3. Un exemple

On souhaite disposer d'une classe *Personne* qui possède trois attributs (*nom*, *prenom*, *notes* qui est un tableau d'entiers) qui implémente deux interfaces *Etudiant* et *Affichable*.

L'interface *Etudiant* possède une constante *AGE_MINIMAL* égale à 15 et une méthode *getNotes* qui renvoie un tableau d'entiers.

L'interface *Affichable* possède une méthode *afficher*.

Pour réaliser cela, il faut un fichier *Affichable.java* :

```
package demo_interfaces;

public interface Affichable {
    void afficher();
}
```

Un fichier *Etudiant.java* :

```
package demo_interfaces;

public interface Etudiant {
   static final int AGE_MINIMAL = 15;
   int[] getNotes();
}
```

Un fichier *Personne.java* :

```
package demo_interfaces;


public class Personne implements Etudiant, Affichable {

    private String nom;
    private String prenom;
    private int[] notes;

    Personne(String nom, String prenom) {
        this.nom = nom;
        this.prenom = prenom;
        notes = new int[20];
    }

    @Override
    public void afficher() {
        System.out.println(prenom + " " + nom);
    }

    @Override
    public int[] getNotes() {
        return notes;
    }
}
```

Et un fichier *Main.java* pour montrer que le tout fonctionne :

```
package demo_interfaces;

public class Main {
    public static void main(String[] args) {
        Main main = new Main();
        main.demarrer();
    }

    private void demarrer() {
        Etudiant e = new Personne("MARTIN", "Paul");
        Affichable a = new Personne("DUPONT", "Marie");
        a.afficher();
    }
}
```

Remarque 1 : dans l'interface *Etudiant*, on aurait pu avoir :

```
  int AGE_MINIMAL = 15;
```

au lieu de :

```
  static final int AGE_MINIMAL = 15;
```

Dans ce cas AGE_MINIMAL est considéré implicitement comme étant *static* et *final*.

Remarque 2 : il n'y a pas de constructeur dans une interface.

Remarque 3 : tout est publique dans une interface, donc le mot clé *public* est superflu bien que l'on puisse l'utiliser quand même.

Remarque 4 : il n'est pas possible d'instancier une interface. Il est donc impossible d'écrire :

```
Etudiant e = new Etudiant(); // impossible
```

Remarque 5 : il faut utiliser l'annotation *@Override* quand on implémente une méthode d'une interface.

La bibliothèque standard de Java contient beaucoup d'interfaces (par exemple la classe *JTextField* implémente six interfaces : *ImageObserver*, *MenuContainer*, *Serializable*, *Accessible*, *Scrollable*, *SwingConstants*).

Nous serons amenés plus à utiliser des interfaces déjà existantes plutôt qu'à en créer de nouvelles.