Les Interfaces

Notion d'interface

Si en programmation orientée Objet, la notion centrale est la Classe, nous savons qu'en Analyse et en Design orienté Objet, la notion fondamentale est le Type. Afin de pouvoir définir des Types au sens du Design orienté objet, Java propose les interfaces. Une interface est une sorte de classe super abstraite où toutes les méthodes sont abstraites et où il n'y a pas d'attributs (sauf éventuellement des constantes).

Comme les classes, les interfaces publiques doivent être sauvées dans un fichier de même nom que l'interface.

On les définira comme suit :

```
public interface MonInterface {
	int CONSTANTE = 1;
	void methode();
}
```

Dans une telle définition, les attributs indiqués sont d'office

```
public static final
```

et doivent donc être initialisés.

Les méthodes sont automatiquement `public abstract`. Si on ne peut pas avoir une autre visibilité dans une interface, on peut cependant y implémenter une méthode par défaut. Pour cela, il faut ajouter le mot clé `default` dans l'en-tête de la méthode :

```
public interface TestInterface {
	public default void methode1(){
		System.out.println("Implémentation par défaut de la méthode 1");
	}
	public default int methode2(){
		return 1;
	}
}
```

On peut également implémenter des méthodes statiques dans une interface.

On peut restreindre l'accès à une interface à son package si on ne met pas `public` devant `interface`.

Implémentation d'interface

Une classe n'étend pas une interface, elle l'implémente

```
public class MaClasse implements MonInterface {
```

Une classe peut implémenter plusieurs interfaces :

```
public class MaClasse implements MonInterface, TonInterface {
```

Une classe qui implémente une interface devra être déclarée abstraite si elle n'implémente pas toutes les méthodes abstraites de l'interface.

Héritage d'interfaces

Les interfaces peuvent hériter les unes des autres. L'héritage d'interface peut être multiple :

```
public interface Trois extends Un, Deux{
```

Dans ce cas `Un` et `Deux` doivent être des interfaces. En aucun cas, une interface ne peut hériter d'une classe.

La redéfinition et le masquage se fait comme pour les classes.

Collision de nom

Si des méthodes ont le même nom dans plusieurs interfaces implémentées par une même classe, et si ces méthodes ont des signatures différentes, la classe devra implémenter les différentes versions.

Si une méthode de même signature est héritée de deux interfaces, il faut qu'elles aient la même valeur de retour dans les deux interfaces. Sinon, il y aura erreur de compilation. Pour l'implémenter, il suffira de le faire une seule fois qui vaudra pour les deux interfaces.

Interfaces vides

Certains interfaces ne définissent aucune constante ni aucune méthode. Cela peut s'avérer nécessaire pour des raisons techniques. Mais cette sorte d'interface est également utilisée en Java pour marquer les classes qui les implémentent. Ainsi les interfaces `Cloneable`, `Serializable`, `Externalizable`, `java.util.EventListener` et `java.rmi.Remote` sont des interfaces vides servant à marquer les classes. Si, par exemple, une classe implémente `Cloneable`, elle aura le droit de redéfinir la méthode `clone()` héritée d'`Object` et de préciser ainsi comment on peut faire une copie de l'objet.

Interfaces fonctionnelles

Java 8 a introduit le concept d'interfaces fonctionnelles qui sont des interfaces disposant d'une unique méthode abstraite. L'utilisation de ces interfaces couplées aux expressions « lambda » permet d'apporter la puissance de la programmation fonctionnelle en Java. Une expression lambda peut être assimilée à une fonction anonyme, ayant potentiellement accès au contexte (variables locales et/ou d'instance) du code appelant. Ces fonctions anonymes peuvent être affectées dans une interface fonctionnelle. Le code de l'expression lambda servira ainsi d'implémentation pour la méthode abstraite de l'interface. La syntaxe utilisée est la suivante : (paramètres) -> {code}.

Exemple :

En Java 8, Comparator est une interface fonctionnelle (elle dispose d'une unique méthode abstraite):

```
public interface Comparator<T> {
       int compare(T t1, T t2);
```

```
}
```

Jusqu'à Java 7, pour utiliser une interface, il était nécessaire de définir une classe qui implémente cette interface et instancier un objet de cette classe (parfois au travers de classes anonymes comme dans l'exemple ci-dessous).

```
Comparator<Integer> comp = new Comparator<Integer>() {
	public int compare(Integer i1, Integer i2) {
		return i1-i2;
	}
};
```

Ci-dessous, voici un code équivalent au précédent utilisant les expressions lambda (Java 8).

```
Comparator<Integer> comp = (Integer i1, Integer i2) -> {
	return i1-i2;
};
```

Conclusion

Une des grandes règles de la programmation orientée Objet est :

> Toujours programmer en fonction des interfaces et non des implémentations.

Cela permet de définir des types tels que pensés lors de l'analyse et du design tout en cachant l'implémentation réelle.