Evolution des interfaces

Les interfaces jusqu'à Java 7 ne peuvent contenir que :

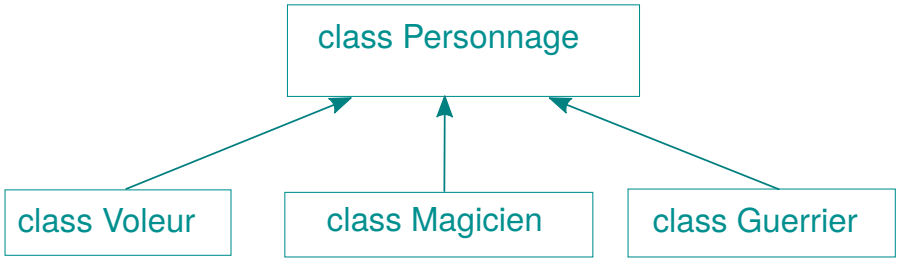
- des constantes
- des méthodes abstraites

Nouveautés depuis Java 8, elles peuvent aussi contenir :

1. des définitions par défaut pour les méthodes
2. des méthodes statiques

Illustration

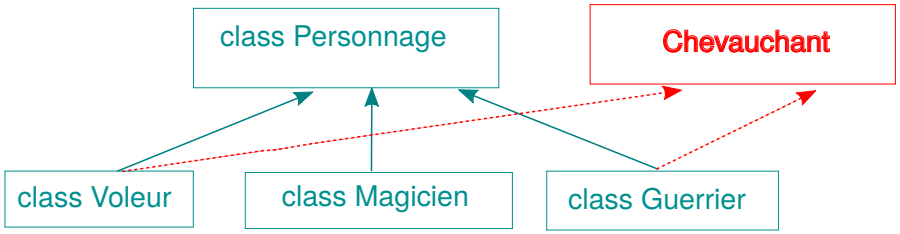
Reprenons notre exemple avec les personnages :



Supposons maintenant que certains personnages puissent chevaucher des montures.

Illustration

On pourrait donc imaginer la conception suivante :



Illustration

Pour le contenu de l'interface `Chevauchant`, on pourrait avoir :

```
interface Chevauchant
{
   void seDeplace();          // effectue le déplacement en chevauchant
   boolean peutDescendre();   // peut-on descendre de la monture ou pas ?
}
```

Supposons que dans la plupart des cas un personnage chevauchant une monture ne puisse pas en descendre ...

Méthode avec définition par défaut

Depuis Java 8, il est possible de donner une définition par défaut à certaines méthodes :

```
interface Chevauchant
{
   void seDeplace();
   default boolean peutDescendre() { return false; }
}
```

Méthode avec définition par défaut

Syntaxe :

```
interface UneInterface
{
   default définition par défaut de la méthode
}
```

Cette nouveauté soulève de nouvelles problématiques :

- quelles sont les règles d'utilisation des méthodes avec définition par défaut ?
- comment gérer les ambiguïtés pouvant se produire lors de définitions multiples (par des classes ou des interfaces) ?

Règle 1 : héritage et définition

Les définitions par défaut des méthodes s'héritent et peuvent être redéfinies plus bas dans les hiérarchies d'interfaces :

```
interface Cavalier extends Chevauchant
{
   default void seDeplace() { System.out.println("au trot"); }
}
```

☞ Inutile de redonner une définition par défaut à `peutDescendre` si on en est satisfait.

Il est aussi possible de le faire si on le souhaite :

```
interface Cavalier extends Chevauchant
{
   default void seDeplace() { System.out.println("au trot"); }
 default boolean peutDescendre() { return true; }
}
```

Règle 2 : redéfinitions inutiles dans les classes

Une classe n'est plus obligée de redéfinir les méthodes des interfaces qu'elle implémente si celles-ci ont une définition par défaut

☞ Si la classe `Guerrier` implémente l'interface `Cavalier` elle est instantiable en l'état

Elle peut néanmoins le faire si nécessaire :

```
class Guerrier extends Personnage implements Cavalier
{
   //...
   @Override
   boolean peutDescendre { return false; }
}
```

Règle 3 : les classes ont la précédence (1)

En cas d'ambiguïté entre une définition faite dans une classe et une définition par défaut dans une interface, c'est la méthode de la classe qui a la priorité :

```
class Personnage
{
   public void seDeplace() {
      System.out.println("en courant");
   }
}
interface Cavalier extends Chevauchant
{
   default void seDeplace() {
      System.out.println("au trot");
   }
}
class Guerrier extends Personnage implements Cavalier {}
```

☞ Appeler `seDeplace` sur une instance de `Guerrier` invoque la méthode de `Personnage`

Règle 3 : les classes ont la précédence (2)

La classe `Guerrier` peut bien sûr redéfinir la méthode `seDeplace` pour plutôt choisir celle de l'interface :

```
class Guerrier extends Personnage implements Cavalier
{
   public void seDeplace() {
      Cavalier.super.seDeplace();
   }
}
```

Règle 4 : les classes doivent lever les ambiguïtés (1)

Les conflits entre définitions par défaut sont désormais possibles :

```
interface Dragonier extends Chevauchant
{
   default void seDeplace() { System.out.println("vole"); }
}

interface SeTeleporte
{
   default void seDeplace { System.out.println("plop !"); }
}

class MageUltime extends Magicien implements Dragonier, SeTeleporte {
    // conflit sur la définition de seDeplace !
}
```

☞ Les classes qui implémentent des interfaces conflictuelles doivent lever l'ambiguïté

Règle 4 : les classes doivent lever les ambiguïtés (2)

La classe `MageUltime` doit lever l'ambiguïté :

```
class MageUltime extends Magicien implements Dragonier, SeTeleporte
{
   // un MageUltime se déplace comme un Dragonier
   public void seDeplace { Dragonier.super.seDeplace(); }
}
```

```
class MageUltime extends Magicien implements Dragonier, SeTeleporte
{
   // ou se téléporte
   public void seDeplace { SeTeleporte.super.seDeplace(); }
}
```

```
class MageUltime extends Magicien implements Dragonier, SeTeleporte
{
   // ou se déplace d'une façon qui lui est propre
   public void seDeplace {
      if (peutDescendre()) {
         SeTeleporte.super.seDeplace();
      } else {
         Dragonier.super.seDeplace();
      }
   }
}
```

Interface ou classe abstraite ?

La différence majeure est que les interfaces ne permettent pas de modéliser des **états** (attributs).

- ☞ Elles sont à privilégier s'il n'y a pas d'état.