# Introduction à la Programmation :

## Modificateur static
## Interface
## Types énumérés

Laboratoire d'Intelligence Artificielle
Faculté I&C







## Objectifs du cours de la semaine

- Le modificateur `static`
- Les arguments de `main`
- Comment définir des composants communs pour des classes non-liées par une relation d'héritage ?
  - ☞ la notion d'interface

Les types énumérés







## Pendant l'heure de cours

- Les types énumérés
- Approfondissements
  - ☞ Le principe fondamental
    *« Code to an interface not to an implementation »*
- (Arguments de `main`, pages 32 à 38)







## Le modificateur `static`

- S'utilise pour les variables et les méthodes
- Si on ajoute `static` à une variable :
  - La valeur de la variable est partagée entre toutes les instances de la classe
  - Pas possible pour les variables locales
- Si on ajoute `static` à une méthode :
  - On peut appeler la méthode sans construire d'objet
  - Diverses restrictions sur le contenu de la méthode statique

# Catégories de variables

Nos variables jusqu'à maintenant :

1. Variables d'instance
   - Décrivent les attributs d'un objet
   - Aussi appelées "variables dynamiques"
2. Variables locales
   - Déclarées à l'intérieur d'une méthode
3. Paramètres
   - Pour envoyer des valeurs à une méthode
   - S'utilisent comme des variables locales

Nouveau cette semaine :

4. Variables statiques = variables de classe
   - Ressemblent aux variables d'instance :
   - Déclarées en dehors des méthodes
   - Visibles partout dans la classe
   - Héritées par les sous-classes







# Variables d'instance et de classe

- La différence entre les variables d'instance et les variables de classe :
  - Le nombre de zones réservées en mémoire
- variable d'instance :
  - Réservation d'une zone pour chaque objet construit avec `new`
  - Résultat : chaque objet a sa propre zone/valeur pour la variable d'instance
- Variable de classe (statique) :
  - Déclaration précédée par `static`
  - Réservation d'une zone lors du chargement de la classe
  - Aucune zone réservée quand un objet est construit avec `new`
  - Résultat : tous les objets se réfèrent à la même zone/valeur pour la variable de classe







# Exemple

```
class Abc {
   public static void main(String[] args) {
      A v1, v2, v3;
      v1 = new A();
      v2 = new A();
      v3 = new A();
      v1.modifier();
      v2.modifier();
      A.c++;
   }
}
class A {
   int b = 1; // variable d'instance
   static int c = 10; // variable de classe
   void modifier() {
      b++;
      c++;
   }
}
```



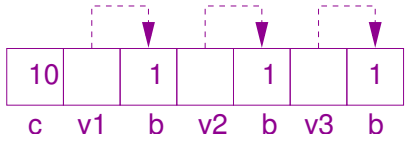




# Exécution de l'exemple

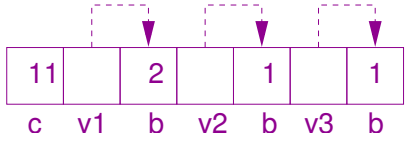
1. `A v1, v2, v3;`
   - Pour la classe `A` ...
   - Réservation d'une zone spéciale pour les variables statiques de la classe
   - Initialisation des variables statiques


2. `v1 = new A(); v2 = new A(); v3 = new A();`
   - Pour chaque objet construit ...
   - Réservation d'une zone pour `b`
   - Aucune réservation de zone pour `c`

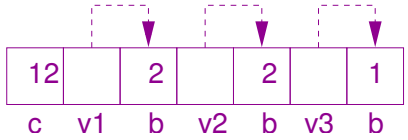


# Exécution de l'exemple (2)

3. `v1.modifier()`
   - Modification de la variable `b` de *l'objet* `v1`
   - Modification de la variable `c` de la *classe* `A`




4. `v2.modifier`
   - Modification de la variable `b` de *l'objet* `v2`
   - Modification de la variable `c` de la *classe* `A` (même zone en mémoire que pour `v1`)



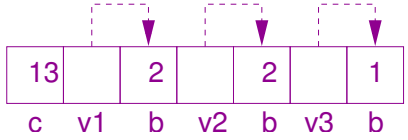







# Exécution de l'exemple (3)

5. `A.c++`
   - Il est possible d'accéder à une variable statique par le nom de la classe, sans passer par un objet
   - Possibilité qui existe *seulement* pour les variables statiques
   - Possible puisque les variables statiques sont initialisées dès que la classe est mentionnée
   - On peut accéder à une variable statique même si *aucun* objet n'a été construit
   - Evidemment pas possible si la variable statique est aussi privée










# Pourquoi utiliser `static`?

Variable d'instance vs variable de classe :

1. Modification d'une variable d'instance :
   - La valeur change *seulement* pour l'objet actuel

2. Modification d'une variable de classe :
   - La valeur change pour *tous* les objets de la classe

A quoi sert une variable statique ?

1. Bonne raison d'utiliser une variable statique :
   - Représentation d'une valeur qui est commune à tous les objets de la classe

2. Mauvaise raison d'utiliser une variable statique :
   - Programmer de manière *non* orientée objet en Java







# Valeur commune

Exercice : intégrons à nos classes `Avatar` le fait que 1000 est la durée de vie maximale possible pour un avatar (permet de faire des tests d'intégrité lors de la construction par exemple)

Considérons les deux versions suivantes :

- Avec une variable d'instance `dureeVieMax`
- Avec une variable statique `dureeVieMax`

# Classe Avatar1

- Version avec une variable d'instance pour la durée de vie maximale :

```
class Avatar1 {
   String nom;
   int energie;
   int dureeVie;
   int dureeVieMax
   Avatar1(String unNom,  int uneDureeVieMax) {
      nom = unNom;
      dureeVieMax = uneDureeVieMax;
      //.. reste des initialisation
   }
   //..
}
```







# Utilisation de Avatar1

- Version avec une variable d'instance pour la durée de vie maximale :

```
class UnJeu {

        public static void main(String[] args) {

      Avatar1[] avatars = new Avatar1[4];
      avatars[0] = new Avatar1("Albus", 1000);
      avatars[1] = new Avatar1("Gandalf", 1000);
      avatars[2] = new Avatar1("Oz", 1000);
      avatars[3] = new Avatar1("Legolas", 1000);
    // La modification de la duree de vie maximale
    // n'ecessite un parcours du tableau car chaque
    // Avatar1 a sa propre version de la variable:
      for (int i = 0; i < cs.length; i++){
         avatar1[i].dureeViemax = 1000;
      }
   }
}
```

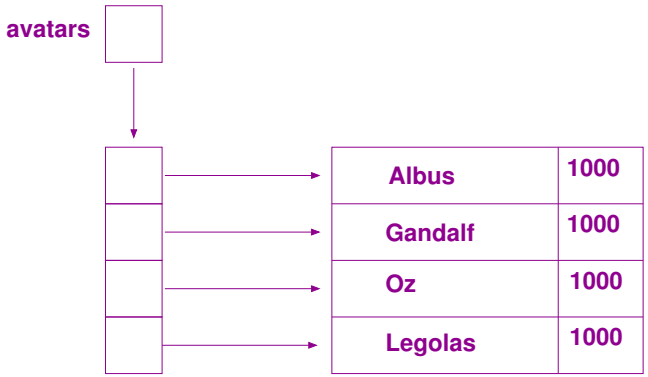






# Situation en mémoire

- Situation en mémoire :










# Avatar2

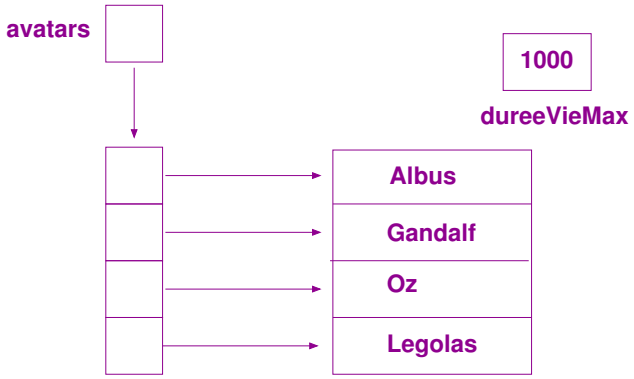
- Version avec une variable *statique* pour la durée de vie maximale

```
class Avatar2 {
        //...
    // Seule modification,  dureeVieMax devient static:
   static int dureeVieMax;
        //...
}
class UnJeu {

   public static void main (String[] args) {
   //    ...
   // Remplissage du tableau comme avant
   // Modification de la dur'ee de  vie maximale:
   // Aucun parcours du tableau n'ecessaire car
   // dureeVieMax est une variable statique
   // accessible a' l'aide de n'importe quel objet
   // de la classe ou bien par le nom de la classe:
      avatars[0].dureeVieMax = 1000;
      avatars[3].dureeVieMax = 1000; // Alternative
      Avatar2.dureeViemax = 1000; // Alternative
   }
}
```

# Situation en mémoire (2)

- Situation en mémoire :










# Constantes : `final` et `static`

- Un dernier détail ...
- Puisqu'une variable finale ne peut pas être modifiée :
  - Inutile de stocker une valeur pour chaque objet de la classe
  - Une constante est normalement toujours `final static`

```
class Cercle {
   double rayon, surface;
   // Une variable PI pour chaque cercle:
   final double PI = 3.14; // Possible
   // Une variable PI pour tous les cercles:
   final static double PI = 3.14; // Beaucoup mieux!!
   Cercle(double r) {
           //...
   }
}
```







# Méthodes statiques

- Similairement, si on ajoute `static` à une méthode :
  - On peut accéder à la méthode à travers un objet mais aussi *sans* objet

```
class A {
   void methode1() {
      System.out.println("M'ethode 1");
   }
   static void methode2() {
      System.out.println("M'ethode 2");
   }
}
class ExempleMethodeStatique {
   public static void main(String[] args) {
      A v = new A();
      v.methode1(); //OK
      v.methode2(); //OK
      A.methode2(); //OK, alternative
      A.methode1(); //Faux
   }
}
```







# Restrictions sur les méthodes statiques

Puisqu'une méthode statique peut être appelée avec ou sans objet :

- Le compilateur ne peut pas être sûr que l'objet `this` existe pendant l'exécution de la méthode
- Il ne peut donc pas admettre l'accès aux variables/méthodes d'instance (car elles dépendent de `this`)

Conclusion pour les accès dans la même classe :

- Une méthode statique peut *seulement* accéder à d'autres méthodes statiques et à des variables statiques

# Restrictions sur les méthodes statiques (2)

```
class A {
   int i;
   static int j;
   void methode1() {
      System.out.println(i); //OK
      System.out.println(j); //OK
      methode2(); //OK
   }
   static void methode2() {
      System.out.println(i); //Faux
      System.out.println(j); //OK
      methode1(); //Faux
      methode2(); //OK
                  //(sauf recursion infinie)
      A v = new A();
      v.methode1(); //OK
   }
}
```







# Utilité des méthodes statiques

- Méthodes qui ne sont pas liées à un objet
- Exemple :
  - Classe mettant à disposition des utilitaires mathématiques divers
  - La création d'un objet de type `MathUtils` est artificielle
  - La classe sert seulement à stocker des méthodes utilitaires

```
class MathUtils {
   final static double PI = 3.14;
   static double puissanceTrois(double d) {
      return d*d*d;
   }
}
```







# Utilité des méthodes statiques (2)

- Utilisation de la classe `MathUtils` :
  - Calculer $y = \pi \cdot x^3$ pour $x = 5.7$ ;
  - On peut accéder aux variables/méthodes statiques sans construire d'objet

```
class Calcul {
   public static void main(String[] args) {
      double x = 5.7;
      double y = MathUtils.PI *
         MathUtils.puissanceTrois(x);
      System.out.println(y);}
}
```

- Egalement possible mais inutile car l'objet `m` n'a pas d'intérêt pour le programme :

```
class Calcul {
   public static void main(String[] args) {
      MathUtils m = new MathUtils();
      double x = 5.7;
      double y = m.PI * m.puissanceTrois(x);
      System.out.println(y);
   }
}
```







# Méthodes et variables statiques

Eviter la prolifération de `static`

- On l'utilise seulement dans des situations très particulières
- définition d'une constante : `final static` + `variable` (situation très courante)
- utilisation d'une valeur commune `static`+ `variable` : (plus rare)
- méthodes utilitaires qu'il est artificiel de lier à un objet : `static` + `methode` : invocable sans objet (plus rare aussi)

Exemples de méthodes statiques :

- la méthode `main`
- `Math.sqrt`

# Levons le voile...

Nous sommes maintenant capables de comprendre trois mystères de Java :

1. Le format bizarre de certaines instructions :
   - `System.out.println()` par exemple !
2. Pourquoi les méthodes auxiliaires de la méthode `main` sont statiques mais pas les méthodes dans les classes
3. L'argument `String[]` de la méthode `main`







# `System.out.println()`

Analysons `System.out.println()` :

1. `System` :
   - Classe prédéfinie de Java
2. `out` :
   - Variable statique de la classe `System`
   - Il doit s'agir d'un objet car suivi d'un point
3. `println` :
   - Méthode de l'objet `out`

```
class System {
   //...
   static X out = new X(...);
   //...
}

class X {
   void println (...)
      {...}
    //..
}
```



Note : Dans l'API de Java `X` = la classe `PrintStream`





# Méthodes auxiliaires de `main`

La méthode `main` a une en-tête fixe :

```
public static void main(String[] args)
```

Puisque la méthode `main` est obligatoirement statique :

- Elle ne peut pas accéder à l'objet `this`
- Elle ne peut pas accéder à des variables/méthodes d'instance
- Elle peut seulement accéder à des variables/méthodes statiques

Sinon, la classe de la méthode `main` est comme n'importe quelle classe :

- Elle peut avoir des constructeurs, des méthodes et des variables
- Exemple ...







# Méthode `main` fausse

**Attention :** la méthode `main` ne peut accéder à des variable ou des méthode non-statiques

```
class MainFausse {
   int a;
   public static void main (String[] args) {
      a = 1;
      afficher();
   }
   void afficher () {
      System.out.println(a);
   }
}
```

Messages d'erreur :

```
MainFausse.java:4: non-static variable a cannot
be referenced from a static context
    a = 1;
    ^
MainFausse.java:5: non-static method afficher()
cannot be referenced from a static context
    afficher();
```

# Méthode `main` possible

Voici une alternative possible au code de l'exemple précédent :

- On ajoute `static` à la variable et à la méthode

```
class MainPossible {
   static int a;
   public static void main (String[] args) {
      a = 1; //OK
      afficher(); // OK
   }
   static void afficher () {
      System.out.println(a);
   }
}
```







# Méthode main recommandée

Une meilleure solution consiste cependant à :

- éviter les membres statiques (car ils vont à l'encontre des principes de POO)
- faire communiquer les méthodes à l'aide de paramètres

```
class MainRecommandee {
   public static void main (String[] args) {
      int a;
      a = 1;
      afficher(a);
   }
   static void afficher (int a) {
      System.out.println(a);
   }
}
```







# Méthode `main` avec objet

1. Seule les méthodes auxiliaires de la méthode `main` doivent être statiques
2. Les méthodes des autres classes sont bien sûr invocable depuis `main` à travers des objets

```
class MainObjet {
   public static void main(String[] args) {
      A v = new A();
      v.methode();
      afficher(v);
   }
   static void afficher (A v) {
      v.methode(); }
}
class A {
   A () {}
   void methode () {
      System.out.println("M'ethode");}
}
```







# L'argument de `main`

Comme nous le savons, la méthode `main` a l'entête obligatoire suivante :

```
public static void main(String[] args)
```

Composants :

1. `public` : méthode publiquement accessible
2. `static` : la méthode peut être appelée sans objet
3. `void` : la méthode ne retourne pas de valeur
4. `main` : nom de la méthode
5. `(String[] args)` : liste des paramètres
   - Tableau de chaînes de caractères

La méthode `main` est donc une méthode semblable à toutes les autres

Comment faire pour lui envoyer un argument ?

# Démarrage de la méthode `main`

Il y a deux façons de démarrer la méthode `main` :

1. Depuis un programme :
   - Plus rare mais faisable
   - Envoyer n'importe quel tableau `String[]` en paramètre
   - Appeler la méthode `main` de façon statique

**Attention** à la récursivité infinie !

```
class Programme1 {
   public static void main(String[] args) {
      System.out.println("Bienvenue");
      String[] ss = { "a", "b", "c" };
      Programme1.main(ss);
   }
}
```







# Démarrage de la méthode `main`

2. Depuis la ligne de commande :
   - Nécessaire pour démarrer un programme depuis la ligne de commande

```
java Programme1
```

Comment envoyer un tableau `String[]` à `main` depuis la ligne de commande ?

1. Ajouter les éléments du tableau à la ligne de commande
2. Séparer les éléments par des espaces
3. Guillemets pas nécessaires
4. Chaque valeur indiquée deviendra un élément du tableau `args`

Exemple : `java Programme2 a b c`







# Argument de la méthode `main`

Le paramètre `args` s'utilise comme n'importe quel tableau `String[]` :

```
class Programme2 {
   public static void main(String[] args) {
      System.out.println
         ("Il y a " + args.length +
          " 'el'ements dans le tableau args");
      for (int i = 0; i < args.length; i++) {
         System.out.println(args[i]); }
}
```

Exemples d'exécution :

```
java Programme2
Il y a 0 éléments dans le tableau args

java Programme2 a b c
Il y a 3 éléments dans le tableau args
a
b
c
```







# Argument de la méthode main

Les arguments de ligne de commande sont utiles pour communiquer des données rapidement à un programme
Exemple : addition de deux nombres

- Programme où l'utilisateur entre les valeurs au fur et à mesure :

```
java Additionner
Veuillez entrez le 1er nombre  : 2
Veuillez entrez le 2ème nombre : 3
La somme est 5
```

- Programme où l'utilisateur fournit les valeurs lors du démarrage :

```
java Additionner 2 3
La somme est 5
```

# Conversion `String` ⇒ `int`

Petit problème :

- Les arguments de `main` sont des `String`s
- On ne peut pas les utiliser tels quels dans des calculs

Code erroné :

```
class Additionner {
   public static void main (String[] args) {
      int somme = args[0] + // Erreur
          args[1];
      somme = (int)args[0] + // Erreur
          (int)args[1];
      //...
    }
}
```







# Conversion `String` ⇒ `int` (2)

Message d'erreur :

```
Incompatible types
found   : java.lang.String
required: int
     int somme = args[0] + args[1];

Inconvertible types
found   : java.lang.String
required: int
     int somme = (int)args[0] + (int)args[1];
```

- Conversion entre `String` et `int` ?
  - Problème informatique classique
  - Programmable avec les méthodes de la classe `String` : `length`, `charAt`
  - Méthodes `valueOf` des Wrapper classes (`Integer` etc.)







# Rappels

- Les avantages d'une hiérarchie de classes :
  - Réduit la duplication de code
  - Bon modèle de la réalité
- Un objet a plusieurs types :
  - Variable de type super-classe possible pour objets de type sous-classe
  - Transtypage explicite parfois nécessaire
- Hiérarchie plus claire/sûre :
  - Classe abstraite : jamais instanciée
  - Méthode abstraite : toujours implémentée par les sous-classes non-abstraites
  - Variable finale : jamais modifiée
  - Méthode finale : jamais masquée (redéfinie)
  - Classe finale : jamais héritée







# Jeu vidéo

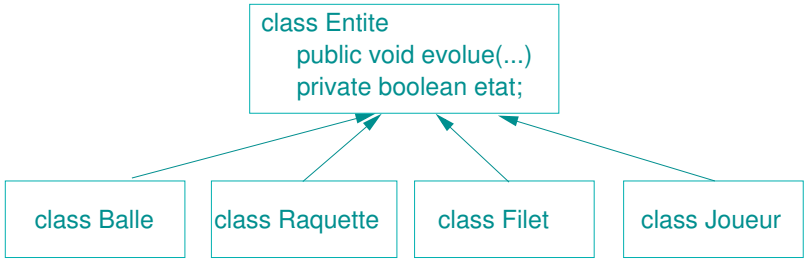
Supposons que l'on souhaite programmer un jeu mettant en scène les entités suivantes :

1. Balle
2. Raquette
3. Filet
4. Joueur

Chaque entité sera principalement dotée :

1. d'une méthode `evolue`, gérant l'évolution de l'entité dans le jeu ;
2. d'un état indiquant si l'entité participe au jeu ;

# Premiè ébauche de conception (1)









# Premiè ébauche de conception (2)

Si l'on analyse de plus près les besoins du jeu, on réalise que :

- Certaines entités doivent avoir une représentation graphique :
  - `Balle`, `Raquette`, `Filet`
- … et d'autres non
  - `Joueur`
- Certaines entités doivent être interactifs (on veut pouvoir les contrôler avec la souris par exemple) :
  - `Balle`, `Raquette`
- … et d'autres non
  - `Joueur`, `Filet`

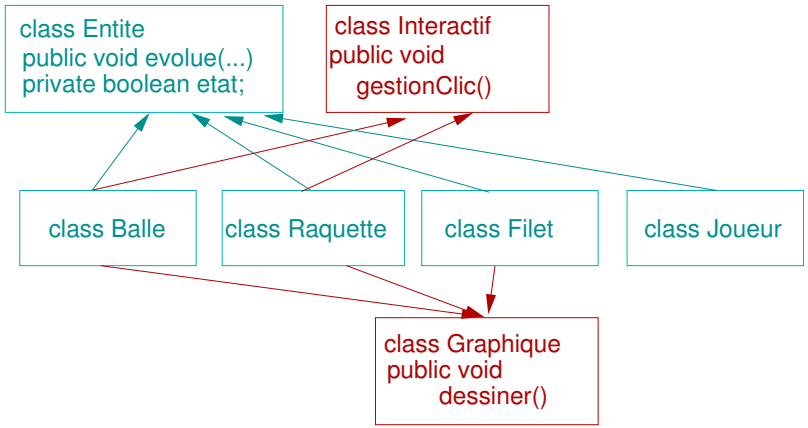
☞ Comment organiser tout cela ?






# Jeu vidéo impossible

Idéalement, il nous faudrait mettre en place une hiérarchie de classes telle que celle-ci :




Mais … **Java ne permet que l'héritage simple** : chaque sous-classe ne peut avoir qu'une seule classe parente directe !






# Héritage simple/multiple

- Pourquoi pas d'héritage multiple en Java ?
  - Parfois difficile à compiler et à comprendre
- Si une variable/méthode est déclarée dans plusieurs super-classes :
  - Ambiguïté : laquelle hériter ?

# Analyse

Mais en fait, que souhaitait-on utiliser de l'héritage multiple dans le cas de notre exemple de jeu vidéo ?

☞ **Le fait d'imposer à certaines classes de mettre en oeuvre des méthodes communes**

Par exemple :

- `Balle` et `Raquette` doivent avoir une méthode `gestionClic` ;
- mais `gestionClic` ne peut être une méthode de leur super-classe (car n'a pas de sens pour un `Joueur` par exemple).

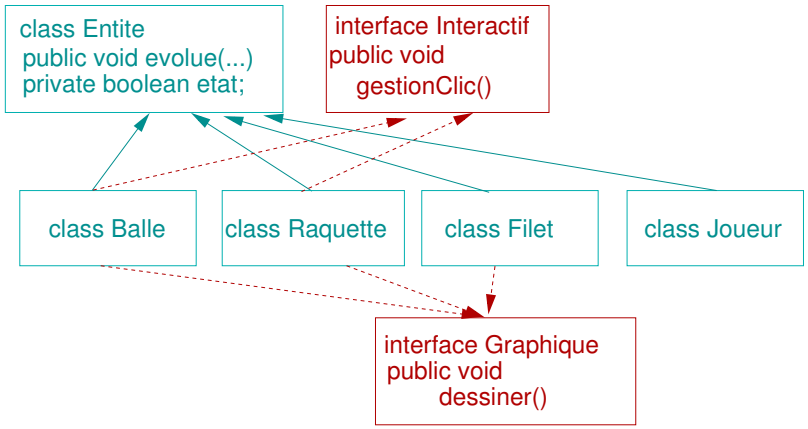
☞ Imposer un contenu commun à des sous-classes en dehors d'une relation d'héritage est le rôle joué par la notion d'`interface` en Java.







# Alternative possible de jeu vidéo




- Interface $\neq$ Classe
- Une interface permet d'imposer à certaines classes d'avoir un contenu particulier sans que ce contenu ne fasse partie d'une classe.







# Interfaces (1)

Une interface permet d'abord d'attribuer un type supplémentaire à une classe :

- Interface = type supplémentaire $\neq$ classe
- Solution Java pour imposer un contenu commun à des classes en dehors d'une relation d'héritage

Déclaration :

- Pas de méthode constructeur (car interface $\neq$ classe)
- Eventuellement des constantes
- Eventuellement des méthodes abstraites
- Rien d'autre

Exemple :

```
interface Graphique {
}
interface Interactif {
}
```







# Interfaces(2)

Attribution d'une interface à une classe :

```
class Filet extends Entite
    implements Graphique {
    ... code habituel
}
```

# Test de type avec une interface

Test du type d'un objet avec `instanceof` :

```
Filet filet = new Filet(...);
boolean b;
b = (filet instanceof Entite); // true
b = (filet instanceof Graphique); // true !
b = (filet instanceof Interactif); // false
```







# Variable de type interface

- Une interface n'est pas une classe :
  - Pas de méthode constructeur
  - Impossible de faire `new`
- Une interface attribue un type supplémentaire à une classe d'objets :
- Par contre, on peut :
  - Déclarer une variable de type interface
  - Y affecter un objet d'une classe qui implémente l'interface
  - Faire un transtypage explicite vers l'interface

```
Graphique graphique;
Balle balle = new Balle(..);
graphique = balle;
Entite entite = new Balle(..);
graphique = (Graphique)entite; // Cast
                               // indispensable
```







# Plusieurs interfaces

- Une classe peut implémenter plusieurs interfaces (mais étendre une seule classe)
  - Séparer les interfaces par des virgules

```
class Balle extends Entite
implements Graphique, Interactif {
... code habituel
}
```

- On peut déclarer une hiérarchie d'interfaces :
  - Mot-clé `extends`
  - La classe qui implémente une interface reçoit aussi le type des super-interfaces

```
interface Interactif { ..}
interface GerableParSouris
   extends Interactif { ..}
interface GerableParClavier
   extends Interactif { ..}
```







# Interface – Résumé (1)

Une interface est un moyen d'attribuer des composants communs à des classes non-liées par une relation d'héritage :

☞ Ses composants seront disponibles dans chaque classe qui l'implémente

Composants possibles (Java 7) :

1. Variables statiques finales (assez rare)
   ☞ Ambiguïté possible, nom unique exigé
2. Méthodes abstraites (courant)
   ☞ Chaque classe qui implémente l'interface sera obligée d'implémenter chaque méthode abstraite déclarée dans l'interface
   ☞ Une façon de garantir que certaines classes ont certaines méthodes, sans passer par des classes abstraites
   ☞ Aucune ambiguïté car sans instructions

# 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

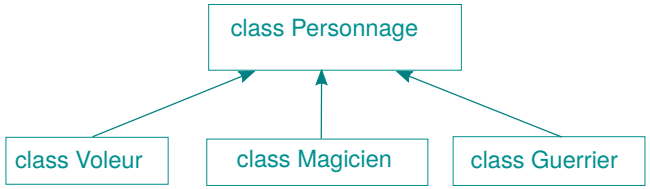






# Illustration

Reprenons notre exemple avec les personnages :




Supposons maintenant que certains personnages puissent chevaucher des montures.

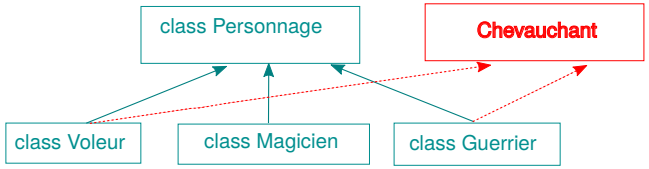






# Illustration

On pourrait donc imaginer la conception suivante :










# Illustration

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

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

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 definition 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 deplace comme un Dragonier
   public void seDeplace { Dragonier.super.seDeplace(); }
}
```

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

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

La classe `MageUltime` doit lever l'ambiguïté :

```
class MageUltime extends Magicien implements Dragonier,
                                              SeTeleporte
{
   // ou se deplace d'une  maniere 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.







# Interface – Résumé (2)

Nous avons vu que l'héritage permet de mettre en place une relation de type **"EST-UN"** ("IS-A") entre deux classes.

Lorsqu'une classe a pour attribut un objet d'une autre classe, il s'établit entre les deux classes une relation de type **"A-UN"** ("HAS-A"), moins forte que l'héritage (on parle de délégation).

Une interface permet d'assurer qu'une classe se conforme à un certain `protocole`.

Elle met en place une relation de type **"SE-COMPORTE-COMME"** ("BEHAVES-AS-A") : une `Balle` est-une entité du jeu, elle se-comporte-comme un objet graphique et comme un objet interactif.







# Interface – Utilité

Les interfaces permettent :

- de «voir» des objets sous le prisme de certaines de leurs fonctionnalités
- d'exposer les fonctionnalités plutôt que la façon de les mettre en oeuvre.

☞ Principe : « *Code to an interface not to an implementation* »

☞ Permet d'éviter d'être tributaire de détails d'implémentation

☞ Outil d'encapsulation

# Type énumérés : Directions

On désire modéliser les 4 directions que peut prendre une souris se déplaçant dans un labyrinthe (`DOWN`, `UP`, `LEFT`, `RIGHT`).

```
class Mouse {
   private  ??? direction;

   Mouse(??? direction) {
      this.direction = direction;
   }
}
```







# Solution 1 : chaînes/entiers

- **`String`** pour la direction
- des constantes pour les directions valides

```
class Mouse {
   public final static String DOWN = "DOWN";
    // ... idem pour UP, LEFT, RIGHT

   private String direction;

   public Mouse(String direction) {
           if (direction.equals(DOWN)) {
                   this.direction = DOWN;
                   //...
   }
}
```







# Solution 1 : chaînes de caractères

1. **`String`** est un type trop général qui inclut énormément de valeurs invalides ( `"vers l'infini et au delà"` pour la direction par exemple).

2. Ce type n'est pas assez "parlant" à la lecture du code.







# Solution 1 : chaînes de caractères

Il y a une **infinité** de valeurs de type **`String`** , mais seules **quatre** d'entre-elles sont des directions valides.

☞ Il faut faire constamment des **vérifications coûteuses** !

```
public Mouse(String direction) {
   if ( ! ( direction.equals(DOWN)
         ||  direction.equals(UP)
         ||  direction.equals(LEFT)
         ||  direction.equals(RIGHT)))
            // prochain cours:
        throw new IllegalArgumentException(...);
}
```

# Solution 2 : les objets à la rescousse

Une classe pour représenter une direction :

```
final class Direction {
   public final static Direction DOWN =
                        new Direction("DOWN");
   // ... idem pour UP, LEFT et RIGHT
   private final String nom;
   private Direction(String nom) {
      this.nom = nom;
   }
}

final class Mouse {
        private Direction direction;
        //...
}
```

Pourquoi un constructeur privé ?







# Solution 2 : les objets à la rescousse

Exemple d'instanciation d'une souris :

```
Mouse speedy = new Mouse(Direction.UP);
```







# Solution 2 : les objets à la rescousse

Avec cette solution orientée objet, le constructeur de **Mouse** par exemple devient plus simple

☞ plus besoin de vérifier ses arguments, qui sont valides par construction (si on ignore `null` !)

```
final class Mouse {
   private Direction direction;

   public Mouse(Direction direction) {
      this.direction = direction;
   }
// ... autres méthodes
}
```







# Solution 2 : les objets à la rescousse

Autre avantage de cette solution orientée-objets : il est possible d'ajouter des méthodes ou des champs aux classes

```
final class Direction {
    // ... comme avant
   public String frenchName() {
      if (this==LEFT) return "gauche";
      if (this==RIGHT) return "droite";
      if (this==UP) return "haut";
      return "bas";
   }
}
```

# Solution 3 : les énumérations

La solution orientée-objets est bonne, mais aussi fastidieuse à mettre en oeuvre.
Elle demande l'écriture de beaucoup de code répétitif.

☞ Pour éviter de devoir écrire ce code à chaque fois, Java offre la notion d'**énumération**






# Solution 3 : les énumérations

Au moyen des énumérations, le type `Direction` peut se définir simplement ainsi :

```
// fichier Direction.java
public enum  Direction {
        UP, DOWN, LEFT, RIGHT;
}
```

☞ Java se charge de traduire ces énumérations en classes encore plus complètes que celles que nous avons écrites.

Exemple de déclaration-initialisation :

```
Direction dir = Direction.UP;
```






# Solution 3 : les énumérations

Une énumération Java est traduite en une classe

Exactement comme dans notre version orientée-objets :

- chaque élément de l'énumération est traduit en une instance de cette classe
- ces instances sont des champs **statiques non modifiables** de la classe

Java définit de plus quelques méthodes utiles sur les énumérations et leurs éléments.






# Enumération : la méthode `values`

La méthode statique `values`, définie sur l'énumération elle-même, retourne un tableau contenant la totalité des éléments de l'énumération, dans l'ordre de déclaration.
Par exemple, pour l'énumération `Direction` suivante :

```
public enum Direction {
        UP, DOWN, LEFT, RIGHT;
}
```

l'appel `Direction.values()` retourne un tableau de taille 4 contenant `UP` à la position 0, `DOWN` à la position 1, et ainsi de suite.

# Enumération : la méthode `ordinal`

La méthode `ordinal` retourne un entier indiquant la position de l'élément dans l'énumération (à partir de zéro). Par exemple, étant donnée l'énumération :

```
public enum Direction {
        UP, DOWN, LEFT, RIGHT;
}
```

`UP.ordinal()` retourne 0, `DOWN.ordinal()` retourne 1, et ainsi de suite.
Les méthodes `ordinal` et `values` sont donc complémentaires
Les éléments des énumérations sont également équipés d'autres méthodes, parmi lesquelles une redéfinition de `toString` qui retourne le nom de l'élément, p.ex. `"UP"`






# Enumération : autres méthodes

Autres méthodes Les éléments des énumérations sont également équipés d'autres méthodes, parmi lesquelles :

- une redéfinition de `toString` qui retourne le nom de l'élément, p.ex. `"LEFT"` ;
- une redéfinition de `compareTo` de l'interface `Comparable`, automatiquement implémentée par toutes les énumérations, qui compare les éléments de l'énumération par ordre de définition






# Enumération : énoncé `switch`

Les éléments d'une énumération sont utilisables avec l'énoncé switch :

```
class Mouse {
   private Direction direction;
   //...
   public void move(Direction direction) {
      switch (direction) {
         case UP:    // ...
         case DOWN:  //...
         case LEFT : //...
         case RIGHT : // ...
         default:     // ...
      }
   }
}
```






# Enumération : ajout de membres

Ajout de membres Il est possible d'ajouter des membres, attributs ou méthodes, aux énumérations :

```
public enum Direction {
   UP, DOWN, LEFT, RIGHT;

   public Vector2D toVector() {
      switch (this) {
      case DOWN:
         return new Vector2D(0, 1);
      case UP:
         return new Vector2D(0, -1);
         //...
      }
   }
}
```

# Enumération : constructeurs

Il est aussi possible de définir un constructeur pour l'énumération, mais il ne peut alors être `public`.

```
enum Direction {
    UP("Haut"), DOWN("Bas"), LEFT("Gauche"),
    RIGHT("Droite");

    private final  String frenchName;

    public String frenchName()
    {
        return frenchName;
    }

    private Direction(String frenchName)
    {
        this.frenchName = frenchName;
    }
}
```







# Enumération : constructeurs (2)

Avec l'exemple précédent, le code suivant :

```
Direction dir = Direction.LEFT;
System.out.println(dir.frenchName());
```

afficherait : `Gauche`







# Enumération : imbrication

Supposons que l'on souhaite préciser que l'énumération `Direction` soit spécifique à une classe `Maze` modélisant notre labyrinthe.

Il est alors préférable d'imbriquer `Direction` dans `Maze`.

```
public class Maze
{

    public enum Direction {UP, DOWN, LEFT, RIGHT};

    public static void main(String[] args)
    {
        Maze.Direction mouseDirection = Maze.Direction.UP;
        System.out.println(mouseDirection);
    }
}
```







# Ce que j'ai appris aujourd'hui

- Que la notion d'héritage multiple n'existe pas en Java
- Que l'on peut cependant utiliser la notion d'**interface** pour attribuer des composants communs à des classes non-liées par une relation d'héritage
- Que les interfaces constituent un puissant outil d'encapsulation
- Ce qu'est le modificateur `static`
- Comment traiter l'argument de la méthode `main`

☞ Je dispose maintenant d'un bon bagage pour manipuler les concepts fondamentaux de la POO en java