Catégories de variables

Nos variables jusqu'à maintenant :

1. Variables d'instance (= attributs)
   - Décrivent les attributs d'un objet
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

Nouveauté :

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

Le modificateur `static`

- S'utilise pour les variables et les méthodes
- Si on ajoute `static` à la déclaration d'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

Variables d'instance et de classe

- 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

Variable statique : exemple

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

Pourquoi utiliser `static`?

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 une classe `Employe` le fait que 65 ans est l'âge officiel de départ à la retraite

Considérons les deux versions suivantes :

- avec une variable d'instance `ageRetraite`
- avec une variable statique `ageRetraite`

Classe `Employe1`

- Version avec une variable d'instance pour l'âge de la retraite :

```
class Employe1 {
  private String nom;
  private int ageRetraite;
  // ...
  public Employe1(String unNom,  int unAgeRetraite) {
    nom = unNom;
    ageRetraite = unAgeRetraite;
    // ... reste des initialisations
  }
  // ...
}
```

Utilisation de `Employe1`

- Version avec une variable d'instance pour l'âge officel de la retraite :

```
class Entreprise {
  public static void main(String[] args) {

    Employe1[] employes = new Employe1[350];
    employes[0] = new Employe1("Albus", 65);
    employes[1] = new Employe1("Oz", 65);
    // ...

    // La modification de l'âge de la retraite
    // nécessite un parcours du tableau car chaque
    // employé a sa propre version de la variable :
    for (int i = 0; i < employes.length; ++i) {
        employes1[i].ageRetraite = 67;
    }
  }
}
```

Classe `Employe2`

- Version avec une variable *statique* pour l'âge officiel de départ à la retraite

```
class Employe2 {
  // ...
  // Seule modification,  ageRetraite devient static:
  static int ageRetraite;
  // ...
}
class Entreprise {
  public static void main (String[] args) {
  //   ...
  // Remplissage du tableau comme avant.
  // Modification de l'âge de la retraite :
  // aucun parcours du tableau nécessaire
  Employe2.ageRetraite = 67;
  employes[0].ageRetraite = 67;   // alternative
  employes[250].ageRetraite = 67; // alternative
  }
}
```

Constantes : `final` et `static`

Pour les constantes communes à toutes les instances d'une classe :

- inutile de stocker une valeur pour chaque objet de la classe
- les déclarer en `final static`

```
class Planete {
   // G = constante gravitationnelle
   // Une variable G pour chaque planète :
   // Possible
   private final double G = 6.674E-8;

   // Une variable G pour toutes les planètes :
   // BEAUCOUP MIEUX !
   private final static double G = 6.674E-8;

   // ...
}
```

Levons le voile...

Nous sommes maintenant capables de comprendre le format bizarre de certaines instructions :

- `System.out.println()` par exemple!

`System.out.println()`

Analysons `System.out.println()` :

- `System` : classe prédéfinie de Java
- `out` :
  - Variable statique de la classe `System`
  - Il doit s'agir d'un objet car suivi d'un point
- `println` : méthode de l'objet `out`

```
class System {
   //...
   static PrintStream out = new PrintStream(...);
   //...
}
class PrintStream {
   void println (...)
      {...}
    //..
}
```

Méthodes statiques

Similairement, si l'on ajoute `static` à une méthode on peut alors y accéder *sans aucun* objet

```
class A {
   static void methode1() {
      System.out.println("Méthode 1");
   }
   void methode2() {
      System.out.println("Méthode 2");
   }
}
class ExempleMethodeStatique {
   public static void main(String[] args) {
      A.methode1(); // OK
      A.methode2(); // Non !
      A v = new A();
      v.methode1(); // OK, alternative
      v.methode2(); // OK (comme d'habitude)
   }
}
```

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 {
   public final static double PI = 3.14159265358979323846;
   public static double auCube(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.auCube(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 : attribut `final static`
  (situation très courante)
- utilisation d'une valeur commune : attribut `static`
  (plus rare)
- méthodes utilitaires qu'il est artificiel de lier à un objet :
  méthode `static`, *invocable sans objet*
  (plus rare aussi)

Exemples de méthodes statiques :

- `Math.sqrt`
- la méthode `main`

Méthodes auxiliaires de `main`

Nous comprennons maintenant pourquoi les méthodes auxiliaires de la méthode `main` sont statiques
(mais pas les méthodes dans les classes)

La méthode `main` a un 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

En dehors de cela, la classe de la méthode `main` est comme n'importe quelle classe.

Elle peut avoir des constructeurs, des méthodes et des variables

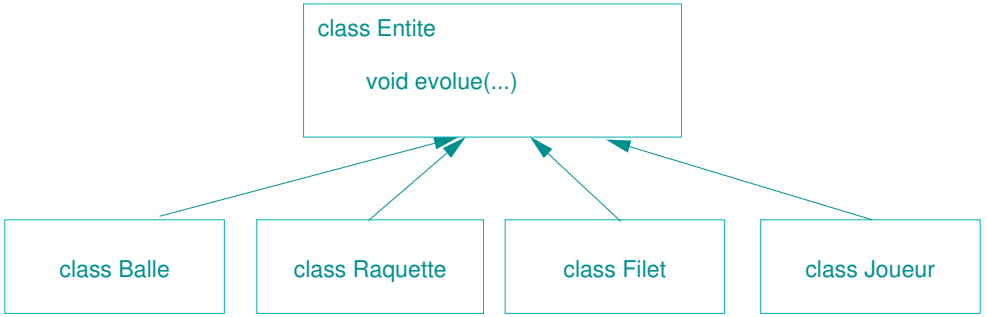
Encore un jeu ...

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 d'une méthode `evolue`, gérant l'évolution de l'entité dans le jeu.

Première ébauche de conception (1)



Première ébauche de conception (2)

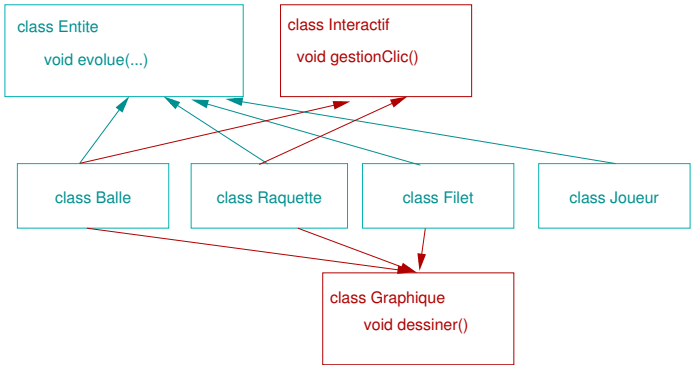
Si l'on analyse de plus près les besoins du jeu, on réalise que :

- certaines entités doivent avoir une représentation graphique (`Balle`, `Raquette`, `Filet`)
- ... et d'autres non (`Joueur`)
- certaines entités doivent être interactives (on veut par exemple pouvoir les contrôler avec la souris) : `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 comprendre (quel sens donner ?), y compris pour le compilateur

    (par exemple si une sous-sous-classe hérite d'une super-super-classe par différents chemins)
- Si une variable/méthode est déclarée dans plusieurs super-classes
  - Ambiguïté : laquelle utiliser, comment y accéder ?

Analyse

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

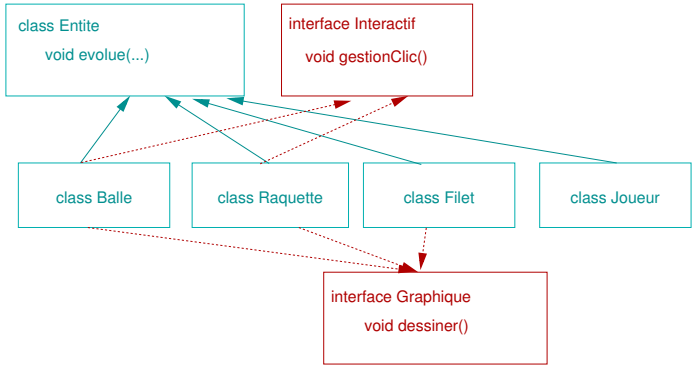
☞ **Le fait d'imposer à certaines classes de mettre en oeuvre des méthodes communes**

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)

Syntaxe :

`interface UneInterface { constantes ou méthodes abstraites }`

Exemple :

```
interface Graphique {
 void dessiner();
}
interface Interactif {
 void gestionClic();
}
```

Il ne peut y avoir de constructeur dans une interface

☞ Impossible de faire `new` !

Interfaces (2)

Attribution d'une interface à une classe :

Syntaxe :

```
class UneClasse implements Interface1, ... , InterfaceN
{ ... }
```

Exemple :

```
class Filet extends Entite implements Graphique {
 public void dessiner() { ... }
}
```

Variable de type interface

Une interface attribue un type supplémentaire à une classe d'objets, on peut donc :

- Déclarer une variable de type interface
- Y affecter un objet d'une classe qui implémente l'interface
- (éventuellement, faire un transtypage explicite vers l'interface)

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

Plusieurs interfaces

Une classe peut implémenter plusieurs interfaces (mais étendre une seule classe)

- Séparer les interfaces par des virgules

Exemple :

```
class Balle extends Entite implements Graphique, Interactif {
 //  code de la classe
}
```

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 :

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 si elle veut pouvoir être instanciée
   - ☞ Une façon de garantir que certaines classes ont certaines méthodes, sans passer par des classes abstraites
   - ☞ Aucune ambiguïté car sans instructions

Interface – Résumé (2)

Nous avons vu que l'héritage permet de mettre en place une relation de type **« est-un »** 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 »** 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 »** : une `Balle` « est-une » entité du jeu, elle « se-comporte-comme » un objet graphique et comme un objet interactif.