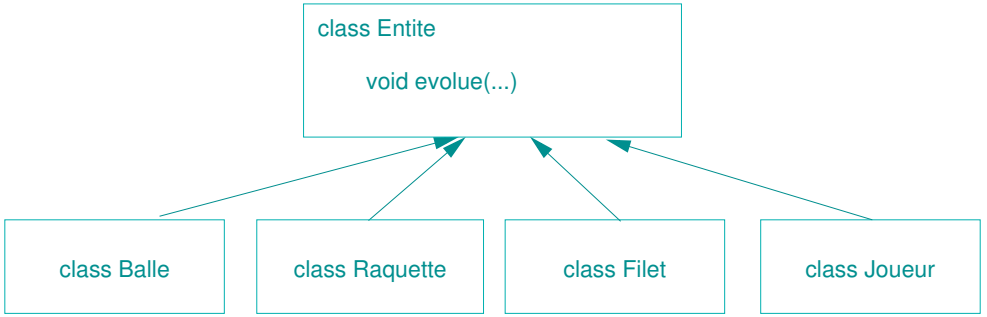
Encore un jeu ...

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)

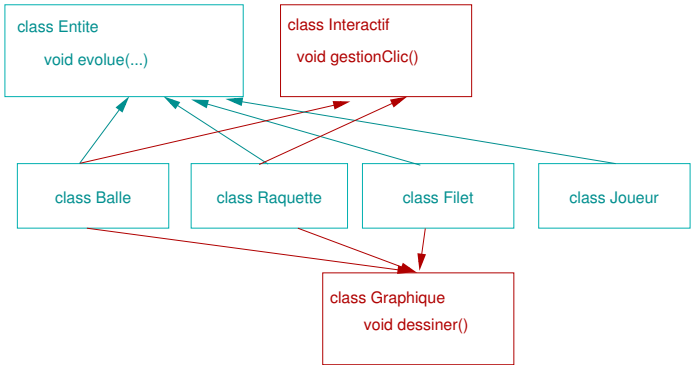
Si l'on analyse de plus près les besoins du jeu, on réalise que :

- 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 ?

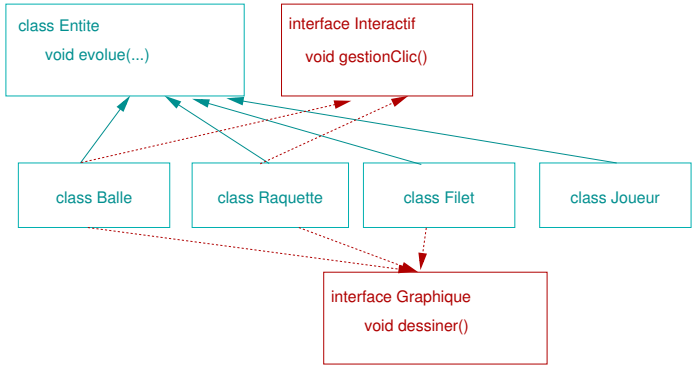
☞ **Le fait d'imposer à certaines classes de mettre en oeuvre des méthodes communes**

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.