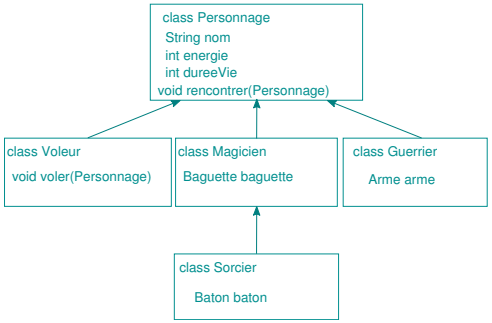
Polymorphisme (universel d'inclusion)

En POO, le **polymorphisme** (universel d'inclusion) est le fait que les instances d'une sous-classe, lesquelles sont *substituables* aux instances des classes de leur ascendance (en argument d'une méthode, lors d'affectations), **gardent leurs propriétés propres**.

☞ Le choix des méthodes à invoquer se fait *lors de l'exécution du programme* en fonction de la *nature réelle des instances* concernées.

La mise en œuvre se fait au travers de :

- l'*héritage* (hiérarchies de classes) ;
- la **résolution dynamique des liens**.




```
Personnage p1, p2;
// ...
p1.rencontrer(p2);
```

Résolution des liens (rappel)

```
class Personnage {
  // ...
  public
   void rencontrer(Personnage p) {
    System.out.print("Bonjour !");
  }
}

class Guerrier {
  // ...
  public
   void rencontrer(Personnage p) {
    System.out.print("Boum !");
  }
}
```

```
class Rencontre
{
 public static void main(...) {
  Guerrier g = new Guerrier(...);
  Voleur   v = new Voleur(...);
  uneRencontre(g, v);
 }
 static void uneRencontre(Personnage a,
                          Personnage b) {
  System.out.print(a.getNom());
  System.out.print(" rencontre ");
  System.out.print(b.getNom() + " : " );
  a.rencontrer(b);
  }
}
```

Méthodes abstraites : problème

Au sommet d'une hiérarchie de classe, il n'est pas toujours possible de :

- donner une définition générale de certaines méthodes, *compatibles avec toutes les sous-classes*,
- ...même si l'on sait que toutes ces sous-classes vont effectivement implémenter ces méthodes

Besoin de méthodes abstraites : exemple

Exemple :

```
class FigureFermee  {
  //  ...

  // difficile à définir à ce niveau !..
  public double surface(...) { ??? }

  // ...pourtant la méthode suivante en aurait besoin !
  public double volume(double hauteur) {
   return hauteur * surface();
  }
}
```

Définir `surface` de façon arbitraire sachant qu'on va la redéfinir plus tard n'est pas une bonne solution (source d'erreurs) !

☞ *Solution* : déclarer la méthode `surface` comme **abstraite**

Besoin de méthodes abstraites : autre exemple (1)

Plusieurs équipes collaborent à la création d'un jeu.

Une équipe prend en charge les classes de base suivantes :

- `Jeu`:
  - Classe pour gérer le jeu
  - Se contente ici d'afficher les personnages
- `Personnage`:
  - Classe de base pour les personnages

Une autre équipe ajoutera des sous-classes de personnages spécifiques.

Besoin de méthodes abstraites : autre exemple (2)

La classe `Jeu` développée par la première équipe :

☞ gère un tableau de personnages et les affiche

```
class Jeu {
   private ArrayList<Personnage> persos;
  // ...
   public void afficher() {
      for (Personnage unPerso : persos) {
        unPerso.afficher();
      }
  }
   public void ajouterPersonnage(...) { // ...
  }
  // ...
}
```

Besoin de méthodes abstraites : autre exemple (3)

Si l'on ne met aucune méthode `afficher` dans `Personnage`, la classe `Jeu` ne compile pas :

```
class Jeu {
  // ...
  public void afficher() {
    for (Personnage unPerso : persos) {
      unPerso.afficher(); // ERREUR !
    }
  }
  // ...
}
```

☞ On **doit** donc mettre une méthode `afficher` dans la classe `Personnage`...

De plus, on aimerait :

- imposer aux sous-classes (`Guerrier`, ...) d'avoir leur méthode `afficher` spécifique

Besoin de méthodes abstraites : autre exemple (4)

☞ On **doit** donc mettre une méthode `afficher` dans la classe `Personnage`...

...mais comment faire ?

```
class Personnage {
   private String nom;
   private int energie;
   // ...
   // constructeurs
   // ...
   public void afficher() {
      // Tous les personnages doivent pouvoir s'afficher !...
      // ...mais comment ???
   }
}
```

Et comment *imposer* que la méthode `afficher` soit redéfinie dans les sous-classes ?

Besoin de méthodes abstraites : autre exemple (5)

Première « solution » :

ajouter une méthode quelconque définie arbitrairement :

```
class Personnage {
  // ...
  // On n'affiche rien : corps de la méthode vide
  public void afficher()  { }
  // ...
}
```

C'est une **très mauvaise idée**

- ☞ Mauvais modèle de la réalité
  (affichage incorrect si une sous-classe ne redéfinit pas la méthode : personnages fantômes !)
- ☞ Cette solution n'impose pas que la méthode `afficher` soit redéfinie

Besoin de méthodes abstraites : autre exemple (6)

Bonne solution :

Signaler que la méthode doit exister dans *chaque* sous-classe sans qu'il soit nécessaire de la définir dans la super-classe

- ☞ Déclarer la méthode comme **abstraite**

Méthodes abstraites : définition et syntaxe

Une méthode *abstraite*, est *incomplètement spécifiée* :

- elle n'a *pas de définition* dans la classe où elle est introduite (pas de corps)
- elle sert à imposer aux sous-classes (non abstraites) qu'elles **doivent définir** la méthode abstraite héritée
- elle doit être contenue dans une classe abstraite

Syntaxe :

```
abstract Type nom_methode(liste d'arguments);
```

Exemple :

```
abstract class Personnage {
  // ...
  public abstract void afficher();
  // ...
}
```

Méthodes abstraites : autre exemple

```
abstract class FigureFermee {
 public abstract double surface();
 public abstract double perimetre();

 // On peut utiliser une méthode abstraite :
 public double volume(double hauteur) {
  return hauteur * surface();
 }
}
```

Classes abstraites

Une classe abstraite est une classe désignée comme telle au moyen du mot réservé `abstract`.

- Elle *ne peut être instanciée*
- Ses sous-classes *restent abstraites* tant qu'elles ne fournissent pas les définitions de *toutes les méthodes abstraites* dont elles héritent.

Un exemple « concret »...

Classes abstraites : exemple

Une autre équipe crée la sous-classe `Guerrier` de `Personnage` et veut l'utiliser :

```
Jeu jeu = new Jeu();
jeu.ajouterPersonnage(new Guerrier(...));
```

S'ils ont oublié de définir la méthode `afficher`, le code ci-dessus génère une erreur de compilation car on ne peut pas créer d'instance de `Guerrier` :

```
Guerrier is abstract; cannot be instantiated
```

Classes abstraites : autre exemple

```
class Cercle extends FigureFermee {
   private double rayon;

   public double surface() {
      return Math.PI * rayon * rayon;
   }
   public double perimetre() {
      return 2.0 * Math.PI * rayon;
   }
}
```

`Cercle` n'est pas une classe abstraite

```
class Polygone extends FigureFermee {
   private ArrayList<Double> cotes;

   public  double perimetre() {
    double p = 0.0;
    for (Double cote : cotes) {
      p += cote;
    }
    return p;
  }
}
```

`Polygone` *reste* par contre une classe *abstraite*