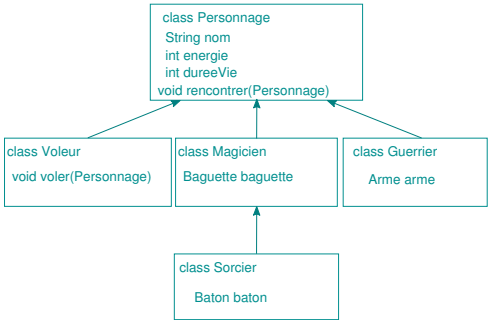
Polymorphisme (universel d'inclusion)

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*

Constructeurs et polymorphisme

Un constructeur est une méthode spécifiquement dédiée à la construction de l'**instance courante** d'une classe, il n'est pas prévu qu'il ait un comportement polymorphique.

Il est cependant possible d'invoquer une méthode polymorphique dans le corps d'un constructeur

☞ Ceci est cependant déconseillé : la méthode agit sur un objet qui n'est peut-être alors que partiellement initialisé !

Un exemple ...

Constructeurs et polymorphisme (2)

```
abstract class A
{
   public abstract void m();
   public A() {
      m(); // méthode invocable de manière polymorphique
   }
}
class B extends A {
   private int b;
   public B() {
      b = 1; // A() est invoquée implicitement juste avant
   }
   public void m() { // définition de m pour la classe B
      System.out.println("b vaut : " + b);
   }
}
// .... dans le programme principal :
B b = new B();
```

☞ affiche : `b vaut 0`

La super-classe `Object`

Il existe en Java une super-classe commune à toutes les classes : la classe `Object` qui constitue **le sommet de la hiérarchie**

Toute classe que vous définissez, si elle n'hérite d'aucune classe explicitement, dérive de `Object`

Il est donc possible d'affecter une instance de n'importe quelle classe à une variable de type `Object` :

```
Object v = new UneClasse (...); // OK
```

La super-classe `Object` (2)

La classe `Object` définit, entre autres, les méthodes :

- `toString` : qui affiche juste une représentation de l'adresse de l'objet;
- `equals` : qui fait une comparaison au moyen de `==` (comparaison des références);
- `clone` : qui permet de copier l'instance courante

Dans la plupart des cas, ces définitions par défaut ne sont pas satisfaisantes quand vous définissez vos propres classes

☞ Vous êtes amenés à les **redéfinir** pour permettre un affichage, une comparaison ou une copie corrects de vos objets

☞ c'est ce que nous avons fait dans une séquence précédente avec `toString` !

☞ La classe `String` aussi par exemple redéfinit ces méthodes

Exemple : redéfinition de `equals` héritée de `Object`

L'entête proposée pour la méthode `equals` dans une séquence précédente était :

```
public boolean equals(UneClasse arg)
```

or l'entête de la méthode `equals` dans `Object` est :

```
public boolean equals(Object arg)
```

☞ Nos définitions de `equals` constituaient jusqu'ici des **surcharges** et non pas des **redéfinitions** de la méthode `equals` de `Object` !

Dans la plupart des cas, utiliser une surcharge fonctionne sans problème, mais il est recommandé de **toujours procéder par redéfinition**.

Surcharge, redéfinition (rappels)

On redéfinit (« override ») une méthode d'instance si les paramètres et leurs types sont identiques et les types de retour compatibles :

```
public boolean equals(Object o)
```

Si c'est le même nom de méthode seulement, on surcharge (« overload ») :

```
public boolean equals(UneClasse c)
```

Redéfinition usuelle de `equals`

Attention ! si l'on redéfinit `equals` pour la classe `Rectangle`, on doit pouvoir comparer un `Rectangle` avec n'importe quel autre objet : `unRectangle.equals("toto")` devrait retourner `false`.

```
class Rectangle {
  //...
  public boolean equals(Object autreObjet) {
   if (autreObjet == null)
    { return false; }
   else {
    if (autreObjet.getClass() != getClass())
     { return false; }
    else {
     Rectangle r = (Rectangle)autreObjet;
     return (largeur == r.largeur &&
             hauteur == r.hauteur);
    }
   }
  }
}
```

Le modificateur `final`

Ce modificateur permet d'indiquer les éléments du programme qui ne doivent pas être modifiés/redéfinis/étendus

- Possible pour les classes, méthodes, attributs, variables
- Utile surtout pour les variables
- Moins courant pour les méthodes et les classes

Méthodes finales

Si l'on ajoute `final` à une méthode :

☞ Impossible de la redéfinir dans une sous-classe

Exemple : on aimerait toujours appliquer la méthode `vieillir` de `Personnage`

```
class Personnage
{
  //...
   final void vieillir() {
      --dureeVie;
   }
}
```

☞ message d'erreur du compilateur si la classe `Sorcier` essaie de redéfinir la méthode `vieillir`

Classes finales

Si l'on ajoute `final` à une classe :

☞ Impossible d'étendre la classe par une sous-classe

Exemple : on aimerait que la classe `Sorcier` n'ait jamais de sous-classe

```
final class Sorcier extends Magicien {
   //...
}
```

```
class MageNoir extends Sorcier { ..}
// illicite!!
```

Classes et méthode finales

Les méthodes et classes finales peuvent être à priori « agaçantes » :

- Exemple: la classe prédéfinie `String` est finale
- Aucune possibilité de définir

  `class MyString extends String`

  afin d'améliorer certaines méthodes par redéfinition!!

  ☞ Mais, permet de fixer une fois pour toute le comportement d'une classe ou méthode

Variables finales et objets référencés (1)

Si l'on ajoute `final` à une variable d'instance, une variable locale ou un paramètre :

- ☞ il devient impossible de lui affecter une valeur plus d'une fois

Un attribut `final` peut être initialisé dans le constructeur mais ne doit plus être modifié par la suite

Attention : `final` empêche l'affectation d'une nouvelle valeur à une variable, mais n'empêche pas de modifier l'éventuel objet référencé par cette variable :

Un exemple ...

Variables finales et objets référencés (2)

```
class Conteneur {
   private int valeur;
   public void setValeur(int val) { valeur = val; }
}

class Test {
   public static void main(String[] args) {
      Conteneur c = new Conteneur();
      c.setValeur(42);
      modifier(c);
   }
   static void modifier(final Conteneur c) {
      c.setValeur(-1); // modifie l'objet référencé !!
      //c = new Conteneur(); //FAUX
   }
}
```