Java et OOP
Héritage et polymorphisme

Transtypage révisité

- **Transtypage implicit** : on peut instancier une variable déclarée appartenant à une classe CLS avec un objet d'une classe qui dérive de CLS.

```
class Personne{... }
class Etudiant extends Personne{... }
class Main {
   public static void main(String[] argv){
       Etudiant e1 = new Etudiant(...);
       Personne p1 = e1;
   }
}
```

- **Transtypage explicit** : on peut instancier une variable déclarée appartenant à une classe CLS avec un objet d'une classe qui est un ancêtre de CLS.

```
       Personne p2 = new Personne();
       Etudiant e2 = (Etudiant) p2;
```

Transtypage révisité

- Par contre, on ne peut pas instancier des variables qui **ne sont pas** en relation d'héritage ou de descendence !

```
class Personne{... }
class Mammifere{... }
class Main {
   public static void main(String[] argv){
       Personne p = new Personne(...);
       Mammifere m = p;   // ERREUR DE COMPILATION
                          // "incompatible types"
   }
}
```

- `Personne` et `Mammifere` sont des "freres" ("soeurs" ?), en tant que fils (filles ?) d'`Object` ! N'empêche que cela soit une erreur !
- Rappel : en C, on peut affecter à une variable `<type1> *p` la valeur d'une variable `<type2> *p` si on fait un *cast*, même si les types n'ont rien à voir l'un avec l'autre.
- Rappel 2 : En Java, une déclaration de variable objet (comme `Personne p`) est en fait une déclaration de **référence à un objet de classe donnée**.

Transtypage révisité

Mais de quoi le compilateur se plaint-il ?

- Le compilateur veut *décider* les types de chaque attribut.
- **Types compatibles** = classes en relation de dérivation.
- Il fait pas confiance au programmeur pour sa géstion de la mémoire :

```
class Personne{ int nb_pieds; }
class Mammifere{ int nb_pieds; }
class Main {
   public static void main(String[] argv){
       Personne p = new Personne(...);
       Mammifere m = p;   // TOUJOURS ERREUR DE COMPILATION
                          // "incompatible types"
   }
}
```

  Donc même si les classes ont (intuitivement) la même taille !

- Plein d'erreurs de programmation dues à la confusion entre types de pointeurs en C !
- Le typage fort rend le programme indépendant de l'architecture – on ne veut pas savoir comment (dans quel ordre) les attributs sont stockés.

Méthode `clone` et transtypage

- Rappel : pour copier des objets il faut utiliser `Clone`.
- En fait, il faut la **`redéfinir`** dans chaque classe !
- Oui, mais son prototype est :

```
class Object{
 ....
 public Object clone(){...}
}
```

- On ne peut pas déclarer dans notre classe `Personne` une méthode

```
 Personne clone()
```

car cela serait contraire à la règle qui intérdit de surcharger des méthodes sur le seul type de retour !

Méthode `clone` et transtypage

- Solution (seule...) : on déclare

```
class Personne {
 String prenom, nom;
 Personne(String p, String n){
    prenom = new String(p);
    nom = new String(n);
 }
 public Object clone(){
   return new Personne(prenom,nom);
 }
}
```

et on croise les doigts...

```
 Personne p = new Personne("X","Y");
 Personne q = (Personne) p.clone();
```

- Et ça marche ! Notre personne a été bien clonée !

Transtypez explicitement, mais pas n'importe comment !

- Le downcasting permet d'avertir le programmeur sur des erreurs possibles, dues au manque d'information : le type à transtyper est plus pauvre que celui dans lequel on transtype.

```
class Personne {
 String prenom, nom;
}
class Etudiant extends Personne {
 String filiere;
}
```

- Une `Personne` a moins d'attributs qu'un `Etudiant`.
- Si vous vous entêtez à transtyper des `Personne`s en `Etudiant`s,

```
class Main{
 public static void main(String[] argv){
   Personne p = new Personne("Jean","X");
   Etudiant e = (Etudiant)p;
 }
}
```

faites gafe à ne pas chercher d'attribut `filiere` dans votre variable ! Ça n'existe pas, le cast ne le fait apparaître miraculeusement !

Listes hétérogènes

- On veut maintenant traiter des listes hétérogènes de personnes : étudiants, profs, personnes banales...
- On voudrait mettre tous les êtres à traiter dans une seule liste.

```
Personne p[] = new Personne[3];
p[0] = new Etudiant("Jean","X","deug");
p[1] = new Professeur("Catalin","Dima","MdC");
p[2] = new Personne("Qqun","Bizarre");
```

Listes hétérogènes

- Déjà, on doit prévoir une hiérarchie de classes pour que le transtypage explicite soit autorisé !

```
class Personne {
 String nom, prenom;
 Personne (String p, String n){
   prenom = new String(p);
   nom = new String(n);
 }
 void affiche(){
   System.out.println("nom = " + prenom + " " + nom);
 }
}
class Etudiant extends Personne {
 String filiere;
 Etudiant (String p, String n, String f){
   super(p,n);
   filiere = new String(f);
 }
 void affiche(){
   super.affiche();
   System.out.println("filiere = " + filiere);
 }
}
```

```
class Professeur extends Personne {
 String cours;
 Professeur (String p, String n, String c){
   super(p,n);
   cours = new String(t);
 }
 void affiche(){
   super.affiche();
   System.out.println("cours = " + cours);
 }
}
```

Listes hétérogènes

- Mais peut-on les traiter de manière uniforme ?

```
class Main {
 public static void main(String[] argv){
    Personne p[] = new Personne[3];
    p[0] = new Etudiant("Jean","X","deug");
    p[1] = new Professeur("Catalin","Dima","java");
    p[2] = new Personne("Qqun","Bizarre");
    for (int i=0; i<3; i++)
        p[i].affiche();
 }
}
```

- Quelle méthode `affiche()` sera appelée ? se rappeler que `p[i]` est déclaré en tant que `Personne`.
- On va alors afficher que les noms et les prénoms – c'est à dire, en utilisant la méthode `affiche()` de la classe `Personne` ? Ça serait embêtant !

Polymorphisme par "late binding"

- **Non** ! Ce sera la méthode `affiche()` du type réel de chaque objet !

```
nom = Jean X
filiere = deug
nom = Catalin Dima
cours = java
nom = Qqun Bizarre
```

- Pour les *méthodes*, Java utilise le **late binding** : le choix de la méthode à appliquer dans chaque cas est laissé au moment de *l'exécution* !
- Le principe implémenté est celui du **polymorphisme** = une même "écriture" correspond aux différents appels de méthode.

"Late binding"

- Parfois le compilateur ne peut pas déduire à l'avance quel sera le type de l'objet qui appelle chaque méthode :

```
class Main {
 public static void fait_qq_chose(Personne p){
    p.affiche();
 }
 public static void main(String[] argv){
    Personne p[] = new Personne[3];
    p[0] = new Etudiant("Jean","X","deug");
    p[1] = new Professeur("Catalin","Dima","java");
    p[2] = new Personne("Qqun","Bizarre");
    for (int i=0; i<3; i++)
        fait_qq_chose(p[i]);
 }
}
```

- On a renvoyé les affichages dans une fonction.
- A l'intérieur de cette fonction le paramètre reçu ne peut pas être autre que `Personne` !
- Du coup, le compilateur est en dilemme : il ne peut pas décider quel code associer à l'appel `p.affiche()` !
- Le "late binding" c'est son seul moyen d'échapper à son dilemme !
- Il rajoute de l'information pour que la MVJ (Machine Virtuelle Java) décide, au moment de l'exécution, quel est le type réel de l'objet caché derrière le nom de `p`.

Polymorphisme et type de retour

- On rajoute (encore) une méthode bidon dans notre classe `Personne` :

```
Personne blabla(){
   return this;
}
```

- Elle ne fait que renvoyer la référence sur l'objet qui l'a appelé.
- On ne la redéfinit pas dans les classes dérivées ! Donc `Etudiant` et `Professeur` vont posseder une méthode `blabla` qui renvoie une référence à une `Personne` !
- On appelle cette méthode pour un `Etudiant` et on affiche l'objet retourné :

```
Etudiant p = new Etudiant("a","b","info");
p.blabla().affiche();
```

- C'est l'`affiche()` de l'`Etudiant` qui est appelé !
- Surpris ? ! Non ! c'est le late binding toujours !

Bloquer le polymorpisme

Toujours le mot-clé `final` :

- Une méthode déclarée `final` ne peut pas être redéfinie.
- Si la classe n'est pas elle aussi finale, on peut tout de même dériver d'elle des nouvelles classes
- Du coup, on ne peut pas faire des appels polymorphes sur notre méthode finale, pour ces classes dérivées !

```
class Personne {
 String nom, prenom;
 Personne (String p, String n){
   prenom = new String(p);
   nom = new String(n);
 }
 final void affiche(){
   System.out.println("nom = " + prenom + " " + nom);
 }
}
class Etudiant extends Personne {
 String filiere;
 Etudiant (String p, String n, String f){
   super(p,n);
   filiere = new String(f);
 }
 // void affiche()    -- INTERDIT !
 // Si on veut une methode affiche, il faut la surcharger !
}
```

```
class Main {
 public static void main(String[] argv){
    Personne p = new Etudiant("X","Y","deug");
    p.affiche() // c'est l'affiche de Personne, donc pas de filiere
 }
}
```

Classe `Class`

- On se retrouve plus dans les appels polymorphes... on voudrait tout de même savoir quel est le vrai type (donc classe !) de l'objet qui est passé en paramètre.
- On a la méthode `getClass` de la classe `Object` qui nous dit cela !
  Elle renvoie un objet de type `Class` qui correspond à la (vraie) classe de l'objet qui l'appelle...
- Bon, allons voir un exemple :

```
class Main {
 public static void main(String[] argv){
    Personne p[] = new Personne[3];
    p[0] = new Etudiant("Jean","X","deug");
    p[1] = new Professeur("Catalin","Dima","java");
    p[2] = new Personne("Qqun","Bizarre");
    for (int i=0; i<3; i++) fait_qq_chose(p[i]);
 }
 public static void fait_qq_chose(Personne p){
    p.affiche();
    System.out.println("classe = " + p.getClass());
 }
}
```

```
nom = Jean X
filiere = deug
classe = class Etudiant
nom = Catalin Dima
cours = java
classe = class Professeur
nom = Qqun Bizarre
classe = class Personne
```

Utilité du polymorphisme

- Le code des différents types impliques est facile.
  - Sans polymorphisme, on aurait besoin du code special dans chaque méthode pour retrouver le type concret de l'objet appelant.
- L'ajout d'une nouvelle classe dérivée est simple – on a besoin que de redéfinir les méthodes polymorphes, on doit pas changer aussi les anciennes méthodes en rajoutant du code.
- On peut pousser même plus loin l'abstraction des classes, en définissant des classes *abstraites* qui réunissent des caractéristiques des classes dérivées qui permettent le traitement unitaire de celles-ci.