Séquence 11

Typage avec de l'héritage

Tant qu'on n'utilise pas l'héritage, on peut considérer qu'un objet a un seul type : la classe qui a servi à le créer lors de l'instaciation avec `new`.

Avec l'héritage, il faut renoncer à cette vision très simple des types des objets. Un objet a plusieurs types et on peut le voir de différentes façons.

1 Un exemple avec de l'héritage

Nous allons présenter un exemple de programme ayant plusieurs classes avec des relations d'héritage. Il s'agit de représenter des mots de la langue française avec différentes caractéristiques (genre, nombre, personne selon le cas).

Nous aurons une première classe qui représente les mots invariables, puis des sous-classes qui permettent d'ajouter des informations de genre, de nombre ou de personne.

```
public class Mot{
    protected String formeMot;
    protected String categorie;
    public Mot(String fm, String cat){
        formeMot = fm;
        categorie = cat;
    }
    public String getForme(){
        return formeMot;
    }
    public String getCategorie(){
        return categorie;
    }
    public String toStringBis(){
        return "Mot␣" + formeMot + "␣categorie:␣" + categorie;
    }
}
```

Un mot sera identifié par sa forme et sa catégorie. Par exemple le mot *alors* a comme forme *alors* et comme catégorie *adverbe*. Au lieu de redéfinir la méthode `toString` comme nous le faisons d'habitude, nous définissons une autre méthode appelée *toStringBis*. Cela nous permettra d'utiliser *toString* pour afficher la référence des objets que nous créerons.

La deuxième classe est une sous-classe des mots qui varient en nombre (singulier ou pluriel). En fait, il n'y a pas de catégorie syntaxique qui ne s'accorde qu'en nombre, donc on ne créera pas d'instance de cette classe. Mais celle-ci servira de classe mère à deux sous-classes : celle des mots qui s'accordent en genre et en nombre (article, nom, adjectif) et celle des mots qui se conjuguent en personne et nombre (les verbes). L'existence d'une sous-classe intermédiaire permet de partager l'attribut `nombre` et le code des méthodes afférentes entre les deux sous-classes.

```
public class MotNombre extends Mot{
    public static final int SINGULIER = 4;
    public static final int PLURIEL = 5;
    protected int nombre;
    public MotNombre(String forme, String cat, int nomb){
        super(forme,cat);
        if (nomb != SINGULIER && nomb != PLURIEL)
            throw new IllegalArgumentException("nombre␣ni␣SINGULIER␣ni␣PLURIEL");
        nombre = nomb;
    }
    public boolean estSingulier(){
        return nombre == SINGULIER;
    }
    public boolean estPluriel(){
        return nombre == PLURIEL;
    }
    public String toStringBis(){
        String res = "Mot␣variable␣en␣nombre␣" + formeMot + "␣categorie:␣" +
            categorie + "␣nombre:␣";
        if (nombre == SINGULIER){
            res = res + "␣singulier";
        }else{
            res = res + "␣pluriel";
        }
        return res;
    }
}
```

Le petit mot-clé `final` utilisé pour définir deux variables entières SINGULIER et PLURIEL est là pour indiquer au compilateur que ces deux variables ne doivent jamais changer de valeur. Dès lors, il est interdit d'utiliser une affectation pour donner une valeur à ces variables. C'est ce qu'on appelle des constantes. Il est assez fréquent que l'on définisse ainsi des valeurs constantes, ce qui permet de les désigner par leur nom. Dans le code, MotNombre.SINGULIER sera plus facile à comprendre que 4, même si au final, c'est la même valeur. Le nom de la variable est une sorte de commentaire qui illustre quel est le sens de la valeur à cet endroit du code. Dans cette classe, le sens en question est : l'entier 5 est utilisé pour coder dans le programme la notion de singulier.

```
public class MotGenreNombre extends MotNombre{
    public static final int MASCULIN = 6;
    public static final int FEMININ = 7;
    protected int genre;
    public MotGenreNombre(String form, String cat, int gen, int nombre){
        super(form, cat, nombre);
        if (gen != MASCULIN && gen != FEMININ)
            throw new IllegalArgumentException();
```

```
        genre = gen;
    }
    public boolean estMasculin(){
        return genre==MASCULIN;
    }
    public boolean estFeminin(){
        return genre==FEMININ;
    }
    public String toStringBis(){
        String res = "Mot␣" + formeMot + "␣categorie:␣" +
            categorie + "␣nombre:␣";
        if (nombre == SINGULIER){
            res = res + "␣singulier";
        }else{
            res = res + "␣pluriel";
        }
        if (genre == MASCULIN)
            res =res + "genre:␣masculin";
        else
            res =res + "genre:␣feminin";
        return res;
    }
}
```

Enfin la classe des mots qui varient en personne et en nombre.

```
public class MotPersonneNombre extends MotNombre{
    public static final int PERS1 = 1;
    public static final int PERS2 = 2;
    public static final int PERS3 = 3;
    protected int personne;
    public MotPersonneNombre(String forme, String cat, int pers, int nombre){
        super(forme, cat, nombre);
        if (pers<PERS1 || pers>PERS3)
            throw new IllegalArgumentException();
        personne = pers;
    }
    public int getPersonne(){
        return personne;
    }
    public String toStringBis(){
        String res = "Mot␣" + formeMot + "␣categorie:␣" +
            categorie + "␣nombre:␣";
        if (nombre == SINGULIER){
            res = res + "␣singulier";
        }else{
            res = res + "␣pluriel";
        }
        res = res + "␣personne:␣" + personne;
        return res;
    }
}
```

Un petit programme qui crée quelques objets.

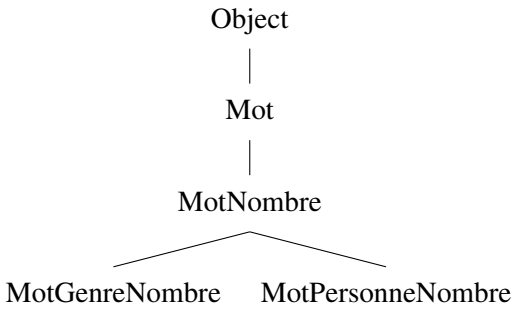
```
public class QuelquesMots{
    public static void main(String[] args){
        Mot unAdverbe = new Mot("alors","adverbe");
        MotGenreNombre unAdjectif = new MotGenreNombre("blanche","adjectif",
                                                       MotGenreNombre.FEMININ,
                                                       MotNombre.SINGULIER);
        MotPersonneNombre unVerbe = new MotPersonneNombre("parlons","verbe",
                                                          1, MotNombre.SINGULIER);
        System.out.println("unAdverbe␣toString:␣" + unAdverbe);
        System.out.println("unAdverbe␣toStringBis:␣" + unAdverbe.toStringBis());
        System.out.println("unAdjectif␣toString:␣" + unAdjectif);
        System.out.println("unAdjectif␣toStringBis:␣" + unAdjectif.toStringBis());
        System.out.println("unVerbe␣toString:␣" + unVerbe);
        System.out.println("unVerbe␣toStringBis:␣" + unVerbe.toStringBis());
    }
}
```

L'exécution de ce programme produit l'affichage suivant.

```
> java QuelquesMots
unAdverbe toString: Mot@65ae6ba4
unAdverbe toStringBis: Mot alors categorie: adverbe
unAdjectif toString: MotGenreNombre@48cf768c
unAdjectif toStringBis: Mot blanche categorie: adjectif nombre:  pluriel
                                                       genre: feminin
unVerbe toString: MotPersonneNombre@59f95c5d
unVerbe toStringBis: Mot parlons categorie: verbe nombre:  singulier
                                                       personne: 1
```

On voit que la méthode `toString` héritée de la classe `Object` affiche le type d'instanciation (classe utilisée par l'instruction `new`), un arobase puis une chaîne de caractère qui est la référence ou adresse en mémoire de l'objet. Cela correspond aux identifiants du genre `id476` de Pythontutor.

Si l'on récapitule la hiérarchie d'héritage de cet exemple, cela donne l'arborescence suivante.



2 Affectation à une variable

Si une variable a le type `Mot`, on peut lui affecter un objet instance de la classe `Mot`. C'est ce que nous faisons depuis le début du cours. Mais on peut également lui affecter une instance d'une sous-classe de `Mot`. Cela va être visible sur le programme suivant.

```
public class AffectationMot{
    public static void main(String[] args){
        Mot unMot = new Mot("alors","adverbe");
        System.out.println(unMot);
        unMot = new MotGenreNombre("blanche","adjectif",
                                   MotGenreNombre.FEMININ,
                                   MotNombre.SINGULIER);
        System.out.println(unMot);
        unMot = new MotPersonneNombre("parlons","verbe",
                                      1, MotNombre.SINGULIER);
        System.out.println(unMot);
        Object objet = new Mot("alors","adverbe");
        System.out.println(objet);
    }
}
```

Ce programme se compile sans erreur et à l'exécution, il affiche ce qui suit.

```
> java AffectationMot
Mot@3b22cdd0
MotGenreNombre@65ae6ba4
MotPersonneNombre@59f95c5d
Mot@5ccd43c2
```

On voit que ce qu'affiche la méthode `toString` (implicitement appelée par `System.out.println`) est le type d'instance de l'objet et non pas le type de la variable qui le contient. En effet, le type de la variable pour les trois premières lignes est `Mot` et pour la dernière, `Object`. Ces types ne sont pas affichés par ce code (sauf sur la première ligne, parce que le type d'instance est le même que le type de la variable).

3 Un objet a plusieurs types

Après avoir vu qu'on peut mettre des instances de différentes classes dans une même variable, faisons l'inverse : créons un objet et voyons dans quelles variables on peut le mettre. Une variable peut contenir un objet instance d'une sous-classe, donc une instance d'une classe donnée peut être affecté à une variable ayant comme type une classe ancêtre de cette classe (classe mère ou mère de la mère, etc).

C'est ce que montre le code suivant.

```
public class TypesDUneInstance{
    public static void main(String[] args){
        MotGenreNombre motgenre = new MotGenreNombre("blanche","adjectif",
                                                     MotGenreNombre.FEMININ,
                                                     MotNombre.SINGULIER);
        MotNombre motnombre = motgenre;
        Mot mot = motgenre;
        Object objet = motgenre;
        System.out.println(motgenre);
        System.out.println(motnombre);
        System.out.println(mot);
```

```
        System.out.println(objet);
    }
}
```

Dans ce programme, il n'y a clairement qu'un seul objet créé puisqu'il y a seulement un new (je néglige ici les objets String). Le même objet est mis dans quatre variables différentes ayant chacune un type différent. Il s'agit de la classe d'instanciation `MotGenreNombre` et de ses trois ancêtres dans l'arbre d'héritage.

Le fait qu'il s'agit bien du même objet est clair lorsqu'on affiche le contenu des variables : c'est le même à chaque fois (même type et même référence).

```
> java TypesDUneInstance
MotGenreNombre@4d591d15
MotGenreNombre@4d591d15
MotGenreNombre@4d591d15
MotGenreNombre@4d591d15
```

Ce qu'on voit sur cet exemple peut se généraliser. On peut considérer un type comme un ensemble de valeurs, l'ensemble des objets que l'on peut affecter à une variable de ce type. Avec cette vision, un type C contient toutes les instances de la classe C mais aussi toutes les instances de ses classes filles et toutes les instances des classes qui sont en-dessous de C dans l'arbre d'héritage (les descendantes de C).

Un objet instance d'une classe C, en tant que valeur, appartient à plusieurs types : le type C, mais aussi tous les types qui sont des ancêtres de C dans l'arbre d'héritage. L'ensemble des types d'un objet sont les classes situées sur le chemin qui relie la classe d'instanciation à la racine de l'arbre d'héritage.

Le type `Object`, qui est à la racine de l'arbre d'héritage, contient toutes les instances d'objets Java. Tous les objets ont `Object` parmi leurs types.

4 Passage de paramètre et héritage

Ce que nous avons vu pour l'affectation est vrai également lorsqu'il s'agit de donner des valeurs aux paramètres d'une méthode lors d'une invocation de méthode. Si un paramètre attend une valeur de type C, on peut lui donner n'importe quelle valeur de ce type, c'est-à-dire non seulement les instances de C, mais aussi les instances des sous-classes de C et de tous les descendants de C dans l'arbre d'héritage.

Si un méthode a un paramètre de type `Mot`, on peut l'invoquer avec une instance de `Mot`, une instance de `MotNombre`, une instance de `MotGenreNombre` ou une instance de `MotPersonneNombre`. En voici un exemple.

```
public class TypageInvocation{
    public static void afficheMot(Mot unMot){
        System.out.println("toString␣" + unMot);
        System.out.println("toStringBis:␣" + unMot.toStringBis());
    }
    public static void main(String[] args){
        Mot unAdverbe = new Mot("alors","adverbe");
        MotGenreNombre unAdjectif = new MotGenreNombre("blanche","adjectif",
                                                       MotGenreNombre.FEMININ,
                                                       MotNombre.SINGULIER);
```

```
        MotPersonneNombre unVerbe = new MotPersonneNombre("parlons","verbe",
                                                          1, MotNombre.SINGULIER);
        afficheMot(unAdverbe); // instance de Mot
        afficheMot(unAdjectif); // instance de MotGenreNombre
        afficheMot(unVerbe); // instance de MotPersonneNombre
    }
}
```

Le programme est correct. À l'exécution, il affiche ce qui suit.

```
> java TypageInvocation
toString Mot@65ae6ba4
toStringBis: Mot alors categorie: adverbe
toString MotGenreNombre@48cf768c
toStringBis: Mot blanche categorie: adjectif nombre: pluriel
                                                    genre: feminin
toString MotPersonneNombre@59f95c5d
toStringBis: Mot parlons categorie: verbe nombre: singulier
                                                    personne: 1
```

5 Que peut-on faire d'un objet ?

Ce qu'on peut faire avec un objet dépend du type avec lequel on le considère. Un objet a plusieurs types et on peut le mettre dans une variable qui a n'importe lequel de ces types. De même pour un paramètre. Mais ce qu'on peut faire avec cet objet va dépendre du type.

Prenons l'exemple d'un adjectif :

```
MotGenreNombre unAdjectif = new MotGenreNombre("blanche","adjectif",
                                               MotGenreNombre.FEMININ,
                                               MotNombre.SINGULIER);
```

Si on le met comme ici dans une variable de type `MotGenreNombre`, on pourra utiliser toutes les méthodes de l'objet : celles de la classe `MotGenreNombre` et aussi par héritage, celles des classes `MotNombre`, `Mot` et `Object`.

Si maintenant, on le met dans une variable de type `Mot`, cela change tout.

```
Mot unAdjectif = new MotGenreNombre("blanche","adjectif",
                                    MotGenreNombre.FEMININ,
                                    MotNombre.SINGULIER);
```

Le compilateur ne permettra d'appeler sur la variable que les méthodes qui existent dans la classe `Mot`. La déclaration du type de la variable détermine quelles méthodes peuvent être invoquées sur cette variable.

L'objet contient les attributs `genre` et `nombre` de même que les méthodes `estMasculin` ou `estSingulier`, mais on ne peut utiliser ces méthodes sur la variable qui contient l'objet et on ne peut pas accéder aux valeurs des deux attributs en question.

Le programme illustre ce point et montre que le même objet s'utilise différemment selon le type de la variable qui le contient, le même objet étant dans deux variables, l'une de type `MotGenreNombre` et l'autre de type `Mot`. Ce programme incorrect ne se compile pas et produit les erreurs que nous étudierons ci-dessous.

```
public class TypesDUneInstanceIncorrect{
    public static void main(String[] args){
        MotGenreNombre motgenre = new MotGenreNombre("blanche","adjectif",
                                                     MotGenreNombre.FEMININ,
                                                     MotNombre.SINGULIER);
        Mot mot = motgenre;
        System.out.println(motgenre);
        System.out.println(mot);
        System.out.println(motgenre.estMasculin());
        System.out.println(motgenre.estSingulier());
        System.out.println(motgenre.getForme());
        System.out.println(mot.estMasculin());
        System.out.println(mot.estSingulier());
        System.out.println(mot.getForme());
    }
}
```

La compilation donne les deux erreurs suivantes.

```
> javac TypesDUneInstanceIncorrect.java
TypesDUneInstanceIncorrect.java:12: error: cannot find symbol
        System.out.println(mot.estMasculin());
                           ^
  symbol:   method estMasculin()
  location: variable mot of type Mot
TypesDUneInstanceIncorrect.java:13: error: cannot find symbol
        System.out.println(mot.estSingulier());
                           ^
  symbol:   method estSingulier()
  location: variable mot of type Mot
2 errors
```

Ces messages sont clairs : le compilateur cherche les méthodes `estMasculin` et `estSingulier` dans la classe `Mot` et ces méthodes ne s'y trouvent pas. En revanche, pas de problème pour `getForme` qui se trouve bien dans la classe `Mot`.

6 Explications complémentaires

Le type donné à une variable ou un paramètre, de même que le type donné au résultat renvoyé par une méthode, ne décrivent pas la nature exacte et précise de l'objet contenu dans la variable, le paramètre ou renvoyé par une méthode.

Ce type est une façon de voir l'objet en question qui peut éventuellement masquer sa nature profonde, certaines de ses données ainsi que certaines de ses fonctionnalités implémentées par des méthodes.

Au peut voir un objet comme une successions d'étages : au rez-de-chaussée, on trouve les attributs et méthodes de la classe instanciée. Au premier étage, les attributs et méthodes de la super-classe qui sont héritées par l'objet et au dernier étage les attributs et méthode de `Object`. Un type est comme une palissade qui empêche de voir le bas de l'immeuble : on ne voit que les étages supérieurs. Et selon

la taille de la palissade, on voit plus ou moins d'étages. Et comme il n'y a pas de type plus restrictif que `Object`, on voit toujours au moins le dernier étage de l'immeuble au-dessus de la palissade.

7 Quelle méthode s'exécute ?

Le type d'une variable détermine quelles méthodes sont invocables sur cette variable, mais il ne détermine pas quelle méthode est utilisée. Prenons un exemple concret. Supposons comme à la section précédente qu'il y un objet instance de `MotGenreNombre` dans une variable de type `Mot`. Sur cette variable, on peut appeler la méthode `toStringBis` car cette méthode existe dans la classe `Mot`.

Est-ce que la méthode qui s'exécute dans ce cas est celle de `Mot` parce que la variable est de type `Mot` ou celle de la classe `MotGenreNombre` parce que l'objet est instance de `MotGenreNombre` ? Pour le savoir, testons cet exemple.

```
public class QuelleMethode{
    public static void main(String[] args){
        Mot mot = new MotGenreNombre("blanche","adjectif",
                                     MotGenreNombre.FEMININ,
                                     MotNombre.SINGULIER);
        System.out.println(mot.toStringBis());
    }
}
```

```
> java QuelleMethode
Mot blanche categorie: adjectif nombre: singulier genre: feminin
```

On voit que c'est la méthode de la classe `MotGenreNombre` qui s'exécute, celle qui affiche le nombre et le genre alors qu'il n'y a ni genre ni nombre dans la classe `Mot` et que la méthode `toStringBis` n'affiche que la forme et la catégorie.

Il y a deux questions différentes qui se posent à deux moments différents :

— a-t-on le droit d'invoquer une méthode sur une variable ? Cette question se pose à la compilation. La réponse dépend du type donné lors de la déclaration de variable.

— quelle méthode est exécutée ? Cette question se pose à l'exécution. La réponse dépend du type d'instanciation de l'objet contenu dans la variable.