TD 7 : Polymorphisme

Objectifs pédagogiques :

- **résolution de surcharge**
- **maîtrise des difficultés de l'héritage spécifiques à Java**
- **généricité**

7.1 Types, héritage et surcharge

Considérons les deux classes `A` et `B` suivantes :

```
public class A {

  public void f() {
    System.out.println("f() dans A");
  }

  public void m(A a) {
    System.out.println("m(A) dans A");
  }
}
```

```
public class B extends A {

  public void f() {
    System.out.println("f() dans B");
  }

  public void m(A a) {
    System.out.println("m(A) dans B");
  }

  public void m(B b) {
    System.out.println("m(B) dans B");
  }
}
```

Question 1. Indiquez dans les trois programmes ci-dessous pour chaque ligne si elle provoque une erreur à la compilation, une erreur à l'exécution ou aucune erreur. Indiquez également, pour chaque erreur, s'il est possible de la corriger. Il sera utile pour cela de déterminer le type statique et le type dynamique de chaque variable et de chaque expression.

Programme 1

```
A a = new A();
B b = new B();
A ab = new B();
B ba = new A();
```

Programme 2

```
A x = new B();
B y = x;
```

Programme 3

```
B[] tb = new B[2];
A[] ta = tb;
tb[0] = new A();
ta[0] = new A();
tb[0] = new B();
ta[0] = new B();
```

Question 2. Donnez l'affichage produit par le programme suivant :

```
A x = new A(); x.f();
A y = new B(); y.f();
B z = new B(); z.f();
```

Comme pour la question précédente, il est utile d'identifier le type statique des variables et le type dynamique des valeurs qu'elles contiennent.

Pour vous aider, l'annexe rappelle l'algorithme de résolution des méthodes en Java.

Question 3. Considérons enfin le programme suivant :

```
A x = new A();
A y = new B();
B z = new B();
x.m(x); x.m(y); x.m(z);
y.m(x); y.m(y); y.m(z);
z.m(x); z.m(y); z.m(z);
```

Donnez, pour chaque appel à `m`, la signature de méthode sélectionnée par le compilateur.
Trouvez ensuite quelle est la méthode appelée pour déterminer l'affichage produit par le programme.

7.2 Héritage et comparaison

Considérons une application dans laquelle un garagiste est amené à faire des comparaisons entre des voitures stockées dans son garage, qui peuvent être soit des voitures normales (à essence), soit des voitures diesel. Pour cela, nous disposons des classes `Voiture` et `VoitureDiesel` ci-dessous avec un opérateur de comparaison `estSimilaire` :

```
public class Voiture {
  private final String couleur;
  private final int longueur;

  public Voiture(String couleur, int longueur) {
    this.couleur = couleur;
    this.longueur = longueur;
  }
  public String getCouleur() {
    return couleur;
  }
  public int getLongueur() {
    return longueur;
  }
  public boolean estSimilaire(Voiture v) {
    return getCouleur().equals(v.getCouleur()) && getLongueur()==v.getLongueur();
  }
}
```

```
public class VoitureDiesel extends Voiture {
  private final boolean commonRail;

  public VoitureDiesel(String couleur, int longueur, boolean commonRail) {
    super(couleur,longueur);
    this.commonRail = commonRail;
  }
  public boolean isCommonRail() {
    return commonRail;
  }
  public boolean estSimilaire(VoitureDiesel v) {
    return super.estSimilaire(v) && isCommonRail()==v.isCommonRail();
  }
}
```

Question 4. Donnez la liste des méthodes de `VoitureDiesel`. Indiquez, pour chaque méthode, sa signature et si elle est héritée de `Voiture`, redéfinie, ou s'il s'agit d'une nouvelle définition.

Question 5. Donnez le résultat des appels suivants à `estSimilaire` sur des `Voiture`.
Vous indiquerez à chaque fois quelle est la signature de `estSimilaire` choisie par le compilateur, et quelle méthode est appelée, en plus de la valeur retournée.

```
Voiture v1 = new Voiture("verte", 3);
Voiture v2 = new Voiture("verte", 3);
Voiture v3 = new Voiture("verte", 4);
v1.estSimilaire(v1) ?
v1.estSimilaire(v2) ?
v1.estSimilaire(v3) ?
```

Question 6. Même question sur des `VoitureDiesel`.

```
VoitureDiesel v4 = new VoitureDiesel("beige", 3, true);
VoitureDiesel v5 = new VoitureDiesel("beige", 3, true);
VoitureDiesel v6 = new VoitureDiesel("beige", 3, false);
v4.estSimilaire(v4) ?
v4.estSimilaire(v5) ?
v4.estSimilaire(v6) ?
```

Question 7. Même question sur ce code qui compare des `Voiture` et des `VoitureDiesel`.

```
Voiture v7 = new Voiture("beige", 3);
Voiture v8 = new Voiture("rouge", 3);
Voiture v9 = new VoitureDiesel("beige", 3, true);
v4.estSimilaire(v7) ?
v7.estSimilaire(v4) ?
v4.estSimilaire(v8) ?
v8.estSimilaire(v4) ?
v4.estSimilaire(v9) ?
v9.estSimilaire(v4) ?
v6.estSimilaire(v9) ?
```

Question 8. Proposez des modifications dans les classes `Voiture` et/ou `VoitureDiesel` pour corriger l'incohérence détectée à la question précédente.

Question 9. Considérons le code suivant qui cherche une voiture `v` dans une liste `l` :

```
List<Voiture> l = ...;
Voiture v = ...;
l.contains(v);
```

Indiquez comment modifier `Voiture` pour que ce code retourne `true` si `l` contient une voiture similaire à `v` (au sens de `estSimilaire`).

7.3 Génériques

Considérons la fonction `echange` suivante, qui échange les éléments de la liste aux indices spécifiés :

```java
public static void echange(List<Object> list, int i, int j) {
   Object tmp = list.get(i);
   list.set(i, list.get(j));
   list.set(j, tmp);
}
```

Question 10. Le code suivant est-il valide ?

```java
List<Object> lo = new ArrayList<Object>();
List<String> ls = new ArrayList<String>();
lo.add("a"); lo.add("b"); lo.add("c");
ls.add("a"); ls.add("b"); ls.add("c");

echange(lo,0,2);
echange(ls,0,2);
```

Question 11. Modifiez la méthode `echange` pour que les deux appels fonctionnent.

Nous souhaitons maintenant écrire une fonction `maxList` qui prend une liste en paramètre et qui retourne l'élément le plus grand de la liste.

Question 12. Écrivez la méthode `maxList`.

Question 13. Les appels suivants fonctionnent-ils ?

```java
maxList(lo);
maxList(ls);
```

Annexe : résolution des appels de méthode

Résolution des appels de méthode. Rappelons qu'une variable (variable locale, attribut, argument formel de méthode) peut référencer, à l'exécution, un objet d'une classe dérivée de la classe mentionnée dans sa déclaration (ou, si un type interface est utilisé, une classe implantant l'interface ou une interface dérivée). Nous distinguons donc, pour une variable :

- son type *statique*, défini une fois pour toutes dans la déclaration de la variable (par exemple, `A`, pour la déclaration de variable `A a`) ;
- son type *dynamique*, la classe de l'objet réellement référencé, qui peut changer au cours de l'exécution (par exemple, `B`, après l'initialisation `a = new B()`, quand `B extends A`).

Le type statique est connu du compilateur, mais pas le type dynamique. En reprenant l'exemple de l'exercice 7.1, quand la classe `A` est compilée, le compilateur sait que l'argument `a` de la méthode `m` a pour type statique `A`, mais il ne connaît pas le type dynamique, qui peut changer d'un appel à l'autre, et peut même référencer une classe qu'il n'a pas encore compilée (comme la classe `B`).

Lors d'un appel de méthode `obj.methode(arg1,...,argN)`, ces deux informations se combinent pour déterminer quel est le code réellement exécuté parmi toutes les définitions et redéfinitions de méthodes nommées `methode`. Il faut donc :

1. Déterminer le type statique `S` de l'expression `obj`.
2. Collecter dans la classe (ou l'interface) `S` la liste des signatures de méthodes ayant le nom `methode` et le bon nombre d'arguments (`N`).

   Une signature a la forme : `T methode(T1 a1,...,TN aN)`, où `T` est le type de retour, et les `Ti` le type statique des arguments formels `ai`. Ces signatures diffèrent dans le choix des types statiques `Ti`.
3. Utiliser le type statique `S1` à `SN` des arguments `arg1` à `argN` utilisés lors de l'appel pour sélectionner une unique signature `methode(T1,...,TN)` dans cette liste.

   Le type de retour `T` n'est pas pris en compte. Il n'est pas nécessaire d'avoir une correspondance exacte de type statique, `Ti`=`Si`. Il suffit que les `Si` soient des classes dérivées des `Ti` pour que la signature convienne ; il y a alors conversion implicite du type `Si` en type `Ti`. Si plusieurs choix sont possibles, le compilateur choisira la signature qui nécessite le moins de conversions (c'est-à-dire, ayant le plus grand nombre d'arguments avec une correspondance exacte de type statique `Ti`=`Si`). S'il y a ambiguïté, i.e. s'il existe plusieurs signatures permettant le nombre minimum de conversions, alors le compilateur indiquera une erreur.
4. Déterminer le type dynamique `D` d'`obj`, qui peut être `S` ou un type dérivé.
5. Déterminer le corps de la méthode ayant pour signature `methode(T1,...,TN)` sélectionnée en 3 et visible dans la classe `D`.

   Il peut s'agir d'une définition dans `D`, ou d'une définition héritée d'une classe ancêtre de `D`.

Les étapes 1 à 3 sont effectuées par le compilateur, à la compilation de l'appel de méthode. Les étapes 4 et 5 sont effectuées à l'exécution, à chaque appel de méthode.

Attention : si la classe `D` du type dynamique d'`obj` possède une méthode d'une signature compatible avec l'appel, mais aucune méthode de cette signature n'est présente dans le type statique `S`, cette méthode ne sera jamais sélectionnée puisqu'elle n'apparaît pas dans la liste construite à l'étape 2 !