POGL 2020 – Liaison dynamique

1 Vue d'ensemble

Transtypage

Une classe fille (autrement appelée sous-classe) peut être considérée comme une *spécialisation* de sa classe mère (autrement appelée super-classe). En particulier un objet de la classe fille peut être vu comme appartenant à la classe mère, car l'objet de la classe fille possède tous les attributs et méthodes des objets de la classe mère. En revanche, un objet de la classe mère ne peut pas être vu comme appartenant à la classe fille, car il lui manque peut-être certains des attributs ou méthodes de la classe fille. En d'autres termes, l'ensemble des objets de la classe fille peut être considéré comme un sous-ensemble des objets de la classe mère.

```
abstract class A {
    abstract public void a();
}
class B extends A {
    public void a() { System.out.println("Classe B"); }
    public void b() { System.out.println("Classe B"); }
}
B objB  = new B();
A objBA = new B();
```

Point important : une fois qu'un objet de la classe `B` est enregistré comme appartenant à la classe mère `A` (c'est ce qui se passe avec `objBA`), on ne peut plus accéder qu'aux attributs et méthodes qui étaient déjà dans `A`. En particulier, on peut accéder à la méthode `a()`, qui est abstraite mais quand même déclarée, et on ne peut pas accéder à la méthode `b()`, qui n'existe que dans la classe `B`.

Appel	Affichage	Erreur ?
`objB.b()`	`Classe B`	
`objB.a()`	`Classe B`	
`objBA.b()`		Erreur de compilation : pas de `b()` dans `A`
`objBA.a()`	`Classe B`	

Liaison dynamique

Lorsqu'une méthode `a()` est redéfinie par une classe fille `B`, le choix de la méthode à appliquer n'est pas fait statiquement sur le seul critère du type. En effet, par l'effet du transtypage évoqué ci-dessus, le type d'un objet peut apparaître différent de ce qu'il est en réalité.

```
class A {
    public void a() { System.out.println("Classe A"); }
}
class B extends A {
    public void a() { System.out.println("Classe B"); }
}
A objA  = new A();
B objB  = new B();
A objBA = new B();
```

Appel	Affichage	Erreur ?
`objA.a()`	`Classe A`	
`objB.a()`	`Classe B`	
`objBA.a()`	`Classe B`	

Dans cet exemple, nous avons d'abord deux objets `objA` et `objB`, respectivement de classe `A` et de classe `B`, pour lesquels sont appelées respectivement les méthodes `a()` définies dans la classe `A` et dans la classe `B`.

Le cas de l'objet `objBA` est un peu plus compliqué : il a été créé avec l'instruction `new B()` ; comme un objet de classe `B` (il s'agit de son type *réel*), mais par la déclaration `A objBA` il a été transtypé en un objet de classe `A` (il s'agit de son type *apparent*). Le type apparent est celui qui est vu par le compilateur et par le vérifieur de type. En revanche, lors de l'appel d'une méthode qui existe à la fois dans la classe mère et dans la classe fille, c'est le type réel qui est consulté : dans notre exemple, même si l'objet `objBA` est *apparemment* dans la classe `A`, l'appel de méthode `objBA.a()` utilise la définition de `a()` fournie par la classe `B` (la classe *réelle* de l'objet).

Cette information sur la classe réelle d'un objet ne peut pas être connue statiquement : si un tableau est rempli d'objets qui sont *apparemment* de la classe `A`, mais dont certains sont *en fait* de la classe `B`, il n'est pas possible lorsqu'on regarde une case au hasard de savoir à l'avance quelle sera la classe réelle de l'objet ("à l'avance" signifiant : "au moment de la compilation"). Le choix définitif de la méthode à appliquer est donc fait *dynamiquement*, pendant l'exécution du programme. On appelle ce phénomène la *liaison dynamique*.

D'un point de vue pratique, pour chaque appel d'une méthode `a()`, le compilateur détermine *statiquement* une *famille* (c'est-à-dire un ensemble) de méthodes `a()` potentiellement applicables. Cette famille regroupe typiquement une définition de méthode `a()` d'une classe mère et toutes les redéfinitions faites dans les classes filles, ou petites-filles, ou descendantes plus lointaines. Ce choix est fait en fonction des types *apparents*, et règle en particulier les cas de surcharge de `a()`. Ensuite, lors de l'exécution du programme, une méthode `a()` est choisie *dynamiquement* parmi cette famille en fonction du type *réel* de l'objet.

Pourquoi utiliser le transtypage et la liaison dynamique ?

En reprenant l'exemple du démineur, nous pourrions imaginer avoir deux classes pour les cases du jeu : une classe principale, et une sous-classe spécifique pour les cases piégées. Cette sous-classe redéfinirait alors la méthode décrivant la réaction aux clics.

```
class Case {
    public void clicGauche() { /* Code OK */ }
}
class Piegee extends Case {
    public void clicGauche() { /* Code Perdu */ }
}
```

Ensuite, toutes les cases, piégées ou non, sont stockées dans un même tableau. Ce tableau ne peut pas avoir le type `Piegee[][]`, car il ne contient pas que des cases piégées. Son type sera donc `Case[][]` et il pourra contenir à la fois des cases normales et des cases piégées, grâce au transtypage. Ainsi, si l'on considère une case `c` prise dans le tableau, cette case `c` a le type apparent `Case`. En revanche, son type réel peut être `Case` ou `Piegee`, et l'on veut qu'un clic sur une case de type réel `Piegee` exécute le code perdant et non le code gagnant.

2 Appel de méthodes et liaison dynamique en détail

On écrit toujours un appel de méthode de l'une des deux façons suivantes :

— `x.f(y)`, où `x` est l'objet dont on déclenche une méthode, `f` est le nom de la méthode appelée, et `y` est l'ensemble des paramètres fournis. On appelle les arguments `y` les *paramètres explicites* et `x` le *paramètre implicite*.

— `f(y)`. Cette deuxième forme est en réalité un abréviation pour `this.f(y)`, où `f` et `y` sont comme avant le nom de méthode et les paramètres explicites, et où le paramètre implicite est `this`, l'objet depuis lequel on invoque la méthode.

Prenons donc un appel de méthode de la forme `x.f(y)`, où le paramètre implicite `x` est éventuellement `this`, et notons `C` la classe réelle de l'objet `x`. La méthode à appeler est déterminée ainsi :

1. On construit la liste L de toutes les méthodes appelées `f` se trouvant dans l'une des deux situations suivantes :
   — définie dans la classe `C`, ou
   — définie dans une super-classe de `C` et qualifiée par `public` ou `protected`.
2. On élimine de la liste L toutes les méthodes dont la signature ne correspond pas aux paramètres explicites `y`. Incidemment, cette étape résout les éventuelles surcharges de notre méthode `f`.
3. Si on a obtenu une méthode `f(y)` associée à l'un des qualificatifs `private`, `static`, ou `final`, alors on invoque celle-ci.
4. Sinon, on cherche la définition de `f(y)` *la plus récente* : si on a une définition de `f(y)` dans `C`, alors on invoque celle-ci ; sinon, on regarde dans la super-classe de `C` ; si on ne trouve toujours pas, on remonte à la super-classe de la super-classe de `C`, etc.

```
class A {
    private a(int n)  { System.out.println("Classe A"+n); }
    public  a(char c) { System.out.println("Classe A"+c); }
}
class B extends A {
    public  a(int n)  { System.out.println("Classe B"+n); }
}
class C extends A {
    public  a(int n)  { System.out.println("Classe C"+n); }
    public  a()       { System.out.println("Classe C");   }
}
class D extends C {
    public  a(int n)  { System.out.println("Classe D"+n); }
}
class E extends D {
    private b(int n)  { System.out.println("Classe E"+n);   }
}
E obj = new E();
```

On considère un appel `obj.a(3)`, où le paramètre implicite `obj` est de la classe `E` et où le paramètre explicite `3` est un unique argument de type `int`. Les étapes sont les suivantes :

1. La liste L contient les définitions `A.a(char)`, `C.a(int)`, `C.a()`, `D.a(int)`. La définition `A.a(int)` n'est pas retenue car elle est qualifiée par `private`, et la définition `B.a(int)` non plus car `B` n'est pas une super-classe de `E`. La définition `E.b(int)` est écartée car elle n'a pas le bon nom.

2. On élimine les définitions `A.a(char)` et `C.a()` car elles ne correspondent pas au type de l'argument explicite `3`.
3. Il n'y a pas dans L de méthode qualifiée par `private`, `static`, ou `final`, donc on passe à l'étape suivante.
4. Il n'y a pas de définition pour `a(int)` dans la classe `E`, donc on regarde dans sa super-classe immédiate `D`. Il y a une définition pour `a(int)` dans `D` : c'est celle-ci qu'on invoque. La définition `C.a(int)` est donc ignorée (elle a été redéfinie par `D.a(int)`).

Finalement, l'appel utilise la définition `D.a(int)`.

En pratique, on ne reproduit pas cette procédure dynamiquement à chaque fois qu'un appel de méthode est exécuté. Pour chaque classe, une *table de méthodes* est précalculée statiquement par le compilateur, qui indique directement les endroits où aller chercher les définitions des méthodes pour chaque combinaison `f(y)` acceptable. Pour exécuter un appel de méthode `x.f(y)`, on va donc consulter la table des méthodes de la classe *réelle* de `x`.

Dans notre exemple, les tables de méthodes des classes `C` et `E` sont les suivantes :

Classe `C`			Classe `E`		
`a()`	:	`C.a()`	`a()`	:	`C.a()`
`a(int)`	:	`C.a(int)`	`a(int)`	:	`D.a(int)`
`a(char)`	:	`A.a(char)`	`a(char)`	:	`A.a(char)`
			`b(int)`	:	`E.b(int)`

Si un appel ne correspond à aucune des lignes de la table de méthodes de la classe concernée, alors une erreur est renvoyée (normalement, cela se fait à la compilation).

3 Transtypage en détail

Lorsqu'une classe `B` est une sous-classe d'une classe `A`, on a déjà vu que tout objet de la classe `B` pouvait être utilisé comme un objet de la classe `A` (en d'autres termes, les objets de la classe `B` forment un sous-ensemble des objets de la classe `A`).

Ceci est une opération de *transtypage* "vers le haut", dans laquelle un objet `obj` d'une sous-classe `B` prend l'apparence d'un objet d'une super-classe `A`. L'objet `obj` a alors un type *apparent* `A` tandis que son type *réel* reste `B`. Le transtypage est une modification du type apparent.

Le transtypage est également possible dans l'autre sens, "vers le bas", via la notation `(B)obj`, mais seulement si le type réel de l'objet `obj` le permet : le type réel de l'objet `obj` doit être un sous-type du nouveau type apparent souhaité `B`.

```
class A { }
class B extends A { }
class C extends A {
    public void print() { System.out.println("Classe C"); }
}
class D extends C { }

public static void f(C obj) { obj.print(); }

A objA = new A();
B objb = new B();
C objc = new C();
D objd = new D();
A objx = new D();
```

Ici, les objets `obja`, `objb`, `objc`, et `objd` ont chacun un type apparent égal à leur type réel `A`, `B`, `C`, ou `D`. L'objet `objx` en revanche a le type réel `D` et le type apparent `A`, ce qui est légitime car `A` est une super-classe de `D`. Une définition `C objy = new A();` n'aurait en revanche pas été possible : le compilateur l'aurait interdite car `C` n'est pas une super-classe de `A` (et le compilateur aurait eu raison, en particulier car les objets de la classe `C` sont censés posséder une méthode `print()`, que n'ont pas les objets de type réel `A`).

La méthode statique `f(C)` attend en paramètre explicite un objet `obj` de la classe `C`. Le compilateur vérifie ceci sur la base du type apparent de `obj`. Voici différents cas d'appel qui seraient acceptés ou rejetés.

Appel	Affichage	Erreur ?
`f(obja)`		Erreur de compilation : `obja` n'est pas de type `C`
`f(objb)`		Erreur de compilation : `objb` n'est pas de type `C`
`f(objc)`	`Classe C`	
`f(objd)`	`Classe C`	
`f(objx)`		Erreur de compilation : `objx` n'est pas de type `C`
`f((C)objx)`	`Classe C`	
`f((D)objx)`	`Classe C`	
`f((C)objb)`		Erreur de compilation : transtypage incompatible
`f((C)obja)`		Erreur d'exécution : transtypage impossible
`f((C)(A)objb)`		Erreur d'exécution : transtypage impossible

Le compilateur rejette les trois cas où le type apparent de l'argument n'est pas un sous-type du type `C` attendu (objets `obja` et `objx` de type apparent `A`, et objet `objb` de type apparent `B`), et accepte les autres appels où l'argument a le type apparent `C` ou `D` (éventuellement grâce à une opération de transtypage `(C)objx` ou `(D)objx`).

Une deuxième sorte d'erreur de compilation apparaît au niveau du transtypage `(C)objb`, car le compilateur détecte que `B` et `C` sont dans des branches différentes, et qu'il n'est pas possible de passer de l'une à l'autre. La combinaison `(C)(A)objb` en revanche, même si elle a le même effet, n'est pas bloquée par le compilateur, puisque ce dernier détecte d'abord un transtypage de `B` vers `A` (toujours possible) puis un transtypage de `A` vers `C` (parfois possible, à déterminer à l'exécution en fonction du type réel).

Enfin, dans le cas d'un transtypage vers un sous-type, le compilateur accepte le code écrit, et laisse à la machine virtuelle la responsabilité de vérifier dynamiquement (à l'exécution) que le type réel de l'objet est compatible avec le transtypage. Les cas `(C)objx` et `(D)objx` s'exécutent correctement car `objx` est de type réel `D`. Les cas `(C)obja` et `(C)(A)objb` provoquent une erreur l'exécution, car les types réels `A` et `B` ne permettent pas un transtypage vers `C`.

Pour éviter ce genre d'erreur de transtypage à l'exécution, on utilise en général un test avec `instanceof` permettant de vérifier que le transtypage est possible :

```
if (obj instanceof C) {
    f((C)obj);
} else { ... }
```