

## 1 Concordance des types : type statique et type dynamique

### Concordance de types :

Un type B concorde avec un type A si l'une des conditions suivantes est vérifiée :

- A et B sont le même type (*réflexivité de la relation de concordance*).
- B hérite de A
- il existe un type T, tel que B concorde avec T et T concorde avec A (*transitivité de la relation de concordance*).

Soient deux variables a et b de types respectifs A et B tels que B concorde avec A mais A ne concorde pas avec B,

- l'affectation a=b est admise par le compilateur,
- l'affectation b=a est refusée.

Concernant le passage de paramètres,

- si il existe une méthode p(A a), l'appel p(b) est accepté par le compilateur,
- alors pour une méthode q(B b) l'appel q(a) sera refusé par le compilateur.

On considère la définition des classes A et B suivantes :

```

1 class Animal {
2     public Animal f(Animal z, Canard t){
3         Canard unCanard;
4         if (!this.estUneSorteDe(z))
5             return z;
6         if (t.estUnCanard())
7             return t;
8
9         unCanard = new Canard();
10        return unCanard;
11    }
12
13    public boolean estUneSorteDe(Animal x){
14        return !(x.estUnCanard());
15    }
16
17    public boolean estUnCanard(){
18        return false;
19    }
20 }
21
22 class Canard extends Animal {
23     public boolean estUneSorteDe(Animal x){
24         return true;
25     }
26
27     public boolean estUnCanard(){
28         return true;
29     }
30 }
```

▷ **Question 1.** En considérant les 16 combinaisons possibles de types des variables x, y, z et t, déterminer les erreurs de compilation de l'instruction `x = y.f(z,t)`.

## Réponse

L'exercice consiste à appliquer la règle de concordance de types pour l'expression  $x=y.f(z,t)$ . On s'aperçoit rapidement que l'on peut éliminer tous les appels où  $t$  est de type `Animal` (c'est à dire la moitié des cas). Ensuite, l'étude de l'affectation montre que l'hypothèse " $x$  est de type `Canard`" n'est pas autorisée, ce qui élimine de nouveau la moitié des appels restants. Au final, seul 4 appels sont possibles, ceux où  $x$  est de type `Animal` et  $t$  de type `Canard` et  $y, z$  de types quelconques `Animal` ou `Canard`.

x	y	z	t	$x = y.f(z,t)$
Animal	Animal	Animal	<b>Animal</b>	erreur
<b>Canard</b>	Animal	Animal	<b>Animal</b>	erreur
Animal	Canard	Animal	<b>Animal</b>	erreur
<b>Canard</b>	Canard	Animal	<b>Animal</b>	erreur
Animal	Animal	Canard	<b>Animal</b>	erreur
<b>Canard</b>	Animal	Canard	<b>Animal</b>	erreur
Animal	Canard	Canard	<b>Animal</b>	erreur
<b>Canard</b>	Canard	Canard	<b>Animal</b>	erreur
Animal	Animal	Animal	Canard	.
<b>Canard</b>	Animal	Animal	Canard	erreur
Animal	Canard	Animal	Canard	.
<b>Canard</b>	Canard	Animal	Canard	erreur
Animal	Animal	Canard	Canard	.
<b>Canard</b>	Animal	Canard	Canard	erreur
Animal	Canard	Canard	Canard	.
<b>Canard</b>	Canard	Canard	Canard	erreur

## Fin réponse

▷ **Question 2.** Étudier le type dynamique du résultat de la méthode `f()`, en déduire une post-condition.

## Réponse

On peut faire exécuter le programme en considérant les différents types pour les variables `y` et `z` (exécution de `estUneSorteDe()`), on s'aperçoit que le résultat de `y.f(z,t)` est toujours de type `Canard`. On peut donc écrire la post-condition `ensures(\result).estUnCanard();` (exprimée en syntaxe JML que l'on n'étudie plus)

## Fin réponse

## 2 Utilisation de instanceof

**Rappel :** `instanceof` est un opérateur qui prend à sa gauche un objet et à sa droite un nom de classe ou d'interface. Il retourne `true` si l'objet passé en première opérande est du type donné en seconde opérande. En interne le calcul est effectué est identique à tenter un transtypage forcé (*cast*) et regarder si une exception `ClassCastException` est levée.

▷ **Question 3.** Déterminer l'affichage du programme suivant :

```

1 class Animal {}
2
3 class Canard extends Animal {}
4
5 class TestInstanceOf {
6
7     public static void main(String args[]) {
8         Object c = new Canard();
9         Animal a = new Animal();
10        System.out.println("c est une instance de Canard : " + (c instanceof Canard));
11        System.out.println("c est une instance de Animal : " + (c instanceof Animal));
12        System.out.println("c est une instance de Object : " + (c instanceof Object));
13        System.out.println("c est une instance de Integer : " + (c instanceof Integer));
14        System.out.println("a est une instance de Canard : " + (a instanceof Canard));
15        System.out.println("a est une instance de Animal : " + (a instanceof Animal));
16        System.out.println("a est une instance de Object : " + (a instanceof Object));
17        System.out.println("a est une instance de Integer : " + (a instanceof Integer));
18    }
19 }
```

## Réponse

```

1 c est une instance de Canard : true
2 c est une instance de Animal : true
3 c est une instance de Object : true
4 c est une instance de Integer : false
5 a est une instance de Canard : false
6 a est une instance de Animal : true
7 a est une instance de Object : true
8 a est une instance de Integer :
9 Erreur de compilation !!!
```

```
10 instanceof.java:17: incompatible types
11 found   : Animal
12 required: java.lang.Integer
13     System.out.println("a est une instance de Integer : " + (a instanceof Integer));
14
15 1 error
```

À noter que l'instruction `c instanceof Integer` ne pose aucun problème de compilation. Il se peut tout à fait qu'une variable de type statique `Object` contienne une référence à un objet de type `Canard`. Par contre, ce n'est pas le cas pour l'instruction `a instanceof Integer`, d'où l'erreur de compilation.

---

Fin réponse

---

▷ **Question 4.** En utilisant l'opérateur de transtypage (*cast*) écrivez une suite d'instructions permettant de simuler le comportement de l'instruction `instanceof`.

---

Réponse

---

```
1 public class instanceofsim {
2
3     public static void main(String[] args) {
4         class Animal {
5             };
6
7         class Canard extends Animal {
8             };
9
10        Animal c = new Canard();
11        Animal a = new Animal();
12
13        // la suite d'instructions
14        boolean flag = true;
15        try {
16            Object o = (Canard) a; // will return false
17            //Object o = (Animal) c; // will return true
18        } catch (ClassCastException e) {
19            flag = false;
20        }
21
22        System.out.println(flag);
23    }
24 }
```

---

Fin réponse

---

### 3 Constructeurs et héritage

**Rappel :** Le constructeur d'une sous-classe doit faire appel au constructeur de la super-classe. Dans le cas, où le constructeur ne fait pas appel explicitement à un constructeur de la super-classe, le compilateur générera de manière implicite un appel au constructeur sans paramètre de la super-classe.

On considère la définition des classes suivantes :

```

1 class A {
2     A() {
3         System.out.println("constructeur de A");
4     }
5 }
6
7 class B extends A {
8     B() {
9         System.out.println("constructeur de B");
10    }
11
12    B(int x) {
13        this();
14        System.out.println("autre constructeur de B");
15    }
16 }
17
18 class C extends B {
19     C() {
20         super(3);
21         System.out.println("constructeur de C");
22     }
23
24     public static void main(String [] arg) {
25         new C();
26     }
27 }
```

▷ **Question 5.** Déterminer l'affichage de la méthode principale void main(String args[]).

Réponse

```

1 constructeur de A
2 constructeur de B
3 autre constructeur de B
4 constructeur de C
```

Fin réponse

▷ **Question 6.** Même question en supposant que l'on supprime l'instruction this() dans la classe B.

Réponse

```

1 constructeur de A
2 autre constructeur de B
3 constructeur de C
```

Fin réponse

▷ **Question 7.** Même question si l'on supprime l'instruction super(3) dans la classe C. (Remarque : on supposera que l'instruction this() de la question précédente est supprimée.)

Réponse

```

1 constructeur de A
2 constructeur de B
3 constructeur de C
```

Fin réponse

▷ **Question 8.** Même question si l'on supprime complètement la définition du constructeur sans paramètre de la classe B. (Remarque : cette fois encore l'instruction `super(3)` n'a pas été ré-introduite.)

Réponse

```
1 constructeurs4.java:14: cannot find symbol
2 symbol   : constructor B()
3 location: class B
4   C() {
5   }
6 1 error
```

Fin réponse

## 4 Typage et héritage

**Algorithme simplifié de sélection de la méthode à appeler :**

**À la compilation :**

- Regarder le type statique du receveur de l'appel de méthode (type de la référence à la déclaration).
- Sélectionner tous les profils des méthodes définies dans la classe correspondant à ce type. Cette sélection se base sur le nombre de paramètres et les types statiques de ces paramètres. Le type de retour de la méthode n'a aucune influence.

**À l'exécution :**

- Regarder le type dynamique du receveur de l'appel de méthode (type de l'objet référencé).
- Sélectionner dans la classe correspondant à ce type, la méthode dont le profil correspond à une des méthodes candidates sélectionnées lors de la phase de compilation. Si une telle méthode n'existe pas remonter dans la super-classe et re-exécuter la sélection.

On considère le programme suivant :

```
1 class Animal {
2     void m(Animal a) {
3         System.out.println("m de Animal");
4     }
5     void n(Animal a){
6         System.out.println("n de Animal");
7     }
8 }
9
10 class Canard extends Animal {
11     void m(Animal a){
12         System.out.println("m de Canard");
13     }
14     void n(Canard c){
15         System.out.println("n de Canard");
16     }
17 }
18
19 class Test {
20     public static void main(String[] argv){
21         Animal a = new Canard();
22         Canard c = new Canard();
23         Animal a1 = new Animal();
24         a.m(c);
25         a.n(c);
26         c.m(c);
27         c.n(c);
28         a.m(a1);
29         a.n(a1);
30         a1.m(c);
31         a1.n(c);
32     }
33 }
```

▷ **Question 9.** Déterminer l'affichage lors de l'exécution de ce programme.

Réponse

Exemple de justification pour le premier appel :

Pour `a.m(c)`, `a` est de type `Animal`, on cherche dans la classe `Animal` la méthode `m` avec un paramètre de type `Canard`. On trouve la méthode `m(Animal a)`, c'est une méthode de ce type que l'on appliquera. À l'exécution, on remarque que `a` a comme type dynamique `Canard` et dans la classe `Canard`, il y a une méthode `m(Animal a)` (redéfinissant la méthode originel de la classe `Animal`). C'est donc cette méthode qui est appelée. On obtient donc le résultat "m de Canard".

```
1 m de Canard          [ a.m(c) ]
2 n de Animal          [ a.n(c) ]
```

Pour ce deuxième appel, à la compilation, la méthode candidate doit :

- être définie dans `Animal`
- s'appeler `n`
- prendre en paramètre un `Animal`

Donc, il est décidé que les méthodes candidates doivent avoir le profil `n(Animal a)`.

À l'exécution, même si le type dynamique de `a` est `Canard`, la méthode `n(Canard a)` de la classe `Canard` ne redéfinit pas la méthode `n(Animal a)`. Ce n'est donc pas cette méthode qui est appelée. C'est surprenant, mais c'est toute la subtilité de l'algorithme de sélection de méthode utilisé.

```
1 m de Canard          [ c.m(c) ]
2 n de Canard          [ c.n(c) ]
3 m de Canard          [ a.m(a1) ]
4 n de Animal          [ a.n(a1) ]
```

Une explication rapide du dernier appel (c'est Martin qui s'y colle) :

- `Canard.n(Canard b)` est inapplicable puisque `a1` est un `Animal` (concordance).
- C'est donc `Animal.n(Animal a)` qui est utilisée

```
1 m de Animal          [ a1.m(c) ]
2 n de Animal          [ a1.n(c) ]
```

Fin réponse

▷ **Question 10.** Déterminer le nouvel affichage si l'on ajoute dans la classe `Animal` une méthode `m(Canard c)` dont la définition est la suivante :

```
void m(Canard c) {
    System.out.println("m de Animal version 2");
}
```

Réponse

```
1 m de Animal version 2 [ a.m(c) ]
2 n de Animal          [ a.n(c) ]
3 m de Animal version 2 [ c.m(c) ]
4 n de Canard          [ c.n(c) ]
5 m de Canard          [ a.m(a1) ]
6 n de Animal          [ a.n(a1) ]
7 m de Animal version 2 [ a1.m(c) ]
8 n de Animal          [ a1.n(c) ]
```

Fin réponse