INSA
CENTRE VAL DE LOIRE

POO Avancée : Java

Divers

Salwa SOUAF

Outline

Variance et héritage

La notion de variance intervient lorsqu'on s'intéresse au sous typage de données complexe, typiquement des classes. Le langage de programmation doit décider de règles de typage en cas d'héritage et pour les classes génériques. On distingue:

- la covariance: préservation du type du plus spécifique au plus générique
- la contravariance: du type le plus générique au plus spécifique
- la bivariance: si les deux sont possibles
- l'invariance: si aucun ne l'est

Pour éviter le problème de variance lors de l'héritage, une solution simple consiste à garder le typage originel, par exemple pour une signature de méthode:

```
public class Livre {
	Texte f(Texte x) {
		System.out.println("Je retourne un Texte d'un Livre.");
		return x;
	}}
```

```
public class LivreRouge extends Livre {
	// Invariant:
	@Override
	Texte f(Texte x) {
		System.out.println("Je retourne un Texte Rouge.");
		return x;
}}
```

Variance et héritage

Héritage et cast

Lorsqu'on appelle une méthode surchargée, Java va toujours choisir la plus spécialisée, même si l'on cast un objet dans son super type. L'exemple suivant illustre cela:

```
public class MainRouge {
      public static void main(String[] args) {
               LivreRouge lr = new LivreRouge();
               Texte texte = new Texte();
               Texte t = lr.f(texte);
               Livre l2 = (Livre)lr;
               Texte t2 = l2.f(texte);
      }}
```

Donne, à l'exécution:

```
Je retourne un Texte Rouge.
Je retourne un Texte Rouge.
```

Personne ne peut appeler la méthode de la super classe, sauf depuis l'intérieur de la classe, à l'aide de super(). Pour permettre de l'appeler tout de même, on peut bricoler une méthode supplémentaire, mais cela va à l'encontre des notions d'encapsulation et d'héritage:

```
Texte originalF(Texte x) {
                return super.f(x);
        }
```

Variance et héritage

Type de retour covariant

Le type de retour d'une fonction est covariant: on peut redéfinir une méthode en remplaçant le type de retour par un type plus spécialisé que dans la classe parente. On peut donc par exemple spécifier une méthode TexteVert f(Texte x):

```
public class LivreVert extends Livre {
	// Type de retour covariant
	@Override
	TexteVert f(Texte x)
	{
		System.out.println("Je retourne un Texte Vert.");
		return new TexteVert();
	}}
```

Qui donne, a l'exécution de:

```
LivreVert lv = new LivreVert();
Texte texte = new Texte();
Texte t = lv.f(texte);
Livre l2 = (Livre)lv;
Texte t2 = l2.f(texte);
```

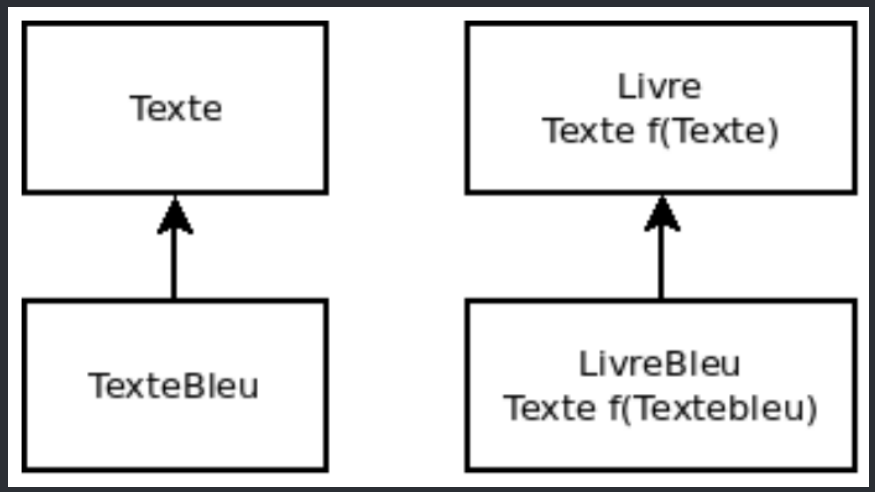
```
Je retourne un Texte Vert.
Je retourne un Texte Vert.
```

On remarquera qu'ainsi, l'appel à la fonction f sur l2 permet de récupérer un objet de type Texte, sans vraiment savoir quel est le type réel. La covariance du type de retour est donc "sure", d'un point de vue typage.

Variance et héritage

Paramètres et héritage

Lorsqu'on construit un modèle objet s'appuyant sur l'héritage, il serait assez naturel de faire un raisonnement similaire pour les paramètres de méthodes qui sont redéfinies. Ainsi, par exemple, si TexteBleu hérite de Texte on peut penser que LivreBleu pourrait avoir une méthode utilisant un paramètre covariant, i.e.:



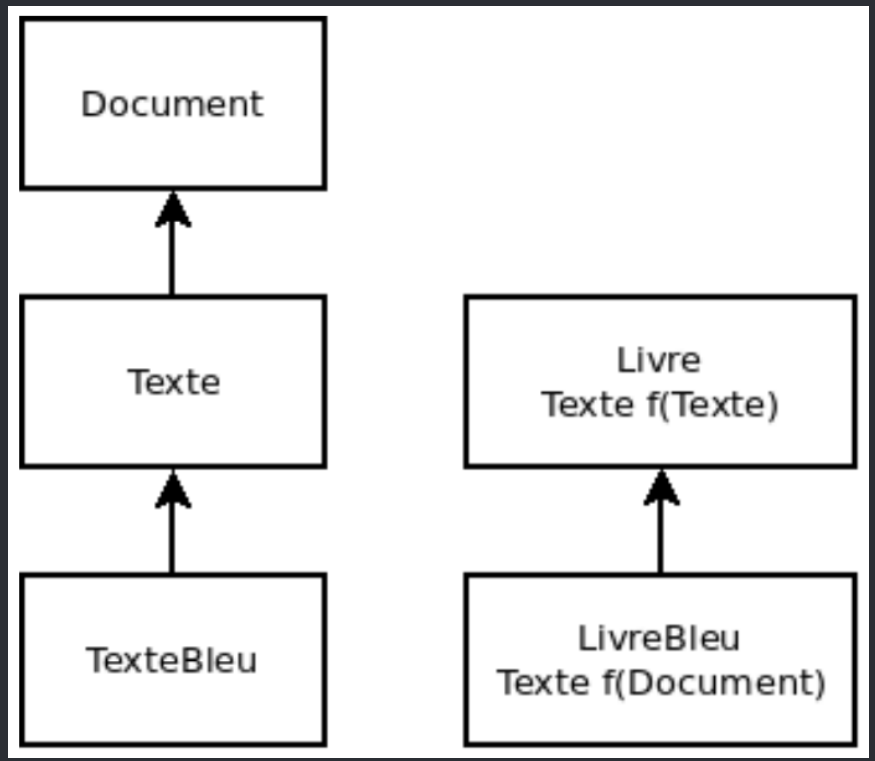
Si Java acceptait une telle redéfinition, le typage ne serait pas sûr car la méthode spécialisée f ne saurait que faire d'un appel avec un objet TexteRouge héritant de Texte:

```
LivreBleu lb = new LivreBleu();
TexteRouge texteRouge = new TexteRouge();
Texte t = lb.f(texteRouge);
```

Variance et héritage

Paramètre contravariant

Contrairement à l'intuition qui pousserait à vouloir considérer un paramètre comme covariant, il est plus sûr d'astreindre un paramètre à la contravariance. Ainsi, une méthode redéfinie doit recevoir en paramètre un type plus général que sa classe parente. Si l'on suppose que Texte hérite de Document, on pourrait donc vouloir faire:



Variance et héritage

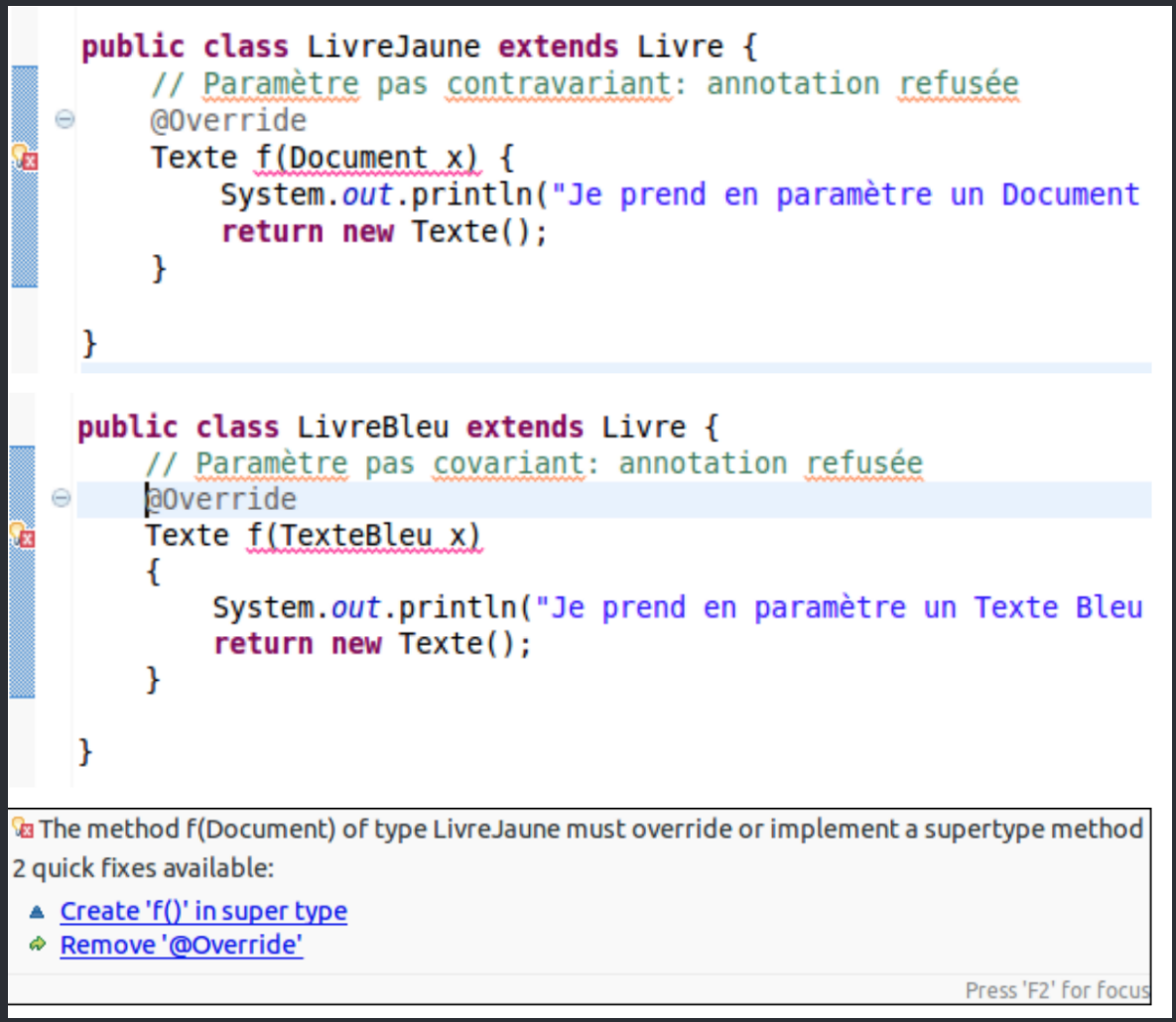
Paramètre contravariant

Configuration qui n'est en général pas vraiment ce qui est recherché par un développeur. De plus, l'ajout de la contravariance au langage pose de sérieux problèmes quand on le combine à l'overloading SOC. Du coup, le langage Java n'utilise ni la covariance ni la contravariance pour un paramètre: il doit être invariant pour surcharger la méthode.

Variance et héritage

Paramètre contravariant et @Override

Et effectivement, l'utilisation de l'annotation @Override combinée à un paramètre contravariant, et à fortiori covariant, ne plait pas du tout au compilateur:



Variance et héritage

Overloading (surcharge)

Sans avoir utilisé l'annotation @Override on peut croire, à tort, avoir réussi à redéfinir une méthode alors qu'en réalité, on l'a surchargée. Suivant le paramètre d'appel, la machine virtuelle choisit l'une ou l'autre des méthodes. Par exemple:

```
public class LivreJaune extends Livre {
// Paramètre pas contravariant: annotation refusée
```

```
//@Override:
Texte f(Document x)
{
        System.out.println("Je prend en paramètre un Document et retourne un Texte.");
        return new Texte();
}}
```

surcharge la méthode f du LivreJaune. Le Main suivant le démontre:

```
LivreJaune lj = new LivreJaune();
Texte texte = new Texte();
Texte t = lj.f(texte);
Document d = new Document();
Texte t2 = lj.f(d);
```

```
Je retourne un Texte d'un Livre.
Je prend en paramètre un Document et retourne un Texte.
```

Le même comportement a lieu pour un paramètre covariant. Le choix de java est l'invariance pour les paramètres d'appel au profit de l'utilisation libre de l'overloading.

Variance et héritage

Variance avec un générique

La gestion des génériques permet de contourner le problème de la variance de paramètres. Bien que non totalement équivalent, l'exemple suivant montre comment spécialiser une méthode d'une sous classe générique.

```
public class LivreToken<T extends Texte> {
Texte f(T x) {
        System.out.println("Je retourne un Texte d'un Livre.");
        return x;
}}
public class LivreMauve extends LivreToken<TexteMauve> {
Texte f(TexteMauve x) {
        System.out.println("Je retourne un Texte Mauve d'un Livre.");
        return x;
}}
```

On peut alors faire:

```
LivreMauve lm = new LivreMauve();
TexteMauve texteMauve = new TexteMauve();
Texte t2 = lm.f(texteMauve);
```

Mais pas :

```
LivreMauve lm = new LivreMauve();
Texte texte = new Texte();
Texte t = lm.f(texte);
// MainMauve.java: error: no suitable method found for lm.f(Texte)
```

Outline

Comparable

Java utilise l'interface Comparable pour permettre de comparer des objets et grâce à cette comparaison, d'appliquer des algorithmes de tri, notamment dans les collections triées. L'interace Comparable ne possède qu'une seule méthode:

```
public interface Comparable<T> {
  int compareTo(T o)
}
```

Mais de nombreuses classes (Integer, Character, Calendar, URI, etc.) l'implémentent. La méthode compareTo doit donner l'implémentation de la comparaison (anticommutative, transitive, symmétrique pour les exceptions) de deux objets instances d'une même classe:

```
public class LivreComparable implements Comparable<LivreComparable> {
  @Override
  String auteur = "";
  String titre = "";
  public int compareTo(LivreComparable o) {
    System.out.println("Je me compare à un Livre Comparable");
    if (! auteur.equals(o.auteur))
      return auteur.compareTo(o.auteur);
    return titre.compareTo(o.titre);
}}
```

Il devient alors très facile de trier une collection:

```
LivreComparable l2 = new LivreComparable();
l2.auteur = "Stroustrup B."; l2.titre = "Le langage C++";
LivreComparable l3 = new LivreComparable();
l3.auteur = "Puybaret E."; l3.titre = "Les Cahiers du Programmeur Java";
LivreComparable l4 = new LivreComparable();
l4.auteur = "Lalande J.-F."; l4.titre = "POO avancée";
ArrayList<LivreComparable> a = new ArrayList<LivreComparable>();
a.add(l2); a.add(l3); a.add(l4);
System.out.println(a);
Collections.sort(a);
System.out.println(a);
```

```
[Stroustrup B. - Le ..., Puybaret E. - Les..., Lalande J.-F. - POO...]
Je me compare à un Livre Comparable
Je me compare à un Livre Comparable
Je me compare à un Livre Comparable
[Lalande J.-F. - POO..., Puybaret E. - Les..., Stroustrup B. - Le ...]
```

Comparable

Héritage et Comparable

Lorsqu'une classe hérite d'une classe implémentant l'interface Comparable, les paramètres de compareTo restent invariants. Il faut donc pouvoir se comparer au type du parent. Bien souvent, l'appel à super suffit (en fait, on ne réimplémente même pas la méthode puisqu'on l'a par héritage):

```
public class LivreRougeComparable extends LivreComparable {
      @Override
      public int compareTo(LivreComparable o) {
              return super.compareTo(o);
      }}
```

On peut tout de même s'amuser à spécialiser la méthode, par exemple:

```
@Override
public int compareTo(LivreComparable o) {
        if (o instanceof LivreRougeComparable)
                return super.compareTo(o);
        else
                return 1;
}
```

Outline

Autoboxing - Unboxing

L'autoboxing désigne la capacité du compilateur à ajouter au code source ce qui manque pour manipuler un type simple sous forme d'une référence. Cette capacité n'existe que depuis la version 5 de Java:

```
Integer x = new Integer(5);
Integer y = 4;
```

```
// ou bien même:
Integer z = x + y - 3 * x;
```

L'autoboxing est possible à d'autres endroits qu'une initialisation d'une référence, par exemple lors d'appels à une méthode d'ajout d'une collection:

```
Vector<Integer> v = new Vector<Integer>();
v.add(y);
v.add(8);
```

Autoboxing - Unboxing

Inversement, l'unboxing permet de repasser au type primitif:

```
Integer x = new Integer(5);
int u = x;
```

Attention à ne pas autoboxer dans un Object, car le compilateur refusera d'unboxer:

```
Object o = 4;
System.out.println("o=" + o); // ok, imprime 4.
int o2 = o; // error: incompatible types, required: int, found: Object
```

Outline

JNI

Java offre la possibilité de faire des appels natifs. Du côté du code java, on charge la librairie de manière statique et on déclare une méthode native:

```
public class TestNatif {
      static { System.loadLibrary("jfllib"); }

      private native void test(String name);

      public static void main(String[] args) {
              TestNatif t = new TestNatif();
              t.test("JFL");
      }}
```

On génère le .h correspondant:

```
javah TestNatif
```

Ce qui produit le fichier d'entête TestNatif.h contenant:

```
JNIEXPORT void JNICALL Java_TestNatif_test
(JNIEnv *, jobject, jstring);
```

JNI

Du côté du C

On doit ensuite implémenter le code C de l'appel natif dans TestNatif.c:

```
#include <jni.h>
#include <stdio.h>
#include <TestNatif.h>

JNIEXPORT void JNICALL Java_TestNatif_test (JNIEnv * env , jobject o, jstring s)
{
  const char *name= (*env)->GetStringUTFChars(env,s,0);
  printf("String %s\n", name);
}
```

Puis compiler la librairie:

gcc -shared -o libjfllib.so TestNatif.c -I/usr/lib/jvm/java-7-oracle/include -I/usr/lib/jvm/java-7-oracle/include -I/u

Et on peut enfin exécuter le code java:

```
java TestNatif
String JFL
```