Polymorphisme et génériques

LU3IN002 : Programmation par objets

L3, Sorbonne Université

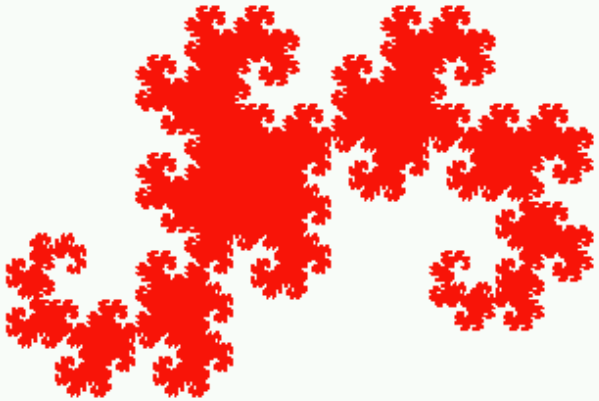
`https://moodle-sciences-23.sorbonne-universite.fr`

Antoine Miné

Cours 7

18 octobre 2023

Année 2023–2024



SORBONNE UNIVERSITÉ

LIP6

Plan du cours

- Cours 1, 2 & 3 : Introduction et bases de Java
- Cours 4 : Collections, itérateurs
- Cours 5 : Exceptions, tests unitaires
- Cours 6 : Design patterns I : Design Patterns structurels
- Cours 7 : Polymorphisme
- Cours 8 : Design patterns II : Design Patterns comportementaux
- Cours 9 : Interfaces graphiques (JavaFX)
- Cours 10 : Design patterns III : Design Patterns créationnels
- Cours 11 : Aspects fonctionnels de Java (lambdas)

Aujourd'hui :

- surcharge de méthodes ;
- types génériques ;
- classes internes, locales et anonymes.

Polymorphisme : motivation

But : écrire du code réutilisable

Java offre plusieurs techniques de polymorphisme :

- **Polymorphisme d'inclusion**
  redéfinition d'une méthode de même signature : *overriding*
  décision à l'exécution, en utilisant le type dynamique ;
- **Polymorphisme de surcharge**
  définition d'une méthode de signature différente : *overloading*
  décision à la compilation, en utilisant le type statique ;
- **Polymorphisme paramétrique**
  types génériques : `List<String>`
  aucune décision, mais offre des garanties de sûreté grâce au typage.

Points délicats : il faut comprendre

- ce qui est autorisé, interdit, garanti par les règles de typage ;
- quel code est effectivement exécuté par un appel de méthode ;
- l'interaction entre surcharge et redéfinition ;
- la différence entre résolution statique et résolution dynamique.

Surcharge et redéfinition

Rappel : type statique

exemple

```
Object m1(Object x) {
    String s = x.toString();
    return s + "toto";
}
```

exemple (suite)

```
String m2(Integer y) {
    Object t = m1(y);
    return (String)t;
}
```

Toute variable, attribut, argument a un **type statique**, donné lors de la déclaration.
(pour `x`, `t` : `Object` ; pour `s` : `String` ; pour `y` : `Integer`)

Toute expression Java a également un type statique,
déduit par le compilateur par propagation dans l'arbre d'expression :

- pour une expression réduite à une variable : le type de la variable (`x : Object`)
- pour un appel de méthode : le type de retour de la méthode (`m2(y) : String`)
- pour une conversion (cast), le type destination (`((String)t : String`)
- les opérateurs arithmétiques ont des règles un peu plus complexes.
  (e.g. : + choisit le type numérique le plus général de ses deux arguments, ou `String`)

Le compilateur vérifie la compatibilité des types statiques lors de toutes les copies :
affectations, passages de paramètre à une méthode, retours de méthode.

Surcharge

exemple

```
class A {
    int    m()          { return 2; }
    Object m(String s)  { return "A"; }
    String m(Integer i) { return "B"; }
}
```

exemple (suite)

```
A a = ...;
a.m();                  // 2 : int
a.m("toto");            // "A" : Object
a.m(new Integer(12));   // "B" : String
```

Définition de méthodes surchargées :

Il est possible de définir plusieurs méthodes différentes avec le même nom.

Chacune doit avoir une signature unique : nombre et type des arguments.

nous avions déjà vu cette possibilité pour les constructeurs ; ceci s'étend aussi aux méthodes

Résolution des appels de méthode :

Lors d'un appel de méthode, le nombre et le type statique des expressions passées en argument déterminent la méthode appelée.

La résolution est statique : elle est effectuée par le compilateur une fois pour toutes et ne varie pas à l'exécution.

Le type de retour n'entre pas en ligne de compte dans le choix de la méthode appelée.

Conversion implicite et ambiguïté de surcharge

exemple correct

```
String concat(String x, String y)
{ return x+y; }

String concat(Object x, Object y)
{ return concat(x.toString(), y.toString()); }

concat("1", "2");  concat("1", new Integer(2));
```

exemple d'ambigüté

```
double add(double x, double y)  { return x+y; }

double add(double x, int y)  { return x+y; }

double add(int x, double y)  { return x+y; }

add(1.0, 2.0); add(1, 2.0); add(1, 2);
```

Rappel : conversion implicite lors d'un appel

Une méthode peut être appelée avec des expressions de types différents de ceux des arguments, si les arguments ont un type plus général ; par exemple :

- passage d'un `int` où un `double` est attendu ;
- passage d'une expression d'une classe là où une classe parent est attendue ;
- passage d'une expression d'une classe là où une interface implantée est attendue.

En cas de surcharge, plusieurs méthodes peuvent donc devenir éligibles.

- Java choisit la méthode la plus spécifique, i.e., avec le moins de conversions ;
- si plusieurs méthodes sont éligibles avec le même nombre minimal de conversions, c'est une erreur de compilation (le compilateur élimine statiquement les ambiguïtés).

Auto-boxing

Rappel : Java distingue :

- les types primitifs : `int`, `double`, etc.
- les types objets : `Object`, `String`, classes utilisateurs, etc.

Pour être vraiment polymorphe, il est utile d'unifier ces deux catégories.

$\Longrightarrow$ Java associe une classe à chaque type primitif :

`Integer`, `Double`, `Boolean`, `Character`

c'est la version dite « emboîtée » (*boxed*), plus coûteuse à représenter, mais qui peut donc être stockée dans une variable `Object` (→ polymorphisme) ; les objets de ces classes sont immuables (comme les chaînes `String`).

Exemple d'utilisation : les listes d'entiers `List<Integer>`.

Conversions :

- d'instance vers primitif : `i.intValue()` ;
- de primitif vers instance : `Integer.valueOf(12)`
  (plus efficace que `new Integer(12)`).

Java insère ces conversions automatiquement si nécessaire lors des affectations, des appels et retours de méthodes.

e.g. : `List<Integer> l; l.add(2);` `int x; for (Integer i : l) x += i;`

Rappel : redéfinition et liaison dynamique

fournisseur

```
class A {
     void m() { System.out.println("X"); }
     void n() { m(); }
}

class B extends A {
    @Override
    void m() { System.out.println("Y"); }
}
```

client

```
void f(A obj) {
    // type statique de obj : A
    // type dynamique : A ou B
    obj.m();
    obj.n();
}

f(new A()); // affiche X X
f(new B()); // affiche Y Y
```

Redéfinition :

Une classe dérivée peut redéfinir une méthode :

nouveau corps, mais signatures identiques (nombre et type des arguments).

le type de retour peut être identique, ou bien plus spécifique (classe dérivée)

$\Longrightarrow$ à bien distinguer de la surcharge (même classe, signatures différente) !

Résolution de l'appel de méthode : liaison tardive, dynamique

Le type dynamique de l'objet, spécifié par `new`, indique la méthode réellement appelée, y compris quand la méthode est appelée sur `this` dans la classe parent.

Interaction entre surcharge et redéfinition

fournisseur

```
class A {
    void m(Object y) { System.out.println("X"); }
}

class B extends A {
    @Override
    void m(Object y) { System.out.println("Y"); }

    void m(String y) { System.out.println("Z"); }
}
```

client

```
B d = new B();
d.m(new Integer(1));  // affiche Y
d.m("toto");          // affiche Z

A c = new B();
c.m(new Integer(1));  // affiche Y
c.m("toto");          // affiche Y
```

Une classe peut à la fois :

- redéfinir une méthode existante du parent, avec la même signature ;
- et surcharger la méthode existante, avec une signature différente.

Pour déterminer la méthode réellement appelée par `obj.m(e1,...,eN)`, il faut :

1. déduire la signature de la méthode `m` appelée
   en utilisant le type statique de `obj` et des arguments `e1`, ... `eN`,
2. puis retrouver cette signature dans la classe correspondant au
   type dynamique de l'objet référencé par `obj`.

Surcharge et redéfinition : renommage des méthodes

fournisseur

```
class A {
    void m_Object(Object y) { System.out.println("X"); }
}

class B extends A {
    @Override
    void m_Object(Object y) { System.out.println("Y"); }

    void m_String(String y) { System.out.println("Z"); }
}
```

client

```
B d = new B();
d.m_Object(new Integer(1));  // affiche Y
d.m_String("toto");          // affiche Z

A c = new B();
c.m_Object(new Integer(1));  // affiche Y
c.m_Object("toto");          // affiche Y
```

Pour s'aider, imaginer que les méthodes surchargées ont des noms différents, par exemple en intégrant le type des arguments au nom de la méthode :

- cette manipulation correspond à la résolution statique des signatures ;
  $\Longrightarrow$ la signature choisie est figée dans le `.class` et ne peut pas dépendre de l'exécution
- il ne reste plus qu'à résoudre la liaison dynamique, à l'exécution.

Ce renommage est d'ailleurs fait implicitement par le compilateur !

`m(Object)` et `m(String)` sont en fait `m[Ljava.lang.Object` et `m[Ljava.lang.String`
Cf. documentation de `getName` dans `java.lang.Class`.

Types génériques

Motivation : les collections avant les types génériques

fournisseur

```
public interface List {
    public boolean add(Object e);
    public Object  get(int index);
}
```

client

```
List l = new ArrayList();
l.add("toto");
l.add(new Integer(12));

String x = (String)l.get(0);
String y = (String)l.get(1); // ClassCastException
```

Les collections Java peuvent contenir des objets arbitraires.
(y compris panacher des objets de classe différente)

Avec le système de types objet « classique » (avant Java 5) :

- `add` a un argument de type `Object` ;
- `get` retourne un type `Object`.

Même si `add` ajoute un objet `String`, `get` le retournera avec le type `Object`

$\Longrightarrow$ le client doit faire une conversion explicite : `(String)l.get(...)` ;
qui échoue à l'exécution si on s'était trompé de type d'objet dans `add`.

Nous aimerions plutôt exprimer la propriété : `l` est une liste de `String` :
`add` est toujours appelée avec des `String`, donc `get` retourne toujours une `String`.

Collections avec types génériques

fournisseur

```
public interface List<T> {
    public boolean add(T e);
    public T       get(int index);
}
```

client

```
List<String> l = new ArrayList<String>();
l.add("toto");
l.add(new Integer(12)); // Erreur de compilation
String x = l.get(0);
String y = l.get(1);
```

Pour toute classe `T`, `List<T>` correspond aux listes contenant des objets de type `T`.

`List<String>` spécialise le type au cas `T` = `String` : liste de chaînes.

La signature de `List<String>` devient, par remplacement de `T` par `String` :

- `public boolean add(String e)`
  seuls des objets `String` peuvent être stockés dans la liste,
  sinon, une erreur est signalée à la compilation ;
- `public String get(int index)`
  `get` a pour type de retour `String`,
  conversion de type inutile, pas de `ClassCastException` possible.

$\Longrightarrow$ le système de types assure statiquement que `l` ne contient que des `String` !

Notes : – `add(T e)` accepte un objet de classe `T` ou dérivée.
– `List<Object>` permet de retrouver le comportement de `List` d'antan, qui accepte tout objet.

Définition d'une classe générique utilisateur

exemple

```
public class NamedObject<A,B> {
    private A       obj;
    private String  name;
    private List<B> properties = new ArrayList<B>();

    public NamedObject(String name, A obj)
    { this.obj = obj; this.name = name; }

    public void addProperty(B p)  { properties.add(p); }
    public A    getObject()       { return obj; }
}
```

Forme générale : `class MaClasse<T1,...,TN> { ...}`

Également possible pour les interfaces et les classes abstraites.

`T1`, ..., `TN` sont des **paramètres de type**, de nom choisi par le programmeur :

- listés en tête de déclaration `<T1, ..., TN>` ;
- puis utilisés à la place de noms de classe dans le corps de la classe définie.

Le corps de la classe doit être bien typé pour tout choix des arguments de type ;
sinon, il y a une erreur lors de la compilation (assure la généricité du code).

À chaque utilisation du type générique, un **argument de type** est spécifié pour chaque paramètre de type (e.g., `NamedObject<Integer,String>`).

Exemples : classe d'arbre et interface actionneur

Applyer.java

```
public interface Applyer<T>
{
    public void apply(T value);
}
```

TreeNode.java

```
public class TreeNode<A>
{
    private A value;
    private List<TreeNode<A>> children = new ArrayList<TreeNode<A>> children;

    public TreeNode(A value) { this.name = value; }

    public A getValue() { return value; }

    public void addChild(TreeNode<A> child)  { children.add(child); }

    public void apply(Applyer<A> f) {
        f.apply(value);
        for (TreeNode<A> a : children) {
            a.apply(f);
        }
    }
}
```

Héritage de génériques, implantation d'interfaces

Triple.java

```
public class Triple<A,B,C>
{
    private A a;
    private B b;
    private C c;
    public Triple(A a, B b, C c)
    { this.a = a; ... }
}
```

NamedPair.java

```
public class NamedPair<X>
extends Triple<String,X,X>
{
    public NamedPair(String name, X x)
    { super(name, x, x); }
}
```

Une classe dérivant d'une classe générique : `extends Triple<A,B,C>`
doit préciser un type pour chaque paramètre de la classe parent `A`, `B`, `C` :

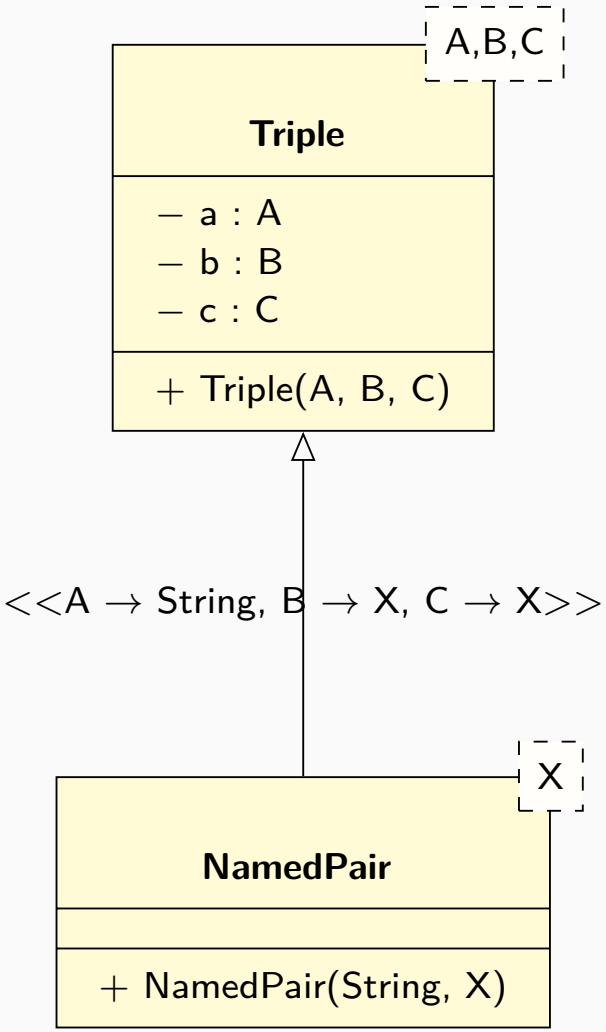
- soit une classe réelle (comme `String`) ;
- soit un paramètre de type de la classe dérivée, si elle est également générique.

Une classe dérivant d'une classe générique :

- n'a pas forcément les mêmes paramètres de type ;
- n'est pas forcément générique ;
- peut fixer la valeur de certains paramètres de type du parent.

Idem pour une interface implantée par une classe générique, ou héritée par une interface générique.

Génériques en UML



- les paramètres de type apparaissent dans une boîte, en haut à droite ;
- la relation d'héritage précise la valeur des paramètres.

Inférence des arguments des types génériques

Dans une déclaration de variable de type générique,
il est souvent inutile de répéter les arguments de type lors de l'initialisation.

En indiquant <>, Java tente d'inférer ces arguments automatiquement
grâce au contexte (membre gauche d'une affectation, type dans une déclaration de variable, etc.).

exemple d'inférence

```
List<String> l = new ArrayList<>();
//             = new ArrayList<String>() inféré
```

En cas d'échec de l'inférence, une erreur de compilation se produit.

En théorie, l'inférence a ses limites, mais il est difficile de trouver un exemple simple où l'inférence échoue !

Génériques et types bruts

À tout type générique `C<T1,...,TN>` correspond un type brut (*raw*) `C`,
sans paramètre de type (assure la compatibilité avec Java sans génériques)

- `List<String>` peut être converti en `List`
  mais on perd des garanties de typage, il faut donc à nouveau convertir chaque résultat de `get`...
- conversion de `List` en `List<String>` autorisée
  mais avec un avertissement du compilateur car dangereuse

```
List l = new ArrayList(); // l de type List brut
l.add(new Integer(12));   // l peut contenir des Integer
List<String> ll = l;      // compilé en : List ll = l, avec alerte à la compilation
String s = ll.get(0);     // compilé en : String s = (String)(ll.get(0))
                          // => ClassCastException à l'exécution !
```

La compilation des génériques introduit des casts qui sont normalement invisibles (n'échouent jamais)...
sauf si le système de types est détourné, d'où l'avertissement
$\Longrightarrow$ à éviter !

Note : ne pas confondre

- `List<String> l = new ArrayList<>()` : inférence du type générique ;
- `List<String> l = new ArrayList()` : conversion depuis un type brut.

Effacement de type

La machine virtuelle Java n'a pas de notion de type générique à l'exécution ;
ceux-ci ne sont visibles qu'au niveau du compilateur (pour la vérification de type),
puis effacés dans le code-octet (remplacés par le type brut, pour des raisons de compatibilité).

Conséquences de l'effacement :

`C<A> o` devient `C o`, donc pas d'information sur le type `A` à l'exécution

- `o.getClass()` ne donne pas l'argument de générique `A` (seulement le type brut `C`) ;
- `o instanceof C<String>` est impossible, seulement `o instanceof C` ;
- `C<A> extends Exception` est impossible, pas d'exception générique ;
  `catch (C<String> a)` impossible (test de type nécessaire à l'exécution, similaire à `instanceof A`)
- `new List<String>[10]` est impossible ;
  un tableau doit connaître le type des éléments pour vérifier les affectations à l'exécution et signaler `ArrayStoreException`
- `new A()` et `new A[10]` sont impossibles (ne connaît pas le constructeur à appeler) ;
- impossible de surcharger une méthode en variant les arguments de générique :

```
public void add(List<String> s)
public void add(List<Integer> s)
```

  ils utilisent le même type brut `List` en interne, et ont donc la même signature dans la JVM

Méthodes génériques

méthode générique

```
class Util {
  public static <T> void addToList(T[] a, List<T> l)
  { for (T e : a) l.add(e); }
}
```

client

```
String[] a = new String[12];
List<String> l = new ArrayList<>();
Util.addToList(a, l);
```

Une méthode peut également être générique, indépendamment de la classe :

- syntaxe : `modificateurs <T1,...,TN> type méthode(arguments...)`
- les paramètres de type `T1`, ... `TN`, sont visibles dans le corps de la méthode, dans les types des arguments et le type de retour ;
- l'appel à la méthode générique devrait s'écrire :

  ```
  Util.<String>addToList(a,l);
  ```

  mais `Util.addtoList(a,l)` fonctionne aussi
  grâce à l'inférence des arguments de type par le compilateur.

La même syntaxe s'applique aussi aux constructeurs :

e.g. : `public <T> Point(T arg) { ... }`

(attention à la position du `<>` : ne pas confondre `public <T> List a()` avec `public List <T> a()`)

Sous-typage des génériques

Rappel :

Une classe ou interface `A` est un sous-type de `B`
si elle dérive de `B` par une séquence de relations `extends` et `implements`.

Toute expression peut être convertie implicitement en un sur-type :
e.g. : `Object o = new Integer(12); // Integer → Object`

Sous-typage des génériques

Règle similaire pour l'héritage des génériques :

- `ArrayList<T> implements List<T>` ;
- donc `ArrayList<String>` est un sous-type de `List<String>`.

exemple de conversion autorisée

```
ArrayList<String> a = new ArrayList<String>();
List<String>      l = a;    // correct
```

Sous-typage des génériques (suite)

Sous-typage et arguments de type :

La relation de sous-typage **ne passe pas aux arguments de type** :

- `String` est un sous-type de `Object` ;
- mais `List<String>` n'est pas un sous-type de `List<Object>`.

exemple de conversion interdite

```
List<String> a = new ArrayList<String>();
List<Object> o = a;    // erreur de compilation
```

Justification :

Si c'était autorisé, nous pourrions écrire :

```
List<String> a = new ArrayList<String>();
List<Object> o = a;
o.add(new Integer(12)); // Problème : a contient maintenant un Integer
String x = a.get(0);    // Erreur : a.get(0) n'est pas compatible avec String
```

Problème similaire à celui des tableaux, mais solution différente :
- tableaux : sous-typage autorisé statiquement, mais test de type dynamique nécessaire et `ArrayStoreExceptions` possible
- génériques : test de type dynamique impossible, sous-typage rejeté à la compilation car non vérifiable à l'exécution

Génériques bornés : `extends`

exemple

```
public class Adder<Cmp extends Number>
{
    public int add(Cmp x, Cmp y) {
        return x.intValue() + y.intValue();
    }
}
```

client

```
new Adder<Integer>();
new Adder<Double>();
new Adder<String>(); // erreur
```

La syntaxe `<A extends C>` impose une restriction sur un argument de type :

- le client ne peut utiliser comme argument qu'une sous-classe de `C`

  $\Longrightarrow$ ceci est vérifié à la compilation ;

- la classe générique peut donc exploiter cette information
  et appeler des méthodes de `C` sur des variables de type du paramètre `A`.

Fonctionne également pour indiquer que le paramètre doit implanter une interface,
e.g. : `<T extends Comparable<T>>`

Attention à la syntaxe exotique pour implanter plusieurs interfaces,
e.g. : `<T extends A & B & C>` où `A`, `B`, `C` sont des interfaces ou des classes.

Génériques bornés avec joker : `? extends`

fournisseur

```
void foo(List<? extends Number> list) {
    list.get(0).intValue();
}

void bar(List<Number> list) {
    list.get(0).intValue();
}
```

client

```
List<Integer> l = new ArrayList<>();
foo(l); // autorisé
bar(l); // interdit

List<Number> n = new ArrayList<>();
foo(n); // autorisé
bar(n); // autorisé
```

La signature `void foo(List<? extends Number> list)` signifie :

- pour chaque appel à `foo`,
- il existe une classe `C`, sous-type de `Number`, telle que
- si l'argument `list` a pour type `List<C>`,
- alors, le corps de la fonction est bien typé ;
- à charge au compilateur de savoir inférer `C` à chaque site d'appel

  sinon, erreur à la compilation.

$\Longrightarrow$ `List<Integer>` est un sous-type `List<? extends Number>`

donc : conversion implicite possible de `List<Integer>` en `List<? extends Number>`

Interprétation du joker ?

Le paramètre de type ? indique un paramètre de type anonyme.

Utile pour éviter les noms de type inutiles dans les signatures :

somme générique

```
public int sum(List<? extends Number> list) {
    int i = 0;
    for (Number n : list) i += n.intValue();
    return i;
}
```

```
sum(new List<Integer>()); // accepté
sum(new List<Double>());  // accepté
```

accepte tout objet `List<T>`, où `T` est un sous-type de `Number`.

Équivalent à `public <T extends Number> int sum(List<T> list)`, mais plus court !

$\Longrightarrow$ remplace avantageusement le sous-typage des paramètres de type.

Note : `<?>` signifie `<? extends Object>`.

Génériques bornés : super

ajout générique

```
void put(List<? super Integer> list) {
   for (int i = 0; i < 10; i++)
      list.add(i);
}
```

client

```
put(new List<Integer>); // accepté
put(new List<Object>);  // accepté
put(new List<Number>);  // accepté
```

La syntaxe `<? super C>` est la restriction de type inverse :
`C` doit être un sous-type du type réel de l'objet.

`List<? super Point>` signifie : « les types de listes qui acceptent des points »

- une liste d'objets peut contenir des points (« `Object super Point` » est vrai) ;
- une liste de points colorés ne peut pas contenir de points (« `PointColore super Point` » est faux).

$\Longrightarrow$ règle de « contravariance ».

`List<? super Integer>` est plus permissif que `List<Integer>`.
Il permet de passer des listes d'`Object`, ce que `List<Integer>` interdit.

On ne peut pas écrire `<T super ...>` ; `super` n'est utilisable qu'avec un joker ?

Classes internes, locales, anonymes

Classes internes

schéma

```
class Outer {
    ...
    class Inner {
        ...
    }
}
```

Un fichier `C.java` contient une classe principale, nommée `C`,
mais le corps de la classe peut aussi contenir des définitions de classes internes.

Différences entre une classe interne `Inner` et une classe séparée :

- `Inner` peut être cachée des autres classes du package (`private`) ;
- `Inner` a un accès privilégié à la classe externe `Outer` ;
  - une nstance de `Inner` a accès aux attributs et méthodes de `Outer`, même privés !
  - une instance de `Inner` est toujours liée à une instance de `Outer` ;
    l'instance de `Inner` se souvient de l'instance de `Outer` qui l'a créée avec `new Inner(...)`

$\Longrightarrow$ améliore l'encapsulation (encourage l'emploi de `private`).

Classes internes : exemple

exemple : gestion de chansons

```
public class Song {
    private List<Tag> tags = new ArrayList<Tag>();
    private class Tag {
        String key, value;
        Tag(String key, String value) {
            this.key = key;
            this.value = value;
            Song.this.tags.add(this);
        }
    }
    public Song(String title, String author) {
        new Tag("title", title);
        new Tag("author", author);
    }
}
```

- chaque chanson a une liste `tags` de paires *clé* / *valeur* ;
- la classe `Tag` est utilisée en interne pour gérer ces paires ;
- une méthode et un constructeur de `Tag` peuvent accéder à :
  - `this` : l'instance de courante de `Tag` ;
  - `Song.this` : l'instance de `Song` qui a créé `this`.

    $\Longrightarrow$ la classe `Tag` a un attribut caché pour stocker `Song.this` et appeler ses méthodes.

Note : quand il n'y a pas d'ambiguïté, le `this.` ou le `Song.this.` ne sont pas nécessaires.

Accès aux classes internes depuis l'extérieur

fournisseur

```
public class Env {
    private Entry[] entries;

    public class Entry {
        private String variable;
        private Object value;

        public Entry(String var, Object val)
        { variable = var; value = val; }

        public String getVariable() { return variable; }
        public Object getValue()    { return value; }
    }

    public Entry[] getEntries() { return entries; }
}
```

client

```
Env env = ...;
for (Env.Entry e
    : env.getEntries()
{
    e.getVariable();
}

...

Env.Entry x;
x = env.new Entry("a",12);
```

Accès à la classe interne non privée en dehors de la classe la définissant :

- `Env.Entry` : nom de la classe interne ;
- `e.getVariable()` : accès à une méthode, si `e` a pour type `Env.Entry` ;
- `env.new Entry(...)` : constructeur de la classe interne ;

  $\Longrightarrow$ il faut préciser l'instance `env` de `Env` associée à l'instance de `Entry` créée.

Classes internes et héritage

parent

```
public class Text {

    public class Letter {
        private char code;
        ...
    }

    public void add(Letter l) ...
    public Letter get() ...
}
```

descendant

```
public class ColoredText
extends Text {

    public class ColoredLetter
    extends Letter {
        private Color color;
        ...
    }

    public void add(ColoredLetter l)
    { super.add(l); }

    public ColoredLetter get()
    { (ColoredLetter)super.get(); }
}
```

Si une classe `ColoredText` hérite de `Text`,
et que la classe interne `Letter` est visible depuis `ColoredText`,

alors `ColoredText` peut définir une classe interne `ColoredLetter`
qui hérite de `Letter`.

Hériter de `Letter` dans une classe qui n'hérite pas de `Text` est une erreur !

`Letter` doit pouvoir accéder à une instance de classe compatible avec `Text`

Classes internes statiques

points ordonnables

```
class Point {
    private double x, y;

    static class Cmp implements Comparator<Point> {
        public int compare(Point a, Point b) {
            if (a.x > b.x) return 1;
            ...
        }
    }

    public static void sort(List<Point> list)
    { Collections.sort(list, new Cmp()); }
}
```

- une instance de `Cmp`, statique, ne référence pas d'objet `Point` ;
  $\Longrightarrow$ `Cmp` ne peut accéder qu'aux attributs et méthodes statiques de `Point` ;
  pas de `Point.this`
- mais `compare` a accès aux attributs privés de toutes les instances de `Point` ;
  $\Longrightarrow$ on peut écrire `a.x` dans `compare` pour accéder aux attributs de `a` ;
- une instance de `Cmp` est créée avec `new Cmp()`, dans `Point` ;
  et par `new Point.Cmp()` en dehors de `Point`.

Note : une classe interne statique peut avoir des méthodes statiques, mais pas une classe interne non-statique.

Classes locales

schéma

```
class Outer {
    ...
    type méthode(type arg1, ...) {
        type l1,l2,...;
        class Local {
            ...
        }
    }
}
```

Les classes locales sont des classes internes définies dans les méthodes.

Une classe locale n'est accessible que dans la méthode (pas de modificateur de visibilité).

Elle peut accéder :

- aux attributs et méthodes de la classe englobante (comme pour une classe interne) ; ils référencent l'objet courant `this` sur laquelle `méthode` est appelée ;
- aux arguments `arg1, ...` de la méthode ;
- aux variables locales `l1, ...` de la méthode (s'ils sont immuables !)

Classes locales : exemple

exemple

```
class Named {
    private String name;

    public Object withName(final Object o) {
        class Tmp {
            @Override public toString() { return name + ":" + o.toString(); }
        }
        return new Tmp();
    }

    public void printPoint(Point p) { System.out.println(withName(p)); }
}
```

- `withName` utilise en interne une classe locale `Tmp`, à usage privé, uniquement pour redéfinir la méthode `toString` ;
- la classe `Tmp` n'est pas visible en dehors de `withName` → elle retourne `Object` ;
- une instance de `Tmp` peut accéder à (`Named.this.`)`name` et à `o` ;
- l'instance de `Tmp` est retournée : elle échappe de la méthode `withName`, et elle continue à accéder à `name` et à `o` après le retour de méthode !
  ⟹ Java garde des références aux arguments et variables utilisées (`Named.this`, `o`) sous forme d'attributs cachés dans la classe locale.

Accès aux variables locales, mot-clé `final`

exemple

```
class Exemple {

    public void m(int nb) {
       final int max = (nb < 1) ? 1 : nb;

       class Vec {
            String[] vec = new String[max];
            public getMax() { return max; }
        }
        Vec v = new Vec();
        // max++ interdit, car max utilisée dans Vec
    }
}
```

Une variable locale utilisée dans la classe locale doit être `final` ;

$\Longrightarrow$ interdit toute modification de la variable après son initialisation.

Justification :

Java doit permettre à `Vec` d'accéder à `max` après le retour de `m`, et maintient donc une copie de `max` dans toute instance de `Vec`.

Mais Java ne maintient pas la cohérence entre les copies ; si une copie est modifiée, les autres copies ne sont pas mises à jour ;
$\Longrightarrow$ par soucis de cohérence, les modifications sont donc interdites.

De nos jours, le mot-clé `final` est optionnel car Java l'infère automatiquement,
mais il faut s'assurer que la variable est effectivement constante après l'initialisation.

Classes anonymes

classe locale

```
class Outer {
    void m() {
        class Loc extends C {
            public Loc(int a, int b)
            { super(a,b); }
            public void m() { ... }
        }
        C f = new Loc(1,2);
    }
}
```

classe locale anonyme équivalente

```
class Outer {
    void m() {
        C f = new C(1,2) {
            public void m() { ... }
        };
    }
}
```

Classe anonyme :

raccourci syntaxique pour une classe locale déclarée sans nom
puis utilisée une seule fois, dans un new.

Dans ce cas, inutile de nommer la classe ; il suffit de préciser :

- la classe parent, ou une interface implantée (`C`) ;
- les arguments du constructeur (`1,2`) ;
  une classe anonyme n'a pas de constructeur ; le constructeur de `C` est appelé, comme avec `super(1,2)`
- les attributs et méthodes de la classe (`public void m ...`).

Classes anonymes : exemple

points ordonnables

```
class Point {
    private double x, y;

    public static void sort(List<Point> list) {
        Collections.sort(
           list,
           new Comparator<Point>() {
                public int compare(Point a, Point b) {
                    if (a.x > b.x) return  1;
                    if (a.x < b.x) return -1;
                    if (a.y > b.y) return  1;
                    if (a.y < b.y) return -1;
                    return 0;
                }
           }
        );
    }
}
```

Ici, la classe anonyme implante une interface `Comparator<Point>` ;
elle accède aux attributs (privés) des instances de `Point`
pour définir une méthode d'ordre `compare`
utilisée directement dans le tri `Collections.sort`.

Fichiers `.class` des classes internes

Classes internes :

- un fichier `.java` peut donc contenir le source de plusieurs classes ;
- mais un fichier `.class` ne peut contenir que le binaire d'une seule classe !
  (restriction de la machine virtuelle Java : JVM)

$\Longrightarrow$ la compilation d'un `.java` peut donc générer plusieurs fichiers `.class`

Les classes internes n'ont de sens qu'au niveau du source Java,
pas au niveau du code-octet des `.class`.

Elles sont un artefact du compilateur, et transformées en classes normales,
via l'ajout de références cachées vers la classe englobante.

Une classe interne, locale ou anonyme a un nom `.class` dérivé de la classe englobante,
en utilisant `$` comme séparateur :

- `Inner$Outer.class` pour une classe nommée `Outer` dans `Inner` ;
- `Inner$1.class`, `Inner$2.class`, etc. pour des classes anonymes de `Inner`.

D'autres fichiers `.class` peuvent être générés, en particulier en cas de classes internes privées,
pour contourner les problèmes de visibilité et d'accès...
C'est un détail d'implantation...