Généricité Java : types paramétrés, bornes et effacement

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Position de la leçon et objectifs 1
2 1. Le problème avant les types paramétrés : collections à `Object`, transtypage, types bruts 2
3 2. Écrire une classe générique : `Couple<T1,T2>`, instanciation, inférence 4
4 3. Méthodes génériques et types bornés 5
5 4. Wildcards et convention PECS 6
6 5. Ce que le compilateur efface : les types paramétrés n'existent pas à l'exécution 9
7 6. Pivot : les collections de base sont génériques 10
8 7. Généricité n'est pas substitution : `List<Chien>` n'est pas un `List<Animal>` 10
9 8. Types paramétrés et types primitifs : autoboxing dans `List<Integer>` 11
10 9. Varargs et types paramétrés (bonus — hors chemin noté) 13
11 10. Ce que vous devez pouvoir expliquer 13

1 Position de la leçon et objectifs

Cette leçon établit la mécanique des **types paramétrés** : écrire une classe générique, la borner, manier les wildcards, comprendre ce que le compilateur efface. Elle occupe la tête de la séquence sur les collections et en est le prérequis déclaré : la leçon `07_java_L_collections` enseigne l'usage des structures (hiérarchie, implémentations, contrats, parcours) et suppose acquis ce qui suit. Les exemples s'exécutent dans l'image du cours (Java 25, annexe `90_prerequis_environnement`).

Objectifs contractuels — à la fin de la leçon, l'étudiant sait :

1. **(GEN-O1)** Énoncer le défaut des collections à `Object` (contrôle perdu à la compilation, transtypage, erreur à l'exécution) ; reproduire une `ClassCastException` issue d'un type brut.

2. **(GEN-O2)** Écrire et instancier une classe générique à un et deux paramètres de type, avec inférence par `<>` ; prévoir le diagnostic d'une mauvaise instanciation.
3. **(GEN-O3)** Restreindre un paramètre par `extends` pour appeler les méthodes du supertype ; citer les deux formes de bornage.
4. **(GEN-O4)** Utiliser `? extends T` / `? super T` et énoncer PECS comme critère de choix en signature ; justifier le refus de `add` sur `List<?>`.
5. **(GEN-O5)** Énoncer l'effacement : deux instanciations distinctes partagent la même `Class` ; en déduire les interdits (`T.class`, `instanceof` paramétré, tableaux génériques).
6. **(GEN-O6)** Distinguer généricité et substitution : `List<Chien>` n'est pas un `List<Animal>` ; produire le contre-exemple compilé.
7. **(GEN-O7)** Expliquer `==` contre `equals` sur `Integer` par la plage de cache, sortie exécutée à l'appui.

2 1. Le problème avant les types paramétrés : collections à `Object`, transtypage, types bruts

Avant les types paramétrés, l'API de base (`List`, `Set`, `Map`) est définie sur `Object` : toute valeur y entre sans contrôle et n'en sort qu'en `Object`. Trois défauts s'ensuivent : aucune erreur de type détectable à la compilation, un transtypage explicite à chaque extraction, une hétérogénéité involontaire qui ne se manifeste qu'à l'exécution.

> ℹ Définition — type brut (raw type)
>
> Un **type brut** est une instance d'un type générique utilisée sans argument de type : `List l = new ArrayList();` déclare un `List` brut dont les éléments sont traités en `Object` ; aucun contrôle d'homogénéité, transtypage obligatoire à la lecture. (Source : JLS SE 25 §4.8 « Raw Types » — le nom d'une classe générique sans liste d'arguments de type associée.)

Décor réutilisé toute la leçon :

```
abstract class Animal { abstract String crier(); }

class Chien extends Animal {
    @Override String crier() { return "Le chien aboie."; }
    void sePromenerEnLaisse() { /* comportement propre au chien */ }
}

class Chat extends Animal {
    @Override String crier() { return "Le chat miaule."; }
}
```

```
import java.util.*;

List l = new ArrayList();   // type brut : aucun argument de type
l.add(new Chien());
l.add(new Chien());
l.add(new Chat());          // accepté en compilation — aucun contrôle
System.out.println("Taille de la liste : " + l.size());
```

```
Taille de la liste : 3
```

Sortie énoncée : `Taille de la liste : 3`. Premier défaut à l'exploitation : `l.get(0)` est de type statique `Object`.

```
l.get(0).crier();
```

```
CompilationException:
|   l.get(0).crier();
cannot find symbol
  symbol:   method crier()
```

Rejet énoncé : méthode `crier()` introuvable sur `Object`. Le transtypage rétablit le type, au prix d'un geste manuel à chaque extraction :

```
((Animal) l.get(2)).crier();
```

```
((Animal)l.get(2)).crier();: Le chat miaule.
```

Sortie énoncée : `Le chat miaule.` Second défaut, le plus coûteux : l'erreur différée à l'exécution.

```
((Chien) l.get(2)).sePromenerEnLaisse();
```

```
EvalException: class REPL.$JShell$28$Chat cannot be cast to class REPL.
$JShell$27$Chien (REPL.$JShell$28$Chat and REPL.$JShell$27$Chien are in unnamed
module of loader jdk.jshell.execution.DefaultLoaderDelegate$RemoteClassLoader
@672872e1)
------------------------------------------------------------------------
java.lang.ClassCastException: class REPL.$JShell$28$Chat cannot be
cast to class REPL.$JShell$27$Chien (REPL.$JShell$28$Chat and
REPL.$JShell$27$Chien are in unnamed module of loader
```

```
jdk.jshell.execution.DefaultLoaderDelegate$RemoteClassLoader @672872e1)
    at .(#36:1)
```

Rejet énoncé : `java.lang.ClassCastException` — l'objet d'index 2 est un `Chat`, le transtypage en `Chien` échoue à l'exécution alors que la compilation avait réussi. C'est précisément le déficit que les types paramétrés suppriment, en reportant le contrôle à la compilation.

3 2. Écrire une classe générique : `Couple<T1,T2>`, instanciation, inférence

> **i** Définition — classe générique, type paramétré
>
> Une **classe générique** est déclarée avec des **paramètres de type** entre angle brackets : `class Couple<T1, T2>`. Chaque paramètre est une variable type substituée à l'instanciation par un **argument de type** ; le type obtenu, `Couple<Chien, Chat>`, est un **type paramétré**. Les conventions usuelles notent `E` l'élément d'une collection, `T` un type quelconque, `K` et `V` les clés et valeurs d'une table d'association. (Source : JLS SE 25 §4.5 « Parameterized Types » — type paramétré et ses arguments de type.)

```
class Couple<T1, T2> {
    private final T1 premier;
    private final T2 second;

    Couple(T1 p1, T2 p2) { this.premier = p1; this.second = p2; }

    T1 getPremier() { return premier; }
    T2 getSecond()  { return second; }
}
```

Le corps utilise `T1` et `T2` comme des types ordinaires : le constructeur les reçoit, les accesseurs les restituent, sans transtypage à l'appel.

```
Couple<Chien, Chat> c = new Couple<>(new Chien(), new Chat());
System.out.println(c.getPremier().crier());
System.out.println(c.getSecond().crier());
```

```
Le chien aboie.
Le chat miaule.
```

Sorties énoncées : `Le chien aboie.` puis `Le chat miaule.` L'**inférence par `<>`** (le diamond) déduit les arguments de type de la variable déclarée : `new Couple<Chien, Chat>(...)` en toutes lettres est équivalent et redondant. La déclaration impose en retour sa contrainte à l'instanciation :

```
Couple<Chien, Chat> d = new Couple<>(new Chat(), new Chien());
```

```
CompilationException:
|   Couple<Chien, Chat> d = new Couple<>(new Chat(), new Chien());
incompatible types: cannot infer type arguments for Couple<>
    reason: inference variable T1 has incompatible bounds
      equality constraints: Chien
      lower bounds: Chat
```

Rejet énoncé : types incompatibles — `Chat` là où `Chien` est attendu, **dès la compilation** : la différence intégrale avec le §1, où la même hétérogénéité ne se déclarait qu'à l'exécution.

4 3. Méthodes génériques et types bornés

Une méthode peut déclarer ses propres paramètres de type, indépendamment de sa classe : c'est une **méthode générique**, introduite par la liste des paramètres avant le type de retour (`static <T> ...`). Un paramètre non borné s'instancie par n'importe quelle référence ; la borne `extends` le restreint.

> ℹ Définition — type borné
>
> Un **type borné** restreint les arguments admissibles d'un paramètre de type : `<T extends Number>` n'accepte que `Number` ou un de ses sous-types. Dans le corps, le compilateur traite `T` par `Number` : les méthodes du supertype sont accessibles sans transtypage. La borne s'écrit `extends` même pour une interface implémentée. (Source : JLS SE 25 §4.4 « Type Variables » — sans borne déclarée, une variable type se traite par défaut en `Object`.)

```
public <T extends Number> void afficher(T nombre) {
    System.out.println(nombre + " — en double : " + nombre.doubleValue());
}

afficher(42);     // Integer
afficher(3.5);    // Double
afficher(10L);    // Long
```

```
42 — en double : 42.0
3.5 — en double : 3.5
10 — en double : 10.0
```

Sorties énoncées : trois lignes affichant la valeur et sa conversion en `double`. `doubleValue()` est une méthode de `Number`, appelée sans transtypage ; sans la borne, seul `Object` serait disponible dans le corps.

```
afficher("quarante-deux");   // String n'est pas un sous-type de Number
```

```
CompilationException:
|   afficher("quarante-deux");   // String n'est pas un sous-type de Number
method afficher in class  cannot be applied to given types;
  required: T
  found:    java.lang.String
  reason: inference variable T has incompatible bounds
    upper bounds: java.lang.Number
    lower bounds: java.lang.String
```

Rejet énoncé : violation de la borne. Deux formes de bornage sont à distinguer : la **classe générique bornée** (borne sur le paramètre de la classe) et la **méthode générique bornée** (borne sur le paramètre de la méthode). La borne peut aussi contraindre un paramètre de classe ou d'interface :

```
interface Processor<T extends Runnable> {
    void process(T t);
}
```

`Processor` n'accepte comme argument de type que des classes exécutables : le contrôle est structurel, il porte sur le type et non sur une valeur. Un paramètre peut cumuler plusieurs bornes (`<T extends A & I>`, au plus une classe suivie d'interfaces) ; cette forme est hors du chemin noté et n'est pas réutilisée ici.

5 4. Wildcards et convention PECS

La borne d'un paramètre est une contrainte posée dans la déclaration du type ; le **wildcard** est une contrainte vue par l'appelant sur une signature : il désigne un argument de type inconnu, éventuellement borné.

> **Définition — wildcard**
>
> Le **wildcard** `?` désigne un argument de type inconnu. `? extends T` est une **borne supérieure** : le type réel est `T` ou un de ses sous-types. `? super T` est une **borne inférieure** : le type réel est `T` ou un de ses supertypes. `?` seul est non borné. Une signature en wildcard accepte plusieurs instanciations : `List<? extends Animal>` reçoit aussi bien `List<Chien>` que `List<Chat>`. (Source : JLS SE 25 §4.5.1 « Type Arguments of Parameterized Types » — wildcards et leurs bornes.)

Comme `List<Chien>` n'est pas un `List<Animal>` (§7), une méthode qui parcourt des animaux ne peut pas signer en `List<Animal>` : le wildcard rend la signature utilisable par toutes les instanciations compatibles.

```
public void afficherListe(List<? extends Animal> liste) {
    for (Animal animal : liste) {
        System.out.println(animal.crier());
    }
}

List<Chien> chiens = new ArrayList<>();
chiens.add(new Chien());
List<Chat> chats = new ArrayList<>();
chats.add(new Chat());

afficherListe(chiens);   // accepte List<Chien>
afficherListe(chats);    // accepte List<Chat>
```

```
Le chien aboie.
Le chat miaule.
```

Sorties énoncées : `Le chien aboie.` puis `Le chat miaule.` La borne supérieure permet la lecture en `Animal` ; elle interdit l'écriture d'un type précis, car le type réel de la liste reste inconnu de la méthode :

```
public void tenterLecture(List<? extends Animal> liste) {
    liste.add(new Chat());   // le type réel de liste pourrait être List<Chien>
}
```

```
CompilationException:
|     liste.add(new Chat());   // le type réel de liste pourrait être List<Chien>
incompatible types: Chat cannot be converted to capture#1 of ? extends Animal
```

Rejet énoncé : `add` refusé (sauf `null`) — l'accepter permettrait d'injecter un `Chat` dans une `List<Chien>`. Inversement, la borne inférieure autorise l'écriture du type borné :

```
public void ajouterChat(List<? super Chat> liste) {
    liste.add(new Chat());   // accepté : tout ? super Chat accepte un Chat
}

List<Animal> animauxCourts = new ArrayList<>();
ajouterChat(animauxCourts);
System.out.println("Après ajout : " + animauxCourts.size());
```

```
Après ajout : 1
```

Sortie énoncée : `Après ajout : 1`. En lecture, les éléments d'une `List<? super Chat>` ne sont garantis que `Object` : la borne inférieure privilégie l'écriture. Le wildcard non borné ne laisse aucune écriture possible :

```
public void tenterUnbounded(List<?> liste) {
    liste.add(new Chien());
}
```

```
CompilationException:
|       liste.add(new Chien());
incompatible types: Chien cannot be converted to capture#2 of ?
```

Rejet énoncé : `add` refusé sur `List<?>` — aucun type ne peut être garanti compatible avec l'argument inconnu.

! Convention — PECS

Producer Extends, Consumer Super. Quand la collection **produit** des valeurs que la méthode lit, écrire `? extends T` ; quand elle **consomme** des valeurs que la méthode y dépose, écrire `? super T`. Critère d'attribution vérifiable : examiner les opérations exercées sur le paramètre dans le corps de la méthode — uniquement des lectures → borne supérieure ; au moins une écriture du type borné → borne inférieure. (Source : Effective Java (Bloch, 3e éd.), Item 31 « Use bounded wildcards to increase API flexibility » ; base normative : JLS SE 25 §4.5.1.)

6 5. Ce que le compilateur efface : les types paramétrés n'existent pas à l'exécution

Les types paramétrés sont un dispositif de la **compilation**. Le compilateur vérifie les arguments de type, insère les transtypages implicites utiles, puis **efface** (type erasure) les paramètres : le bytecode ne conserve que le type brut et les bornes. L'effacement est ce qui a permis d'exécuter du code générique sur des JVM anciennes et d'étendre l'API progressivement — `List<Chien>` s'exécute comme un `List` brut, avec des contrôles en plus, pas comme une nouvelle classe d'exécution. Conséquence observable : deux instanciations distinctes d'un même type générique partagent une seule `Class`.

```
boolean memeClasse = new ArrayList<String>().getClass()
                   == new ArrayList<Integer>().getClass();
System.out.println("Même Class : " + memeClasse);
```

```
Même Class : true
```

Sortie énoncée : `Même Class : true`.

> ℹ Propriété — interdits déduits de l'effacement
>
> Puisqu'aucun argument de type ne subsiste à l'exécution : 1. `T.class` est illégal — aucun `Class` paramétré n'existe à consulter ; 2. `instanceof List<String>` est illégal — non autorisé au titre de la compatibilité de conversion vérifiée (JLS SE 25 §5.5) et de l'applicabilité du motif `instanceof` (JLS SE 25 §14.30.3) (réifiabilité assouplie depuis Java SE 16) ; 3. `new T[n]` est illégal — un tableau est réifié (son type d'élément existe à l'exécution) et contrôlerait une information effacée. (Source : JLS SE 25 §4.6 « Type Erasure » ; types réifiables : §4.7 ; tableaux génériques (`new T[n]`) : §15.10.1.)

```
Object o = new ArrayList<String>();
if (o instanceof ArrayList<String>) {
    System.out.println("jamais testable");
}
```

```
CompilationException:
|   if (o instanceof ArrayList<String>) {
java.lang.Object cannot be safely cast to java.util.ArrayList<java.lang.String>
```

Rejet énoncé : `instanceof` refusé sur un type paramétré — la conversion n'est pas compatible au sens de JLS SE 25 §5.5 (§14.30.3 pour le motif `instanceof`) ; l'argument de type, effacé, ne serait de toute façon pas testable à l'exécution.

7 6. Pivot : les collections de base sont génériques

Les types de `java.util` **sont** des types paramétrés : `List<Chien>` remplace le `List` brut du §1.

```
List<Chien> listeDeChiens = new ArrayList<>();
listeDeChiens.add(new Chien());
for (Chien chien : listeDeChiens) {
    System.out.println(chien.crier());
}
```

```
Le chien aboie.
```

Sortie énoncée : `Le chien aboie.` — sans transtypage à l'extraction. L'écriture qui menace l'invariant est refusée dès la compilation :

```
listeDeChiens.add(new Chat());
```

```
CompilationException:
|   listeDeChiens.add(new Chat());
incompatible types: Chat cannot be converted to Chien
```

Rejet énoncé : types incompatibles — là où le §1 produisait une `ClassCastException` à l'exécution. La syntaxe de la boucle améliorée vient de la séquence bases (sec-04) ; son mécanisme — `iterator()`, les traductions, la spécification enhanced-for — et le rôle des types paramétrés dans la hiérarchie `Collection` appartiennent à la leçon `07_java_L_collections` (§7).

8 7. Généricité n'est pas substitution : `List<Chien>` n'est pas un `List<Animal>`

Avec l'héritage, `Chien extends Animal` entraîne la substitution. Avec les types paramétrés, cette implication **ne se propage pas** : `List<Chien>` n'est pas un sous-type de `List<Animal>` — la relation entre instanciations est **invariante**.

```
List<Animal> animaux2 = listeDeChiens;   // refusé
```

```
CompilationException:
|   List<Animal> animaux2 = listeDeChiens;   // refusé
incompatible   types:   java.util.List<Chien>   cannot   be   converted   to
java.util.List<Animal>
```

Rejet énoncé : types incompatibles. Le refus est une nécessité de correction : supposons l'affectation admise ; l'alias `animaux2`, de type `List<Animal>`, accepterait un `Chat` (`animaux2.add(new Chat())` serait légal), et ce `Chat` se retrouverait dans `listeDeChiens`, dont la garantie du §6 prétendait l'exclure. Deux écritures chacune bien typées produiraient un conteneur violant l'une d'elles ; l'invariance est le prix de la consistance.

Le contournement légitime n'est pas un cast mais une **copie** : le constructeur de copie fonde une nouvelle liste, typée `Animal`, alimentée par l'existante.

```
List<Animal> animaux3 = new ArrayList<>(listeDeChiens);
animaux3.add(new Chat());
System.out.println("Taille de la liste d'animaux : " + animaux3.size());

for (Animal anim : animaux3) {
    System.out.println(anim.crier());
}
```

```
Taille de la liste d'animaux : 2
Le chien aboie.
Le chat miaule.
```

Sorties énoncées : `Taille de la liste d'animaux : 2`, puis une ligne `Le chien aboie.` et une ligne `Le chat miaule.` Les deux listes sont des objets distincts : y déposer un `Chat` dans l'une ne contredit plus la garantie de l'autre.

9 8. Types paramétrés et types primitifs : autoboxing dans `List<Integer>`

Un argument de type est nécessairement un type de **référence** : `List<int>` n'existe pas. Entre les primitifs et leurs classes wrapper (`int`/`Integer`, `double`/`Double`, etc.) le compilateur assure des conversions implicites.

Définition — autoboxing / unboxing

L'**autoboxing** convertit implicitement une valeur primitive en son instance wrapper (`int` → `Integer` par `Integer.valueOf(3)`) ; l'**unboxing** fait l'inverse (`Integer` → `int` par `intValue()`). Dans `List<Integer>`, `l.add(3)` box et `l.get(0) + 1` débox. La conversion est implicite à l'écriture, réelle en sémantique : l'objet est alloué. (Source : JLS SE 25 §5.1.7 « Boxing Conversion » et §5.1.8 « Unboxing Conversion » ; plage de cache : javadoc Java SE 25, `Integer#valueOf(int)` — met toujours en cache −128 à 127 inclus.)

```
List<Integer> nombres = new ArrayList<>();
nombres.add(3);   // autoboxing : int → Integer
nombres.add(4);

int somme = nombres.get(0) + nombres.get(1);   // unboxing des deux extraits
System.out.println("La somme est " + somme);
```

```
La somme est 7
```

Sortie énoncée : `La somme est 7`. Le point d'attention est l'égalité : sur des `Integer`, `==` compare les **références**, `equals` compare les **valeurs**. La javadoc de `Integer.valueOf(int)` garantit le cache des valeurs de −128 à 127 inclus : dans cette plage le boxing retourne des instances partagées, hors de la plage l'identité des instances n'est pas garantie.

```
Integer a = 127;
Integer b = 127;
Integer i1 = 128;
Integer i2 = 128;

System.out.println("==     sur 127 : " + (a == b));
System.out.println("equals sur 127 : " + a.equals(b));
System.out.println("==     sur 128 : " + (i1 == i2));
System.out.println("equals sur 128 : " + i1.equals(i2));
```

```
==     sur 127 : true
equals sur 127 : true
==     sur 128 : false
equals sur 128 : true
```

Sorties énoncées : `true`, `true`, `false`, `true`. Dans la plage de cache les deux variables désignent la même instance ; au-delà, deux boxing distincts produisent deux instances et `==` répond `false` alors qu'`equals` répond `true`. Par conséquent : sur des types wrapper, la comparaison d'égalité s'écrit `equals` (ou après unboxing explicite en `int`), jamais `==`.

10 9. Varargs et types paramétrés (bonus — hors chemin noté)

Un paramètre varargs est un tableau : `void f(int... ns)` équivaut à `void f(int[] ns)`. Appliqué aux types paramétrés, le tableau — réifié, son type d'élément existe à l'exécution — croise l'effacement : le compilateur ne peut plus garantir la conformité de ce qu'il reçoit.

```
@SafeVarargs
static <T> List<T> copie(T... elements) {
    return new ArrayList<>(Arrays.asList(elements));
}

System.out.println(copie("a", "b", "c"));
```

```
[a, b, c]
```

Sortie énoncée : `[a, b, c]`. Le risque a un nom, la **pollution de tas** (heap pollution) : une écriture dans le tableau, autorisée par le type réifié de l'un, invaliderait la lecture par le type effacé de l'autre. L'annotation `@SafeVarargs` atteste que la méthode n'écrit pas dans son varargs et masque l'avertissement `unchecked` ; elle engage la responsabilité de l'auteur, non celle du compilateur.

11 10. Ce que vous devez pouvoir expliquer

Énumération close, une ligne par objectif contractuel :

1. **(GEN-O1)** Ce qu'un type brut laisse passer jusqu'à l'exécution ; citer les rejets du §1 (`ClassCastException`, méthode introuvable).
2. **(GEN-O2)** Écrire `Couple<T1,T2>` de mémoire ; expliquer le `<>` de `new Couple<>(...)`.
3. **(GEN-O3)** Ce que `extends Number` rend appelable que `T` seul ne rendrait pas ; citer les deux formes de bornage.
4. **(GEN-O4)** Décider `? extends T` ou `? super T` en signature : énoncer PECS et son critère d'attribution (§4).
5. **(GEN-O5)** Ce qui subsiste de `List<String>` dans le bytecode ; en déduire les trois écritures interdites (§5).

6. **(GEN-O6)** Pourquoi `List<Animal> a = listeDeChiens;` est refusé alors que `Chien extends Animal` ; reproduire le raisonnement par l'absurde (§7).
7. **(GEN-O7)** Pourquoi `==` sur deux `Integer` de 128 répond `false` ; citer la source de la plage de cache (§8).

La mécanique est en place ; son application systématique à l'API — hiérarchie `Collection`, implémentations, contrats, parcours — constitue l'objet de la leçon `07_java_L_collections`.