Généricité

Programmation générique

La **programmation générique** consiste à écrire du code réutilisable pour des objets de types différents.

Avant la programmation générique

Avant l'ajout des classes génériques à Java, la **programmation générique** était réalisée à l'aide de l'héritage (tout classe hérite au moins de la classe $Object$).

- **Problèmes de cette approche**
  1. Nécessité de cast : Chaque récupération d'un élément nécessite un cast explicite :
     $$String\ filename = (String)\ files.get(0);$$
  2. Absence de vérification des types : On peut ajouter des objets de n'importe quelle classe, ce qui peut provoquer des erreurs d'exécution :
     $$files.add(new\ File("test.pdf"));$$
     $$String\ filename3\ =\ (String)\ files.get(1);$$ // Erreur à l'exécution !

La programmation générique : Une meilleure solution

Introduction des paramètres de type : Permettent de **spécifier le type des éléments** d'une collection.

- **Avantages**
  - Sécurité des types : Les erreurs de type sont détectées à la compilation, pas à l'exécution.
  - Le type des éléments est clair dès la déclaration.
    Ici, on sait immédiatement que $files$ ne contient que des $String$.
  - Réduction de la duplication de code : Une seule clase pour tout type.

L'opération diamond

Depuis Java SE 7, il est possible **d'omettre le type générique** dans le constructeur :

$$ArrayList < String > fichiers = new\ ArrayList <> ( );$$

Le type omis est déduit du type de la variable.
L'opérateur <> est appelé **opérateur diamond**.
Depuis Java 9, l'opérateur diamond est autorisé à la création d'une classe anonyme.

Remarques

- Le paramètre de type (ou type variable) est déclaré à côté du nom de la classe ;
- Une fois déclaré, il peut être utilisé dans la classe comme tout autre type ;
- Le paramètre T peut être n'importe quel identificateur, mais il est recommandé d'utiliser une lettre majuscule pour plus de lisibilité.

Utilisation de la classe générique $Couple < T >$

Java permet de simplifier la manipulation des types primitifs avec auto-boxing.

Sans auto-boxing :
$$Integer\ oi1 = new\ Integer(3);\quad Integer\ oi2 = new\ Integer(5);$$
$$Couple < Integer > ci = new\ Couple < Integer > (oi1, oi2);$$

Avec auto-boxing (depuis Java 5) :
$$Couple < Integer > ci = new\ Couple <> (3, 5);$$

Classe générique à plusieurs paramètres de type

- L'exemple précédent ne comportait qu'un seul paramètre de type ;
- Bien entendu, une classe générique peut en comporter plusieurs.

Types variables

Une classe générique peut définir plusieurs types variables, par exemple :

$NomClasse < V, K >$

Les noms des types variables sont arbitraires, mais la Java API suit les conventions suivantes :

- E : Pour les éléments d'une collection.
- K, V : Pour les clés et valeurs dans une collection indexée
- T, U, S : Pour des types génériques arbitraires.

Compilation du code générique

Compilation du code générique

Les types génériques (T) sont déclarés dans une classe ou une méthode.

Mais qu'en fait le compilateur ?

Dans le JDK 5.0, les concepteurs ont opté pour un compromis :

1. **Maximiser les diagnostics à la compilation** (détecter les erreurs dès la compilation)
2. **Assurer la compatibilité avec les anciennes versions** de Java (mélange de code générique et non générique).

Ce compromis repose sur un mécanisme appelé **effacement des types** (type erasure).

Effacement des types

- Lorsqu'on utilise une classe générique avec un type spécifique (comme $Couple < String >$ ou $Couple < Integer >$), Java ne crée pas un type distinct dans le bytecode pour chaque invocation de type générique.
- Quand le code générique en Java est compilé, toutes les informations sur les types génériques spécifiés (par exemple $< T >$, $< E >$, $< K, V >$) sont effacées pour produire un bytecode compatible avec les versions de Java antérieur à Java 5.

Compilation d'une classe générique

- Lors de la compilation, la déclaration générique $< T >$ est supprimée, et tous les types T sont remplacés par $Object$.
- Les types génériques sont remplacés par leur type brut (raw type), généralement $Object$.

Explication du mécanisme d'effacement

- Une déclaration d'un type générique $C < T > \{\dots\}$ crée une seule classe $C\{\dots\}$ appelée « effacement » (erasure) de la classe générique ou bien classe « brute » (raw class) ;
- Ainsi, le type $Couple < T >$ est remplacé par un type brut nommé $Couple$;
- La machine virtuelle n'a pas de connaissance de la généricité ! Elle **ne manipule que les classes ordinaires** ;

- Le compilateur insère des castings automatiques pour maintenir la sécurité des types lors de l'exécution.

Pour garantir une utilisation de la classe brute qui respecte le type effectif, le compilateur introduit des casts.

- **Conséquence :**

Une classe générique ne peut pas être invoquée avec un type primitif.

La déclaration suivante serait rejetée : $Couple < int > c;$ // Erreur int n'est pas une classe

Après l'effacement, tous les éléments du type variable sont traités comme des instances de $Object$, ce qui implique qu'ils **doivent obligatoirement être des objets**.

Limitation des classes génériques

- On ne peut pas instancier un objet d'un type paramétré ;
- L'appel $new\ T(\ )$ est rejeté à la compilation ;
- Explication : A l'exécution, le type T sera effacé (remplacé par $Object$) ;
- La machine virtuelle ne peut pas déterminer le type exact de l'objet à l'instancier.

Remarque importante

Il ne faut pas confondre : L'instanciation d'un type paramétré au sein d'une classe générique avec l'instanciation d'un objet d'une classe générique :

$Couple < Double > couple = new\ Couple <> (\ldots);$

Ceci est autorisé et constitue la base de la programmation générique.

Conséquence de l'effacement = Restriction sur les membres statiques

Un type variable ne peut pas être utilisé dans un champ ou une méthode statique d'une classe générique.

Explication :

- Les membres statiques appartiennent à la classe et non aux instances ;
- Après l'effacement, il n'y a qu'une seule version de la classe A, quel que soit le type T ;
- La classe générique ne peut pas définir un membre statique pour chaque valeur possible de T.

Méthodes génériques

Méthodes génériques

Il est possible de définir des méthodes génériques (avec leurs propres paramètres de type) :

Exemple : Méthode statique permettant de tirer au hasard un élément d'un tableau fourni en argument, de type quelconque. [Generique2.java, class Methode]

Le type variable $< T >$ est déclaré après les modificateurs et avant le type de retour.

Remarque : Les méthodes génériques peuvent être définies aussi bien dans des classes normales que dans des classes génériques.

Méthode générique à deux arguments

Ce calcul est accepté bien que les deux arguments soient de type différent.
Pour imposer que les arguments soient du même type.

Méthode générique statiques

- Remarque : Les méthodes génériques **peuvent** être statiques car elles ne dépendent pas du type variable de la classe générique.
- Pourquoi ça fonctionne ?
  - Le type variable T est défini localement dans la méthode et non au niveau de la classe ;
  - Cela permet à la méthode d'être indépendante des types définis dans la classe.

Types bornés

Types bornés

- Un type variable $< T >$ peut représenter n'importe quel type (à l'exception des types primitifs) ;
- Cependant, lors de la déclaration de T, il est possible de spécifier des bornes pour **restreindre** les types admissibles comme valeurs pour T.

Mécanisme d'effacement

Lors de la compilation, le type T sera remplacé non plus par $Object$ mais par $Number$.

Limitation des paramètres de type, Cas des méthodes

La même démarche peut être appliquée à une méthode générique.
Après compilation, cette méthode est traduite comme si elle avait été écrite ainsi :

$static\ Number\ hasard(Number[\,]\ valeurs)\{\dots\}$

Remarques

Rappel : $public\ interface\ Comparable < T > \{\ int\ compareTo(T\ o);\ \}$

Restreindre T : $public\ static < T\ extends\ Comparable < T \gg T \min(T[\,]\ a)\ \{\dots\}$

- $T\ extends\ Comparable < T >$ signifie que T doit implémenter l'interface $Comparable$;
- Cela garantit que la méthode peut appeler $compareTo$ sur les objets de type T.

Remarque sur les types bornés

Un type variable $< T >$ peut être restreint (borné) par une classe ou une interface.

- $< T\ extends\ Classe >$: T doit être une sous-classe ou la classe elle-même ;
- $< T\ extends\ Interface >$: T doit implémenter l'interface ;
- $< T\ extends\ Classe\ \&\ Interface1\ \&\ Interface2 >$: T doit étendre la classe et implémente toutes les interfaces.

- On utilise $extends$ pour les classes et les interfaces ;
- Une seule classe peut être spécifiée comme borne ;

- Plusieurs interfaces peuvent être spécifiées (séparées par &) ;
- La classe doit toujours être déclarée avant les interfaces ;
- A l'intérieur de la classe/méthode qui déclare un type borné $< T\ extends\ C >$, les méthodes de C peuvent être utilisées sur les objets de type T ;
- Tout type variable est implicitement borné ($< T >$ équivalent à $< T\ extends\ Object >$).

Généricité et covariance

Les types génériques ne sont pas covariants.
Si A est un **sous-type** de B, alors $C < A >$ **n'est pas un sous type** de $C < B >$!

Covariance des tableaux

Contrairement aux collections génériques, les tableaux sont covariants.
Si A est un sous type de B, alors $A[\,]$ est un sous-type de $B[\,]$.

- **Risques et garanties**

  Java garantit l'intégrité des tableaux par un contrôle dynamique de type (à l'exécution).

Type anonyme ($Wildcard$)

Type anonyme ($Wildcard$)

- Le wildcard ? représente un type inconnu ;
- Il permet d'utiliser une classe générique sans préciser le type exact de ses éléments ;
- Utilisé lorsque le type concret n'a pas d'importance dans le contexte.

Compatibilité : N'importe quel type (sous-type de $Object$) peut être utilisé avec ?.

Wildcard (?) vs Type variable

Utiliser ? lorsqu'aucune autre méthode ou logique ne dépend du type exact.

- Afficher les éléments de la liste ne dépend pas du type exact des éléments, car ils peuvent tous être traités comme des $Object$;
- Le même effet peut être obtenu avec une variable de type dans ce cas précis.

Quand le $Wildcard$ ne fonctionne pas

Le wildcard (?) convient lorsqu'on ne se soucie pas du type exact.
Mais s'il existe une dépendance entre le type d'entrée et le type de sortie,
il faut utiliser une variable de type ($< T >$).

Si on utilisait un **wildcard** (?) à la place de T, cela ressemblerait à ceci (ce qui est invalide) :

$$public\ static\ ?[\,]\ permuter(?[\,]\ tab)\{\ \ldots\ \ return\ tab;\}$$

Problème : Le ? (wildcard) est anonyme et ne crée aucun lien entre les types.

- Le type du tableau passé en paramètre n'est pas clairement identifié ;
- Le type du tableau retourné ne peut donc pas être garanti identique à celui d'entrée.

Dans une méthode comme $permuter$, le type de retour doit dépendre directement du type du paramètre d'entrer.
Cela permet d'assurer que :

1. Si un $String[\,]$ est passé en argument, la méthode retourne un $String[\,]$;
2. Si un $Integer[\,]$ est passé en argument, la méthode retourne un $Integer[\,]$.

Déclaration de $Wildcard$

Un $Wildcard$ ne peut pas être **déclaré explicitement**.

$$ArrayList <?> l = new\ ArrayList < String > ();$$

$l.add("Bonjour");$ // Erreur : On ne peut pas ajouter d'éléments

$Object\ obj = l.get(0);$ // OK : On peut lire en tant qu'$Object$

- On peut lire des éléments, mais pas en écrire : Avec un $ArrayList <?>$, le type exact des éléments est inconnu.
- La méthode add est désactivé, car le compilateur ne peut pas garantir que le type de l'élément ajouté est compatible avec celui attendu par la liste.

Wildcards avec bornes

Les wildscards (?) peuvent également être bornés, mais avec des **restrictions** :

- Un seul type de borne est autorisé : Une borne supérieure ($extends$) ou une borne inférieure ($super$) mais **pas les deux à la fois** ;
- Pas de combinaison avec plusieurs bornes (par exemple, pas de & comme pour les types génériques classiques).

Wildcard avec borne supérieur ($extends$)

$List <? extends\ T >$

Un wildcard avec ? $extends\ T$ permet de spécifier une liste contenant des éléments de type T ou de tout type qui hérite de T.

Wildcard avec borne inférieure ($super$)

$List <? super\ T >$

Un wildcard avec ? $super\ T$ permet de spécifier une liste contenant des éléments de type T ou d'un type qui est une super-classe de T.

Wildcard et covariance

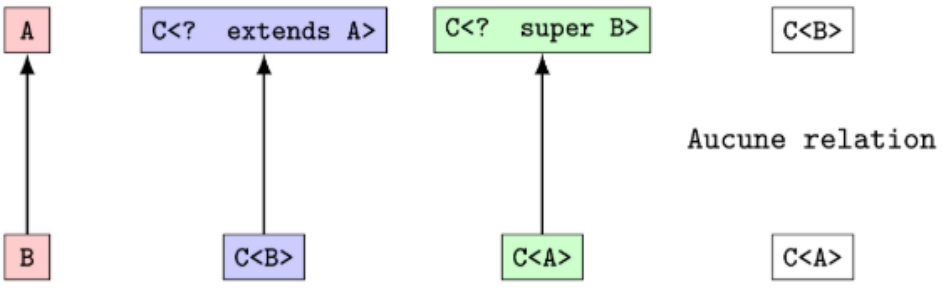
Problème de base : Les types génériques ne sont pas covariants par défaut.

- **Solution = Wildcard et covariance**

Le **wildcard** ? **permet de résoudre ce problème** en simulant la covariance entre types génériques.

Remarque : $ArrayList < Student >$ est un sous-type de $ArrayList <? extends\ Person >$ (covariance).

Wildcard et covariance



Capture de wildcard ($<? extends\ A >$)

Lorsqu'un wildcard est utilisé ($<? extends\ A >$), le compilateur considère la liste comme ayant un type inconnu, mais qui hérite de A.

Conséquences :

- Lecture possible : Les éléments peuvent être lus et assignés à une variable de type A ou une de ses super-classes ;
- Ecriture interdite : Impossible d'ajouter des éléments, car le type exact des éléments est inconnu.

Capture de wildcard ($<? \ super \ A>$)

Avec $<? super \ A>$, on spécifie une liste dont les éléments sont de type A ou une **super-classe** de A.

Conséquences

- Ecriture possible : On peut ajouter des éléments de type A ou de ses sous-classes ;
- Lecture restreinte : Les éléments peuvent être lus, mais uniquement comme $Object$.

Wildcard, covariance et intégrité des collections

Covariance — Avec $? \ extends \ Type$, les collections deviennent covariantes :

$ArrayList < String > l = new \ ArrayList <> ();$

$ArrayList <? \ extends \ Object > lo = l;$ // OK

Sécurité — Impossible de compromettre l'intégrité de la collection :

$lo.add(new \ Integer(5));$ // Erreur à la compilation

Récapitulatif = Capture et utilisation des wildcards

Wildcard	Lecture	Écriture
`<? extends A>`	Possible (type assignable à A)	Interdit
`<? super A>`	Restreinte (assignable à `Object`)	Possible (type A ou sous-type)

Règle générale

- Utiliser $<? \ extends \ A>$ pour lire des données ;
- Utiliser $<? \ super \ A>$ pour écrire des données.

Généricité et héritage

Généricité et héritage

En Java, il est possible d'hériter d'une classe générique ou d'implémenter une interface générique.

Exemple : Dans $java.util$:

$public \ class \ ArrayList < E > \ extends \ AbstractList < E > \ implements \ List < E >$

Lorsqu'on utilise une classe générique avec un type concret (comme $String$ ou $Integer$), ce type est **automatiquement propagé** aux classes et interfaces génériques dont elle hérite où qu'elle implémente.

- **Overriding :**

Overriding et signature des méthodes :

- L'overriding (redéfinition) consiste à redéfinir une méthode de la classe mère dans une classe fille ;
- Pour qu'il y ait overriding, les signatures des deux méthodes doivent être identiques ;
- Avec des types génériques, la vérification des signatures s'effectue après l'effacement des types génériques (par exemple T), c'est-à-dire après que les types génériques sont remplacés par leurs bornes, généralement $Object$.

- **Problème** : Lors de l'effacement des types génériques, le compilateur considère T comme $Object$ entrainant une **ambiguïté**.
- **Overriding et surcharge :**
  Une classe générique peut surcharger des méthodes tout en utilisant différentes signatures, mais les signatures doivent rester distinctes après effacement.
  - La surcharge ajoute de nouvelles méthodes avec des signatures distinctes ;
  - L'overriding remplace une méthode existante avec une même signature (**après effacement**).
- **Problème d'ambiguïté avec la surcharge**
  La surcharge peut entraîner des ambiguïtés, en particulier lorsque les signatures de méthodes deviennent similaires après effacement.
  Dans le cas de $D < String >$, le compilateur ne peut pas déterminer laquelle des deux méthodes doit être appelée.

Code antérieur aux classes génériques

Classes brutes (raw types)

Avant Java 5, les classes comme $List$, Set, etc., n'étaient pas génériques. Les développeurs pouvaient insérer n'importe quel type d'objet dans ces collections, ce qui demandait un cast manuel lors de la récupération des éléments.

Exemple avant Java 1.5 :
$List\ list = new\ ArrayList(\ );$
$list.add("Hello");$
$String\ s = (String)\ list.get(0);$ // Cast manuel nécessaire

Avec l'introduction des génériques, les types des collections sont explicitement spécifiés :

$List < String >\ list\ =\ \boldsymbol{new}\ ArrayList <>\ (\ );$
$list.add("Hello");$ // Typé correctement
$String\ s\ =\ list.get(0);$ // Plus besoin de cast

Cependant, le code ancien qui utilise des types bruts est toujours compatible avec le code moderne.
Cela permet la réutilisation de bibliothèques écrites avant Java 5, mais introduit des risques et des limitations.

Ignorer les warnings avec précaution

Si on est sûrs que la méthode appelée ne peut pas violer l'intégrité de type, on peut ignorer les warnings.

Une autre situation problématique

Les warnings apparaissent également pour les types retournés par les classes legacy.

Warning en Compilation : $[unchecked]\ unchecked\ conversion$

- Rien ne garantit que les éléments de la liste retournée sont du type attendu ($String$ ici).

Suppression des warnings

Si on est certain que la méthode respecte l'intégrité de type, il est possible d'utiliser l'annotation $@SuppressWarnings$ pour ignorer les warnings.

Annotation $@SuppressWarnings$

L'annotation $@SuppressWarning$ permet de désactiver certains avertissements (warnings) générés par le compilateur Java.

Elle s'applique sur :

- Une classe entière ;
- Une méthode spécifique ;
- Une variable, un champ, ou un bloc de code.

Syntaxe : $@SuppressWarnings("type\_de\_warnings")$

Types de warnings courants

Type	Description
`"unchecked"`	Avertissements liés aux types génériques et la sécurité des types.
`"deprecation"`	Méthodes ou classes marquées comme **obsolètes** (deprecated).
`"unused"`	Variables, méthodes ou champs **non utilisés**.
`"cast"`	Casts potentiellement dangereux.
`"all"`	Supprime **tous les warnings** (à utiliser avec précaution).

Conversion d'une collection de tableau

Problème : Lorsqu'on souhaite convertir une collection générique en tableau, plusieurs limitations apparaissent :

- Impossible d'instancier un tableau générique :

  $T[\,]\ array = new\ T[10]$ // Interdit
- Impossible de créer un tableau de type générique directement :

  $Couple < Double > [\,]\ tab = new\ Couple < Double > [5];$ // Interdit

Limitation

- Warning en compilation : $unchecked\ cast$;
- Erreur à l'exécution :

  Si T n'est pas une superclasse de E, cela génère une $ClassCastException$.

Maps

Approches avec des tables associatives

Une table associative est une structure de données qui associe une clé à une valeur.

- Une clé est transformée en un indice pour localiser efficacement la valeur associée ;
- La valeur associée à la clé est stockée à l'indice généré.

Tables associatives

Une table associative est une structure permettant d'associer deux éléments :

- Une clé ;
- Une valeur.

Principale utilité : Retrouver une valeur à partir d'une clé donnée.

Exemples :

- Dictionnaire : mot (clé) -> définition (valeur) ;
- Annuaire : nom (clé) -> numéro de téléphone (valeur) ;
- Annuaire inversé : numéro de téléphone (clé) -> nom et adresse (valeur).

Interface Map

- L'interface Map en Java est la base des tables associatives ;
- Associations clé/valeurs uniques :
  - Chaque clé est unique.
  - Une valeur peut être partagée par plusieurs clés.
- Depuis Java 5, les $Maps$ sont **génériques** :$Map < K, V > map = new\ HashMap <> ( );$ Deux paramètres de type $< K, V >$ où K représente les clés et V représente les valeurs.

Ajout et suppression d'éléments

- Ajouter un élément :
  - Méthode : $put(K\ clé, V\ valeur)$;
  - Exemple : $map.put("Yannis", 19);$
- Supprimer un élément :
  - Méthode : $remove(K\ clé);$
  - Exemple : $map.remove("Yannis");.$
- Remarque :
  - Si une clé existe déjà, put remplace l'ancienne valeur associée ;
  - $remove$ Retourne la valeur supprimée si elle existait, sinon $null.$

Recherche dans une table

- Retrouver une valeur à partir d'une clé :
  - Méthode : $get(K\ clé)$;
  - Exemple : $String\ valeur = map.get("Yannis");.$
- Vérifier l'existence d'une clé :
  - Méthode : $containsKey(K\ clé)$;
  - Exemple : $if\big(map.containsKey("Yannis")\big)\{\ System.out.println("Clé\ présente\ !\ ");\}$

Parcourir une table associative

Les méthodes de « vue de collection » permettent de visualiser une Map comme une collection de trois manières différentes :

- $keySet$: Représente l'ensemble des clés.
  Méthode : $Set < K > keySet( );$
- $values$: Représente la collection des valeurs.
  Méthode : $Collection < V > values( );$
- $entrySet$: Représente l'ensemble des couples (clé, valeur).
  Méthode : $Set < Map.Entry < K, V \gg entrySet( )$.

Interface $Map.Entry < K, V >$

L'interface $Map.Entry$ représente une paire clé/valeur et fournit trois méthodes principales :

- $getKey( )$: Renvoie la clé de l'entrée ;
- $getValue( )$: Renvoie la valeur associée à la clé ;
- $setValue(V\ value)$: Remplace la valeur associée par une nouvelle valeur.

Implémentation des tables associatives

Deux types d'implémentation principales :

1. Table de hachage :

- Classe : $HashMap$;
- Accès rapide : Complexité moyenne de la recherche $O(1)$;
- Exemple : $HashMap < K, V > map = \textbf{new } HashMap <> ();$

2. Arbre binaire :
   - Classe : $TreeMap$;
   - Clés triées : Complexité moyenne de la recherche $O(\log N)$, mais des clés triées automatiquement.
   - Exemple : $TreeMap < K, V > map = \textbf{new } TreeMap <> ();$

Les fonctions de hachage

Une fonction de hachage est une fonction mathématique :
- Prend en entrée une donnée (souvent de longueur variable) ;
- Produit une empreinte unique ou haché (longueur fixe). $H : \{0,1\}^* \to \{0,1\}^n$

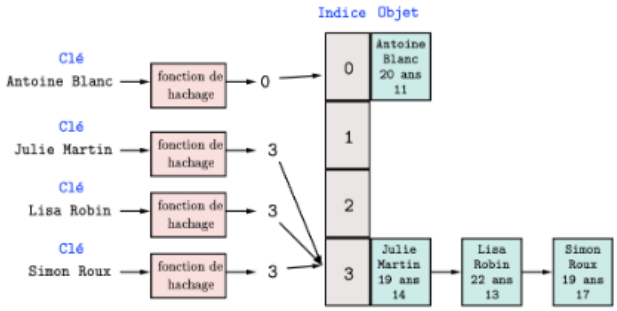
Cette empreinte sert à identifier rapidement la donnée d'origine.

Propriété d'une fonction de hachage

- Si $k_1 = k_2$, alors $H(k_1) = H(k_2)$.

Problème de collisions :
- Si $k_1 \neq k_2$, il est difficile (voire impossible) de garantir que $H(k_1) \neq H(k_2)$;
- Lorsque deux clés différentes produisent le même haché, on parle de collision.

Une bonne fonction de hachage doit **minimiser les collisions et répartir les clés uniformément**.



Une approche pour gérer les collisions

Si plusieurs clés ont le même indice (collision), la $HashMap$ utilise une de ces approches pour traiter les collisions :

Liste chaînée **(avant Java 8)** : Les paires sont stockées dans une liste chaînée à l'indice calculé ;

Arbre rouge-noir **(depuis Java 8, lorsque la liste dépasse un seuil)** : Si le nombre de collisions dépasse un certain seuil (8 par défaut), la liste chaînée est convertie en arbre rouge-noir, ce qui améliore les performances de recherche.

La méthode $hashCode()$ de la classe $String$

La méthode $hashCode()$ est une méthode standard de la classe $String$ en Java, utilisée pour calculer une empreinte unique pour chaque chaîne de caractères.
Elle peut être utilisée (combinée avec d'autres opérations) comme fonction de hachage pour déterminer l'indice dans une table de hachage. $public\ int\ hashCode()$

Fonctionnement : Si s est une chaîne de caractère de longueur n, l'appel $s.hashCode()$ retourne un entier calculé de la manière suivante :

$$hasCode(s) = s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \cdots + s[n-1].$$

Enumérations

Enumérations

- Une énumération est une liste de constantes nommées qui définit un nouveau type de données ;
- Un objet d'un type énumération ne peut prendre qu'une des valeurs prédéfinies dans cette énumération ;
- Les énumérations permettent de représenter un ensemble fixe de valeurs valides.

L'intérêt des énumérations

- Les énumérations permettent de représenter un ensemble de valeurs prédéfinies ;

Alternative aux variables final :

- Avant Java 5, on utilisait des variables $final$ pour définir ces valeurs.

Limites des constantes finales

L'utilisation de constantes $final$ pour représenter des valeurs prédéfinies présente plusieurs limitations importantes :

- Pas de contrôle de valeur : Une censée représenter un « statut » reste un simple int.
- Impossible de parcourir automatiquement les constantes :
  - Les constantes sont éparpillées sous forme de simples variables $final$;
  - Aucun moyen de faire une boucle du type :

    $for\ (Status\ s : Status.values())\{\dots\}$ // Impossible avec des int

    Il faut maintenir manuellement une liste, ce qui est source d'erreurs.
- Aucune capacité à associer un comportement : Les constantes $final$ ne sont que des entiers :
  - Impossible de leur attacher une méthode, un message, etc. ;
  - On est obligé d'écrire des $if/switch$ dispersés dans le code pour traiter chaque cas.

Définition d'une énumération

Les énumérations sont définies avec le mot-clé $enum$.

Bonnes pratiques :

- Par convention, les constantes dans une énumération sont souvent écrites en majuscules.
  Pourquoi utiliser les majuscules ?
  1. Héritage des conventions :
     - Les énumérations remplacent souvent des **variables** $final$ qui, par tradition, étaient en majuscules ;
  2. Clarté visuelle :
     - Les majuscules permettent de **différencier facilement les constantes** d'autres identifiants dans le code.
- Utiliser un style cohérent : Si on adopte les majuscules pour une énumération, le**s garder partout**.

Utilisation d'une énumération

- Déclaration d'une variable :

- $Transport\ tp;$ // Variable de type Transport
- $tp = Transport.AVION$ // Assignation d'une valeur

- Comparaison

$if\ (tp == Transport.TRAIN)\{\ System.out.println("tp\ contient\ TRAIN.");\}$

Méthodes prédéfinies

- Méthode $values()$: Retourne un tableau contenant toutes les constantes de l'énumération.

$Transport[\,]\ allTransport = Transport.values();$
$for(Transport\ t : allTransports)\{\ System.out.println(t);\}$

- Méthode $valueOf(String)$: Retourne la constante correspondant à la chaîne passée en paramètres.

$Transport\ tp = Transport.valueOf("AVION");$
$System.out.println(tp);$ // Affiche "AVION"

Les énumérations comme types de classes

En Java, une énumération est une classe.

Les énumérations peuvent :

- Avoir des constructeurs ;
- Définir des variables d'instances ;
- Ajouter des méthodes ;
- Implémenter des interfaces.

Restrictions des énumérations

1. Une énumération ne peut pas hériter d'une autre classe.
   Raison : Toutes les énumérations héritent déjà implicitement de la classe $java.lang.enum$.
   Une énumération est **déjà liée à une hiérarchie de classe**, et ne peut donc pas hériter d'une autre classe.
2. Une énumération ne peut pas être une superclasse.
   Raison : Les énumérations sont conçues pour définir un ensemble fixe et immuable de constantes, et ne peuvent donc pas être étendues.

Méthodes de $java.lang.Enum$

- Méthode $ordinal()$: Renvoie la position d'une constante dans la liste, à partir de 0.
- Méthode $compareTo()$: Compare les positions ordinales de deux constantes.