SpaceFox

Zeste de savoir

Java : la NullPointerException sortie de nulle part ?

18 mars 2021

Table des matières

1. Une exception sauvage apparaît! 1
2. Boxing, unboxing 2
3. Mais alors, pourquoi le code plante? 3

1. Une exception sauvage apparaît !

Ce code, si on l'appelle de la façon suivante: `parseInteger(null, null)` plante avec une `NullPointerException` à la ligne 4.

```
1   public static Integer parseInteger(final String s, final
        Integer defaultValue) {
2       try {
3           return s == null
4                   ? defaultValue
5                   : Integer.parseInt(s);
6       } catch (NumberFormatException e) {
7           Log.trace("Can't convert " + s +
                " to Integer, use default value " +
                defaultValue);
8           return defaultValue;
9       }
10  }
```

Étrange, non?

Pourtant en première approche, on pourrait croire que si le premier paramètre `s` est `null`, on se contente de renvoyer le second paramètre `defaultValue` qui peut aussi être `null`.

? Alors pourquoi ça plante? D'où sort cette `NullPointerException`?

Pour le comprendre, il va falloir creuser dans les notions de «Boxing», d'«Unboxing» et de la priorité dans les opérateurs ternaires.

2. Boxing, unboxing

Ces deux notions sont normalement connues des développeurs expérimentés en Java, mais pour les débutants et les curieux, je la réexplique.

La JVM (qui fait tourner les programmes Java), ne connait que 9 types pour manipuler les données:

— Les huit types «natifs», «primitifs» ou «de base»: `boolean`, `byte`, `short` (presque jamais utilisé), `int`, `long`, `float`, `double` et `char`;
— Et Les objets, indépendamment des données qu'ils contiennent.

`String` a droit à un traitement un peu particulier, parce que techniquement c'est un objet, mais il a droit à des spécificités intégrées directement dans la JVM. De même avec les exceptions (plus exactement, tout ce qui implémente `Throwable` à un degré ou un autre).

Et… c'est tout. Absolument tout ce qu'on peut faire en Java se repose sur ces concepts.

Le problème là-dedans, c'est que **les types natifs *ne sont pas* des objets**. Ça veut dire que:

1. Ils ont des comportements différents (par exemple, ils sont passés par copie et pas par référence dans les méthodes), et surtout
2. On ne peut pas les utiliser là où on a besoin d'un objet.

En particulier, ça veut dire qu'ils ne peuvent pas être `null`, la notation désignant une étiquette (variable ou autre) qui *n'est pas* rattachée à un objet en mémoire. Ils ne peuvent pas non plus être utilisés dans les mécanismes tels que la généricité, qui ne fonctionne qu'avec des objets.

Pour pouvoir utiliser ces types en tant qu'objets, pour chaque type natif, on a créé un objet correspondant (`Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double` et `Character` – ça n'est pas toujours «juste le même nom avec une majuscule», c'eut été trop simple). Mais si on gagne la possibilité d'utiliser nos types comme des objets, on perd plein de choses dans l'opération. Par exemple, aucune arithmétique n'est possible sur les objets. Pas très pratique.

Jusqu'en Java 1.4 inclus, il fallait faire les conversion entre les types natifs et leurs objets à la main. Ça impliquait d'écrire beaucoup de code trivial – parce que c'est une opération dont on a *souvent* besoin –, ce qui a aidé à cette réputation de verbosité de Java.

Et puis on s'est dit que c'était quand même con de passer du temps à écrire du code trivial sans aucune valeur ajoutée, et qu'on pouvait déléguer le boulot en sous-marin au compilateur, qui va ajouter tout seul les conversions en type natif ou en objet au besoin.

C'est ce qu'on appelle **l'auto-boxing** (on range automatiquement notre type natif dans une boite: son objet) et **l'auto-unboxing** (l'opération inverse).

Et ça marche tellement bien qu'en général on oublie complètement que ça existe.

3. Mais alors, pourquoi le code plante ?

Reprenons le code (je supprime la gestion d'exception pour la lisibilité, ce code ne compile pas en l'état):

```
1        public static Integer parseInteger(final String s, final
             Integer defaultValue) {
2            return s == null
3                                ? defaultValue
4                                : Integer.parseInt(s);
5        }
```

Les types de sortie de la méthode comme de la valeur par défaut sont `Integer`, donc l'objet «entier de 32 bits signé». Comme tout objet en Java (hélas), ils peuvent être `null`.

Par contre, `Integer.parseInt(String)` renvoie un `int`, le type natif. Donc il y a un boxing ou un unboxing caché dans cette histoire pour que ça puisse fonctionner.

Puisque le type du retour et le premier paramètre de la condition ternaire utilisent chacun objet `Integer`, on pourrait supposer que le compilateur va être intelligent et nous faire un beau boxing pour gérer tout ça à moindre frais:

```
1        public static Integer parseInteger(final String s, final
             Integer defaultValue) {
2            return s == null
3                                ? defaultValue
4                                :
                                     Integer.valueOf(Integer.parseInt(s));
5        }
```

Mais non! Les règles de gestion de l'auto(un)boxing couplées à celles des ternaires font que le compilateur va plutôt choisir de convertir les deux branches de la condition en type natif (en suivant la directive du «`else`», puis de re-convertir le tout en objet avant de le renvoyer:

```
1        public static Integer parseInteger(final String s, final
             Integer defaultValue) {
2            return Integer.valueOf(s == null
3                                ? defaultValue.intValue()
4                                : Integer.parseInt(s));
5        }
```

Et là, c'est le drame: à la ligne 3, on essaie d'appeler une méthode sur un objet `null`… et ça renvoie la `NullPointerException` observée. Et comme `NullPointerException` n'est pas une fille de la `NumberFormatException` que l'on attrape, la méthode plante.

3. *Mais alors, pourquoi le code plante?*

La combinaison de ternaires et de l'autoboxing peut donc provoquer des erreurs là où le code a l'air tout à fait inoffensif.

Parce que oui, la version avec un `if` normal se comporte bien comme prévu et ne plante pas quand on l'appelle ainsi: `parseInteger(null, null)`.

```
1     public static Integer parseInteger(final String s, final
          Integer defaultValue) {
2         try {
3             if (s == null) {
4                 return defaultValue;
5             }
6             return Integer.parseInt(s);
7         } catch (NumberFormatException e) {
8             Log.trace("Can't convert " + s +
                  " to Integer, use default value " + defaultValue);
9             return defaultValue;
10        }
11    }
```

Ou même sa version avec une seule instruction `return`:

```
1     public static Integer parseInteger2(final String s, final
          Integer defaultValue) {
2         Integer result = defaultValue;
3         try {
4             if (s != null) {
5                 result = Integer.parseInt(s);
6             }
7         } catch (NumberFormatException e) {
8             Log.trace("Can't convert " + s +
                  " to Integer, use default value " + defaultValue);
9         }
10        return result;
11    }
```

Attention aux effets de bord imprévus si vous aimez jouer avec la fonction *«Remplacer ce `if` par un ternaire»* de votre IDE favori!