25/11/2021

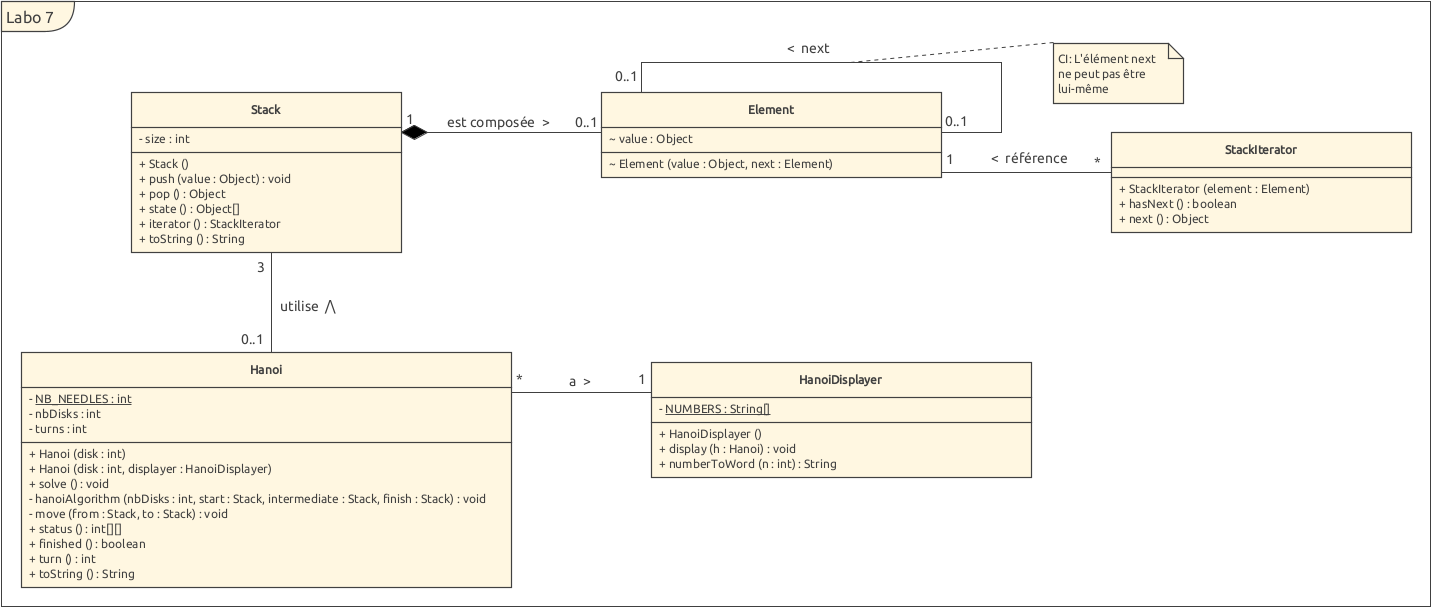
Valentin Kaelin et Jonathan Friedli

Rapport Laboratoire 7 : Hanoi

# Introduction

Dans le cadre de ce laboratoire n°7, nous avons dû implémenter un programme permettant de résoudre le problème des tours d’Hanoi de deux façons différentes : soit directement dans le terminal ou alors via une interface graphique. Dans les deux cas l’utilisateur peut choisir la taille de la tour initiale. L’implémentation utilise une pile que nous devons créer à la main.

# Diagramme des classes



# Description des classes

## Package hanoi

### Hanoi

Classe contenant toute la logique de la résolution du problème des tours d’Hanoi. Elle s’occupe notamment de contenir les trois aiguilles ainsi que d’appliquer l’algorithme récursif.

### HanoiDisplayer

Classe permettant l’affichage des étapes de résolution du problème dans la console.

## Package util

### Stack

Représentation générique d’une pile grâce à l’utilisation de la classe Object. Il est possible d’ajouter ou de retirer un élément à la fois ainsi que de la parcourir. La pile possède également une représentation graphique.

### StackIterator

Classe représentant un itérateur pointant sur un élément. Il est possible d’accéder à l’élément suivant.

### Element

Classe conteneur permettant de lier un élément avec un potentiel élément suivant.

## Package global

### Hanoi

Classe principale du programme, s’occupe de lancer le programme de résolution des tours d’Hanoi. Elle contient également les tests de l’implémentation de la Stack.

# Algorithme utilisé

Nous avons utilisé l’algorithme récursif, présenté notamment sur Wikipédia : [https://fr.wikipedia.org/wiki/Tours\_de\_Hanoï - Solution\_récursive](https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF#Solution_r%C3%A9cursive)

Cette solution a l’avantage d’être simple à implémenter et très compréhensible. L’idée de cet algorithme est la suivante :

Pour déplacer une tour de *n* disques de A vers C, on effectue ces trois étapes :

1. Déplacer la tour des *n-1* premiers disques de A vers B (appel récursif).
2. Déplacer le plus grand disque de A vers C.
3. Déplacer la tour des *n-1* premiers disques de B vers C (appel récursif).

# Question 1

Le nombre de déplacements pour résoudre la tour de Hanoï vaut 2n-1, avec n le nombre de disque de la tour. Dans notre cas, n vaut 64 donc il faut 264-1 déplacements. Ce qui vaut 18'446'744'073'709'552'000 déplacements.

Les moines faisant un déplacement par seconde il faudrait 18'446'744'073'709'552'000 secondes. L'univers ayant environ 13.8 milliards d'année mais les moines n’existaient sûrement pas (à ma connaissance) au début de l'univers. Cependant, nous allons supposer que le premier déplacement a eu lieu il y a exactement 13.8 milliards d'années. Nous soustrayons donc 13'800'000'000 \* 365,25 \* 24 \* 3600 à 18'446'744'073'709'552'000 ce qui donne :

18'446'744'073'709'552'000 - 435'494'880'000'000'000 = 18'011'249'193'709'552'000 secondes restantes ce qui représente environ 570'742'046'090 années. Donc environ **570,7 milliards d'années**.

# Choix de modélisation et d’implémentation

## Hanoi

La représentation graphique du tour a été implémentée dans la classe Hanoi afin d’éviter de devoir faire des accesseurs sur les différentes aiguilles pour le réaliser dans la classe HanoiDisplayer. Cette dernière se contente donc d’avoir une fonction helper et appelle le toString de l’instance de la classe Hanoi reçue en paramètre.

## Stack

Nous avions tout d’abord essayé de faire une vraie version générique de la Stack mais il n’est pas aisé de travaille avec des tableaux génériques (à la place d’ArrayList). Nous nous sommes donc rabattus sur la version « générique » vue en cours grâce à la classe Object.

Afin de suivre les conventions Java, la méthode next() dans l’itérateur retourne l’élément courant tout en le remplaçant par le prochain élément par après. De même, la méthode hasNext() vérifie que l’élément courant existe bien et pas un élément plus loin. Cela nous permet de boucler facilement sur tous les éléments de la pile.

Nous avons choisi la visibilité package pour tous les éléments de la classe Element car nous l’utilisons plus comme une structure utile à l’interne dans l’implémentation de la pile. En effet, il n’est pas utile d’avoir accès à l’implémentation de l’élément en dehors du package util, dans le code métier.

La CI comme quoi le next d’un élément ne doit pas être lui-même n’a pas été implémentée directement dans le code étant donné que la classe est en visibilité package. L’utilisateur utilisant notre librairie ne pourra faire cette bêtise et nous partons du principe que nous ne faisons pas ça non plus dans notre code.

# Tests effectués

Nous avons tout d’abord réalisé le test demandé dans la donnée permettant de lancer

Par la suite, nous avons testé l’implémentation de notre classe Stack avec via les tests suivants :

|  |  |
| --- | --- |
| **Test** | **Résultat attendu et observé** |
| Ajouter une valeur dans la pile | La valeur est ajoutée tout au haut de la pile |
| Retirer une valeur dans la pile | L’élément tout en haut de la pile est supprimé et sa valeur est retournée |
| Création d’une pile vide et affichage de celle-ci | Les opérations se déroulent sans soucis |
| Affichage des éléments d’une pile grâce à sa représention sous forme de chaîne de caractères | Les éléments sont affichés sous la forme demandée, c’est-à-dire : [ <a> <b> <c>] |
| L’itérateur de la pile fonctionne correctement | Il est possible d’itérer sur toutes les valeurs de la pile sans problème et dans le bon ordre |
| Récupération des données de la pile via sa méthode state retournant un tableau | Le tableau contient toutes les valeurs de la pile dans le bon ordre |
| Récupérer les données de la pile via la méthode state ne brise pas l’encapsulation | Modifier le tableau retourné ne modifie pas le tableau dans la classe de la pile |
| Retirer un élément d’une pile vide | Exception lancée lors de l’opération |
| Ajouter une valeur d’un type différent (String dans notre cas) dans la pile | Les opérations se déroulent sans soucis |