eFrei
PARIS PANTHÉON-ASSAS UNIVERSITÉ

Structure de données : Rapport sur les piles

Rédigé et présenté par :

- Cédric Alonso
- Jade Hatoum

Année académique 2023 - 2024

SOMMAIRE

I. Pile statique

II. Pile dynamique

III. Pile fichier

Introduction

Structure Générale : Une pile, par sa nature, suit le concept “FiLo” soit “First in, Last out”. Ceci signifie donc que la première information empilée dans cette structure, sera la dernière à être dépilée, et vice-versa.

Nous avons décidé d’effectuer ce TP en Java. Nous remplacerons donc init_stack() par le constructeur de la pile respective.

De plus, nous avons ainsi créé une interface *Stack* pour que nos différentes piles puissent redéfinir le comportement de *push( )*, *pop( )*, *top( )* et *display( )*.

En effet, les interfaces servent à créer des comportements génériques: si plusieurs classes doivent obéir à un comportement particulier, on créé une interface décrivant ce comportement, on l’a fait implémenter par les classes qui en ont besoin. Ces classes devront ainsi obéir strictement aux méthodes de l'interface (nombre, type et ordre des paramètres, type des exceptions), sans quoi la compilation ne se fera pas.

Concernant le type de données stocké dans nos piles, nous avons décidé d’utiliser un type paramétrable, noté *T*, à l’instanciation de la pile afin de permettre une meilleure modulation d’utilisation.

Pour simplifier les tests, nous avons créé une classe *Test* reprenant les méthodes pouvant lever une exception (*pop( ), top( ), display( )*).

Pour finir, nous avons créé 2 classes d’exceptions *EmptyStackExceptions* et *FileException* afin de mieux gérer les erreurs possibles.

1. Interface *Stack*

```java
package TP2.stack;

/**
 * L'interface Stack<T> définit les opérations de base d'une pile (stack).
 *
 * @param <T> Le type des éléments stockés dans la pile.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
15 usages  3 implementations  mrredcoding
public interface Stack<T> {

    /**
     * Ajoute un élément au sommet de la pile.
     *
     * @param value L'élément à ajouter à la pile.
     */
    9 usages  3 implementations  mrredcoding
    void push(T value);

    /**
     * Retire et renvoie l'élément situé au sommet de la pile.
     *
     * @return L'élément retiré de la pile.
     * @throws EmptyStackExceptions Si la pile est vide au moment de l'appel.
     */
    1 usage  3 implementations  mrredcoding
    T pop() throws EmptyStackExceptions;

    /**
     * Renvoie l'élément situé au sommet de la pile sans le retirer.
     *
     * @return L'élément au sommet de la pile.
     * @throws EmptyStackExceptions Si la pile est vide au moment de l'appel.
     */
    1 usage  3 implementations  mrredcoding
    T top() throws EmptyStackExceptions;

    /**
     * Affiche les éléments de la pile. L'ordre d'affichage dépend de l'implémentation
     * spécifique de la pile.
     *
     * @throws EmptyStackExceptions Si la pile est vide au moment de l'appel.
     */
    1 usage  3 implementations  mrredcoding
    void display() throws EmptyStackExceptions;
}
```

2. Classe *Test*

```java
package TP2.utils;

import TP2.stack.EmptyStackExceptions;
import TP2.stack.Stack;

/**
 * Classe utilitaire pour afficher des informations sur les piles.
 * Fournit des méthodes statiques pour générer des boîtes de titre stylisées,
 * afficher le sommet et le contenu d'une pile, et traiter les opérations de pop.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
30 usages  mrredcoding +1
public class Test {

    /**
     * Génère une boîte de titre stylisée pour afficher un titre.
     *
     * Utilise des caractères encodés en UTF-8 pour créer une boîte de titre stylisée.
     * Les lignes de la boîte sont constituées de caractères de ligne horizontale (─),
     * de coins (┌, └, ┐, ┘), et d'espaces entourant le titre.
     *
     * @param title Le titre à afficher dans la boîte.
     * @return Une chaîne représentant la boîte de titre stylisée.
     */
    3 usages  mrredcoding +1
    public static String titleBox(String title) {
        String repeat = "─".repeat(Math.max(0, title.length() + 2));

        return "\u001B[34m" +
                System.lineSeparator() +
                "┌" + repeat +  "┐" +
                System.lineSeparator() +
                "│ " + title + " │" +
                System.lineSeparator() +
                "└" + repeat +  "┘" +
                "\u001B[0m";
    }

     * Affiche et retire le sommet de la pile.
     *
     * @param stack La pile à manipuler.
     */
    9 usages  mrredcoding
    public static void pop(Stack<?> stack){
        try {
            System.out.println(stack.getClass().getSimpleName() + " Popped value = " + stack.pop());
            System.out.println();
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * Affiche le sommet de la pile sans le retirer.
     *
     * @param stack La pile à inspecter.
     */
    3 usages  mrredcoding
    public static void top(Stack<?> stack){
        try {
            System.out.println(stack.getClass().getSimpleName() + " Top value = " + stack.top());
            System.out.println();
        } catch (EmptyStackExceptions e) {
            System.out.println(e.getMessage());
        }
    }
```

```java
    /**
     * Affiche le contenu complet de la pile.
     *
     * @param stack La pile à afficher.
     */
    12 usages  mrredcoding
    public static void display(Stack<?> stack){
        try {
            System.out.println(stack.getClass().getSimpleName() + " Contents :");
            stack.display();
            System.out.println();
        } catch (EmptyStackExceptions e){
            System.out.println(e.getMessage());
        }
    }
}
```

3. Les exceptions

```java
package TP2.stack;

/**
 * Exception levée lorsqu'une opération est effectuée sur une pile vide.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
31 usages  mrredcoding
public class EmptyStackExceptions extends Exception {

    /**
     * Construit une nouvelle exception avec le message spécifié.
     *
     * @param message Le message d'erreur associé à l'exception.
     */
    9 usages  mrredcoding
    public EmptyStackExceptions(String message) { super("\u001B[31m" + message + "\u001B[0m"); }

}
```

```java
package TP2.stack;

/**
 * Exception spécifique pour les erreurs liées aux opérations sur les fichiers dans le contexte de la pile.
 *
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
6 usages  mrredcoding
public class FileException extends Exception {

    /**
     * Construit une nouvelle exception avec le message spécifié.
     *
     * @param message Le message d'erreur associé à l'exception.
     */
    2 usages  mrredcoding
    public FileException(String message) { super("\u001B[31m" + message + "\u001B[0m"); }
}
```

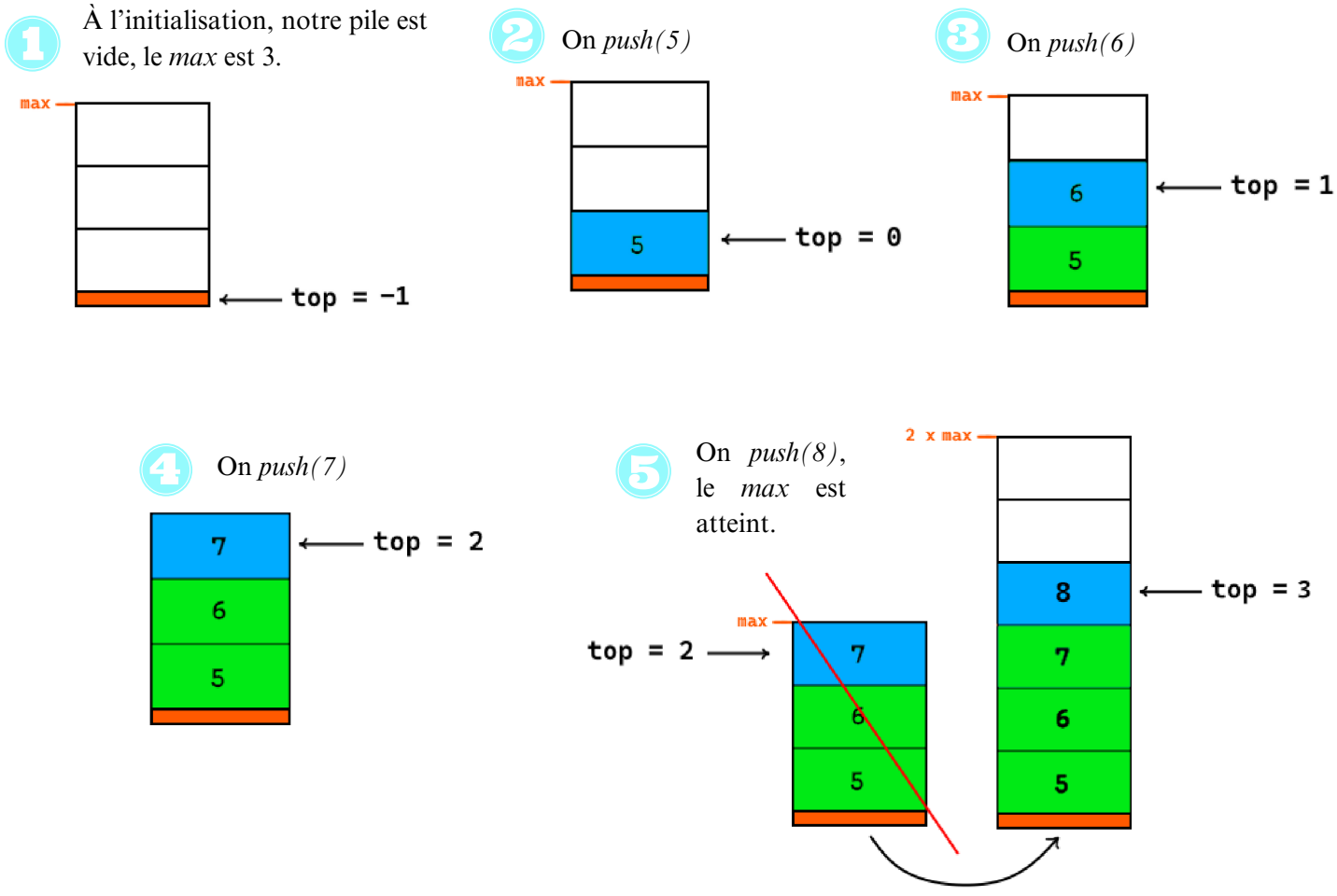
I. Pile statique

1. Algorithme explicité

Dans notre cas, il s'agit d'une pile statique, qui elle fait recours à un tableau statique pour sa structure. Ainsi, un attribut *max* symbolise la taille maximale de ce tableau statique appelé *tab.* De plus, l'attribut *top* représente l'indice de l'information en tête de la pile, dans *tab*.

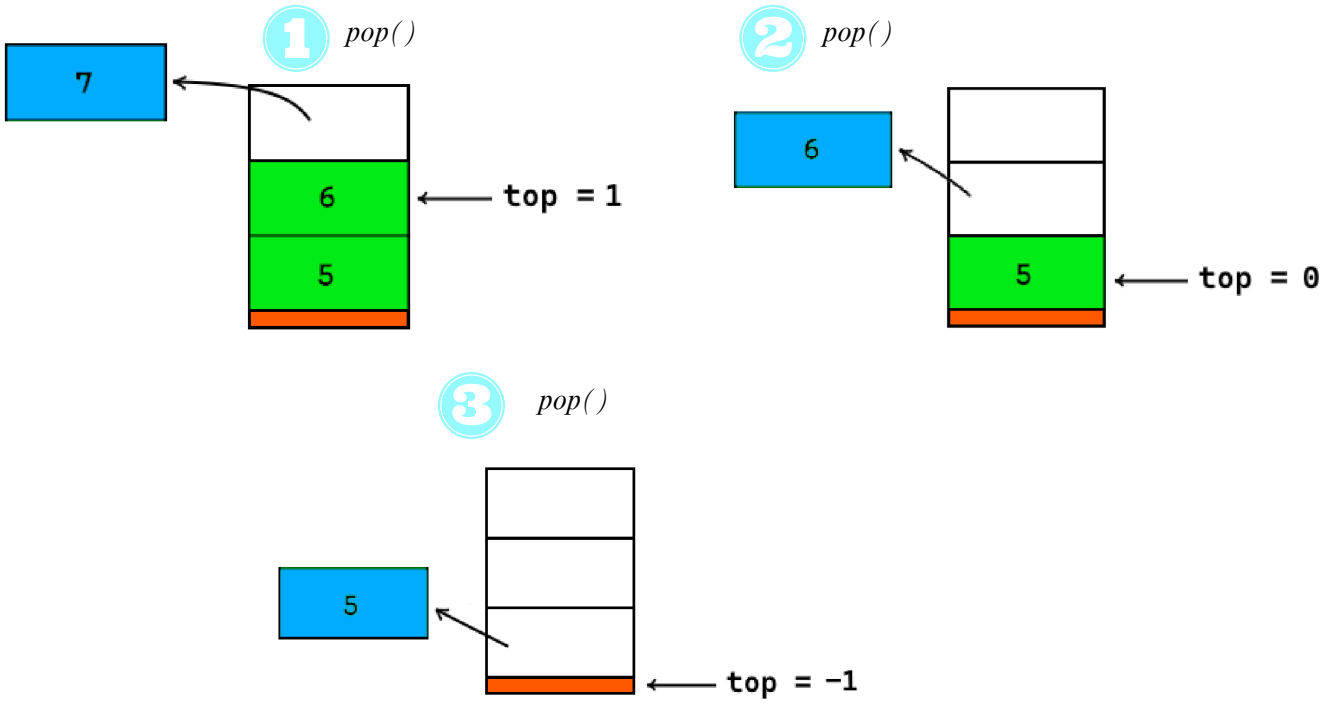
La méthode *push( )* est utilisée pour ajouter un nouvel élément *de type T* contenant une valeur spécifique au sommet de la pile. L'algorithme se déroule comme suit :

- On vérifie si le *top* est égal à la taille maximale du tableau statique allouée.
  - Si c'est le cas, notre pile est pleine, et donc on alloue un nouveau tableau statique 2 fois plus grand en mémoire. Puis on copie les données de la pile pleine vers le nouveau tableau statique. Enfin, on change la référence en mémoire : l'ancien tableau pointera donc vers le nouveau tableau et l'ancien tableau sera donc détruit par le Garbage Collector automatiquement.
- Finalement, on ajoute la nouvelle donnée au sommet de la pile
- On incrémente *top* de 1.



La méthode *pop( )* est utilisée pour récupérer la valeur de l'élément *de type T* du sommet de la pile tout en le supprimant de la pile statique. L'algorithme se déroule comme suit :

- On vérifie si le *top* est supérieur ou égal à 0
  - Si c'est le cas, on copie la valeur de l'élément *de type T* du sommet de la pile dans une variable puis on décrémente *top* de 1. Finalement on retourne la valeur sauvegardée dans la variable temporaire.
  - Sinon, on lève une exception, indiquant que la pile est vide.



La méthode *top( )* est utilisée pour lire la valeur de l'élément *de type T* du sommet de la pile. L'algorithme se déroule comme suit :

- On vérifie si le *top* est supérieur ou égal à 0
  - Si c'est le cas, on retourne la valeur de l'élément *de type T* du sommet de la pile.
  - Sinon, on lève une exception, indiquant que la pile est vide.

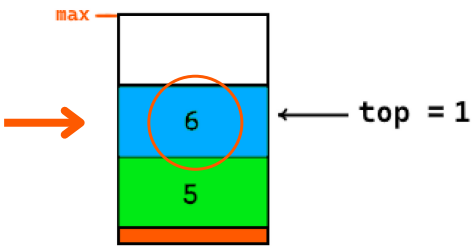
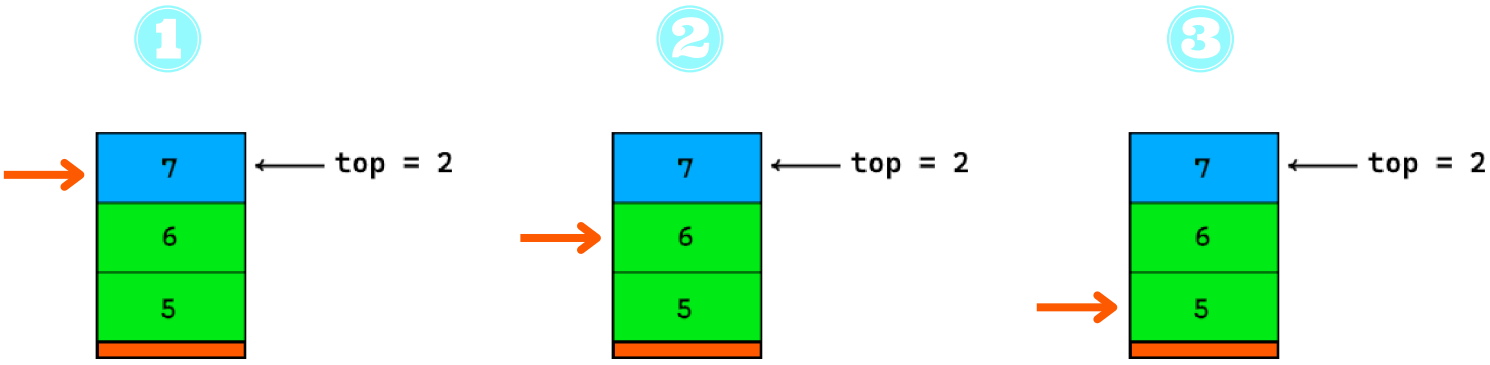


Schéma d'une instance de StaticStack<Integer>
max = 3.

La méthode *display( )* est utilisée pour afficher le contenu de la pile. On parcours le tableau statique en partant du sommet, *top*. L'algorithme se déroule comme suit :

- On vérifie si le *top* est supérieur ou égal à 0
  - Si c'est le cas,
    - On effectue une boucle affichant chaque valeur de l'élément *de type T* pour chaque indice de notre pile statique.
  - Sinon, on lève une exception, indiquant que la pile est vide.



2. Le code réalisé

```java
package TP2.staticStack;

import TP2.stack.EmptyStackExceptions;
import TP2.stack.Stack;

/**
 * Implémentation d'une pile statique avec redimensionnement automatique.
 *
 * @param <T> Le type des éléments stockés dans la pile.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
1 usage  mrredcoding
public class StaticStack<T> implements Stack<T> {

    6 usages
    private int max;
    7 usages
    private T[] tab;
    9 usages
    private int top;

    /**
     * Constructeur de la classe StaticStack.
     *
     * @param max La capacité initiale de la pile.
     */
    1 usage  mrredcoding
    public StaticStack(int max) {
        this.top = -1;
        this.max = max;
        this.tab = (T[]) new Object[this.max];
    }
    9 usages  mrredcoding
    @Override
    public void push(T value) {
        if (top == max - 1) {
            // Si la pile est pleine, crée un nouveau tableau avec le double de la taille
            T[] newTab = (T[]) new Object[max * 2];
            System.arraycopy(tab, srcPos: 0, newTab, destPos: 0, max);
            tab = newTab;
            max *= 2;
        }
        tab[++top] = value;
    }

    1 usage  mrredcoding
    @Override
    public T pop() throws EmptyStackExceptions {
        if (top >= 0) {
            return tab[top--];
        } else {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
    }

    no usages  mrredcoding
    @Override
    public T top() throws EmptyStackExceptions {
        if (top >= 0) {
            return tab[top];
        } else {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
    }
```

```
    1 usage   mrredcoding
    @Override
    public void display() throws EmptyStackExceptions {
        if (top >= 0) {
            for (int i = top; i >= 0; i--) {
                System.out.println(tab[i]);
            }
        } else {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
        System.out.println();
    }
}
```

3. Les tests et résultats de l'implémentation

```java
package TP2.staticStack;

import TP2.stack.Stack;
import TP2.utils.Test;

/**
 * Programme de test pour la classe StaticStack.
 * Crée une pile statique, effectue des opérations de push et pop, et affiche les résultats.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
mrredcoding *
public class Main {

    /**
     * Méthode principale du programme de test.
     *
     * @param args Les arguments de la ligne de commande (non utilisés dans ce programme).
     */
    mrredcoding *
    public static void main(String[] args) {
        // Crée une pile statique d'entiers avec une capacité initiale de 3.
        Stack<Integer> staticStack = new StaticStack<>( max: 3);

        // Affiche une boîte de titre stylisée avec le nom de la classe de la pile.
        System.out.println(Test.titleBox(staticStack.getClass().getSimpleName()));

        // Effectue des opérations de push sur la pile.
        staticStack.push( value: 5);
        staticStack.push( value: 6);
        staticStack.push( value: 7);

        // Affiche le contenu de la pile.
        Test.display(staticStack);

        // Effectue une opération de pop et affiche le résultat.
        Test.pop(staticStack);

        // Affiche à nouveau le contenu de la pile après la première opération de pop.
        Test.display(staticStack);

        // Affiche le sommet de la pile sans le retirer.
        Test.top(staticStack);

        // Effectue une deuxième opération de pop et affiche le résultat.
        Test.pop(staticStack);

        // Affiche à nouveau le contenu de la pile après la deuxième opération de pop.
        Test.display(staticStack);

        // Effectue une troisième opération de pop (pile vide) et affiche le résultat.
        Test.pop(staticStack);

        // Affiche à nouveau le contenu de la pile après la troisième opération de pop.
        Test.display(staticStack);
    }
}
```

```
| StaticStack |

StaticStack Contents :
7
6
5


StaticStack Popped value = 7

StaticStack Contents :
6
5


StaticStack Top value = 6

StaticStack Popped value = 6

StaticStack Contents :
5


StaticStack Popped value = 5

StaticStack Contents :
StaticStack is empty !
```

II. Pile dynamique

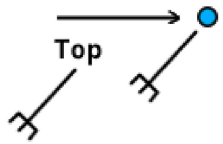
1. Algorithme explicité

Dans notre cas, il s'agit d'une pile dynamique, qui elle fait recours au principe de la liste chaînée simple (voir rapport sur les listes chaînées pour plus d'information) pour sa structure. Ainsi, nous avons une classe *Node* représentant un nœud de la liste chaînée pointant sur le nœud suivant. La pile dynamique a donc un nœud *top* pointant sur la tête de la liste chaînée.

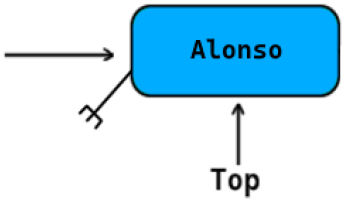
La méthode *push( )* permet d'ajouter un nouveau nœud au début de la liste chaînée. L'algorithme se déroule comme suit :

- On crée un nouveau nœud avec la valeur de l'élément *de type T*.
- Le suivant du nouveau nœud pointe donc sur le *top* actuel.
- Finalement, on affecte le nouveau nœud au *top*, indiquant donc le début de la nouvelle liste chaînée.

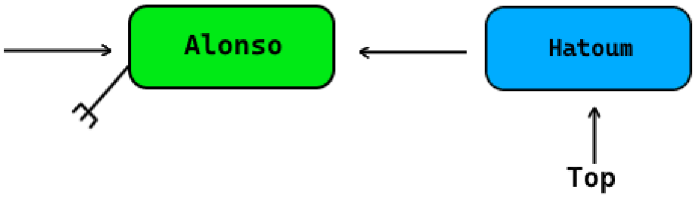
1. À l'initialisation, notre pile est vide. Le *nœud top* est null



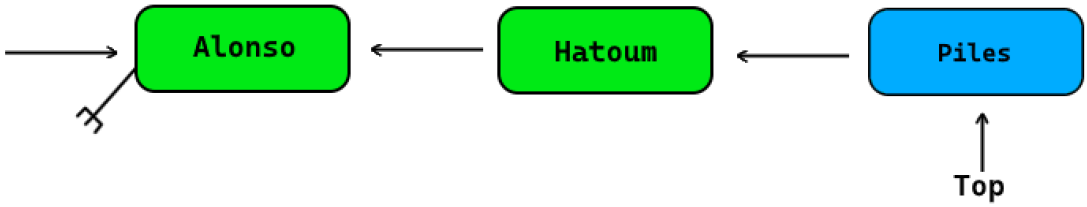
2. On *push("Alonso")*



3. On *push("Hatoum")*

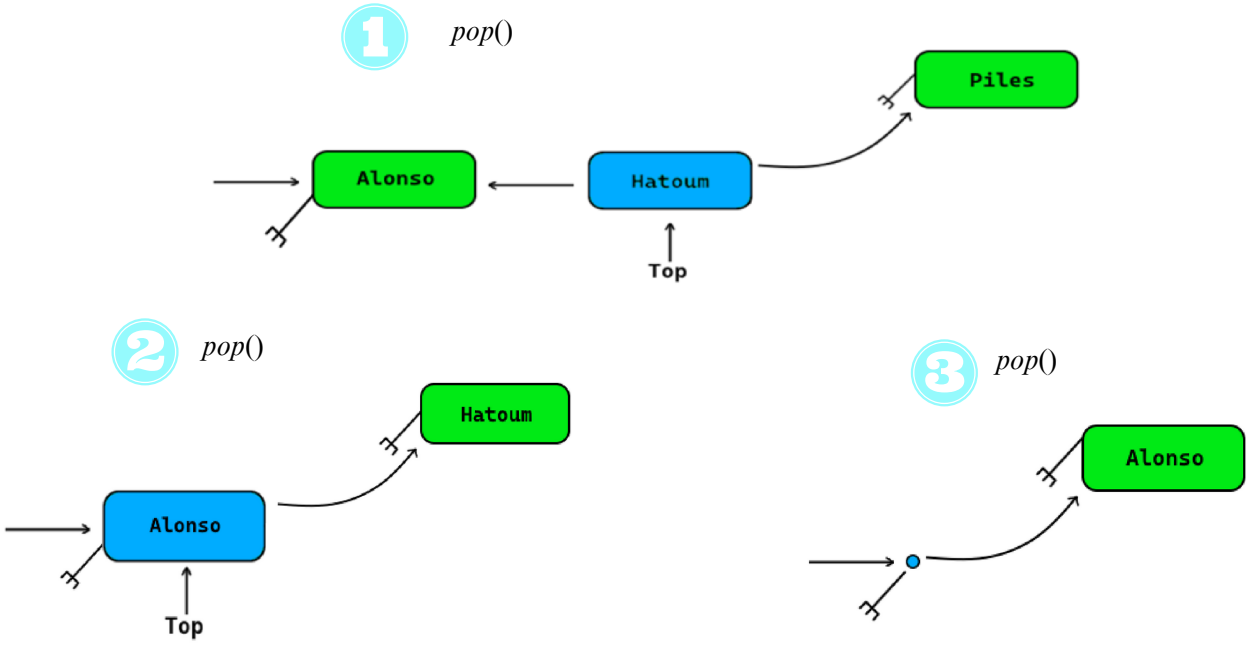


4. On *push("Piles")*



La méthode *pop( )* permet de récupérer et retirer le premier nœud correspondant au top de la *pile*. L'algorithme se déroule comme suit :

- On vérifie si le *top* est différent de null
  - Si c'est le cas, on affecte la valeur de l'élément *de type T* du nœud *top* à une variable, puis on change la référence du *top* actuel en le faisant pointer sur le nœud suivant du *top*. Finalement on retourne la valeur sauvegardée dans la variable temporaire.
  - Sinon, on lève une exception, indiquant que la pile est vide.



La méthode *top( )* est utilisée pour lire la valeur de l'élément *de type T* du nœud *top* de la pile. L'algorithme se déroule comme suit :

- On vérifie si le *top* est différent de null
  - Si c'est le cas, on retourne la valeur de l'élément *de type T* du nœud *top* correspondant au début de la liste chaînée.
  - Sinon, on lève une exception, indiquant que la pile est vide.

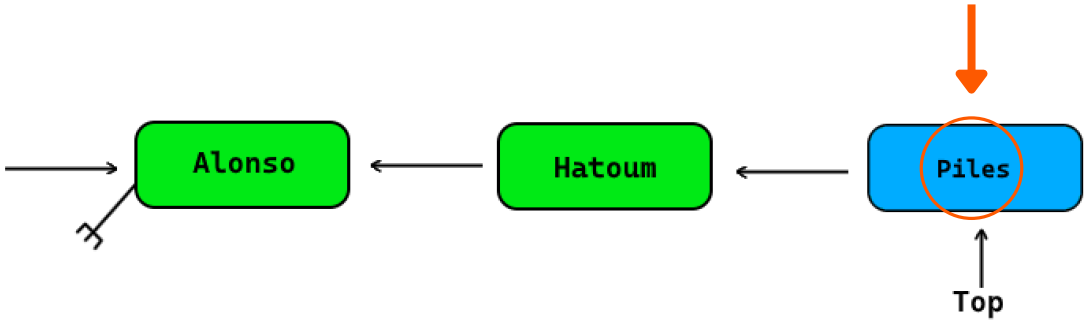
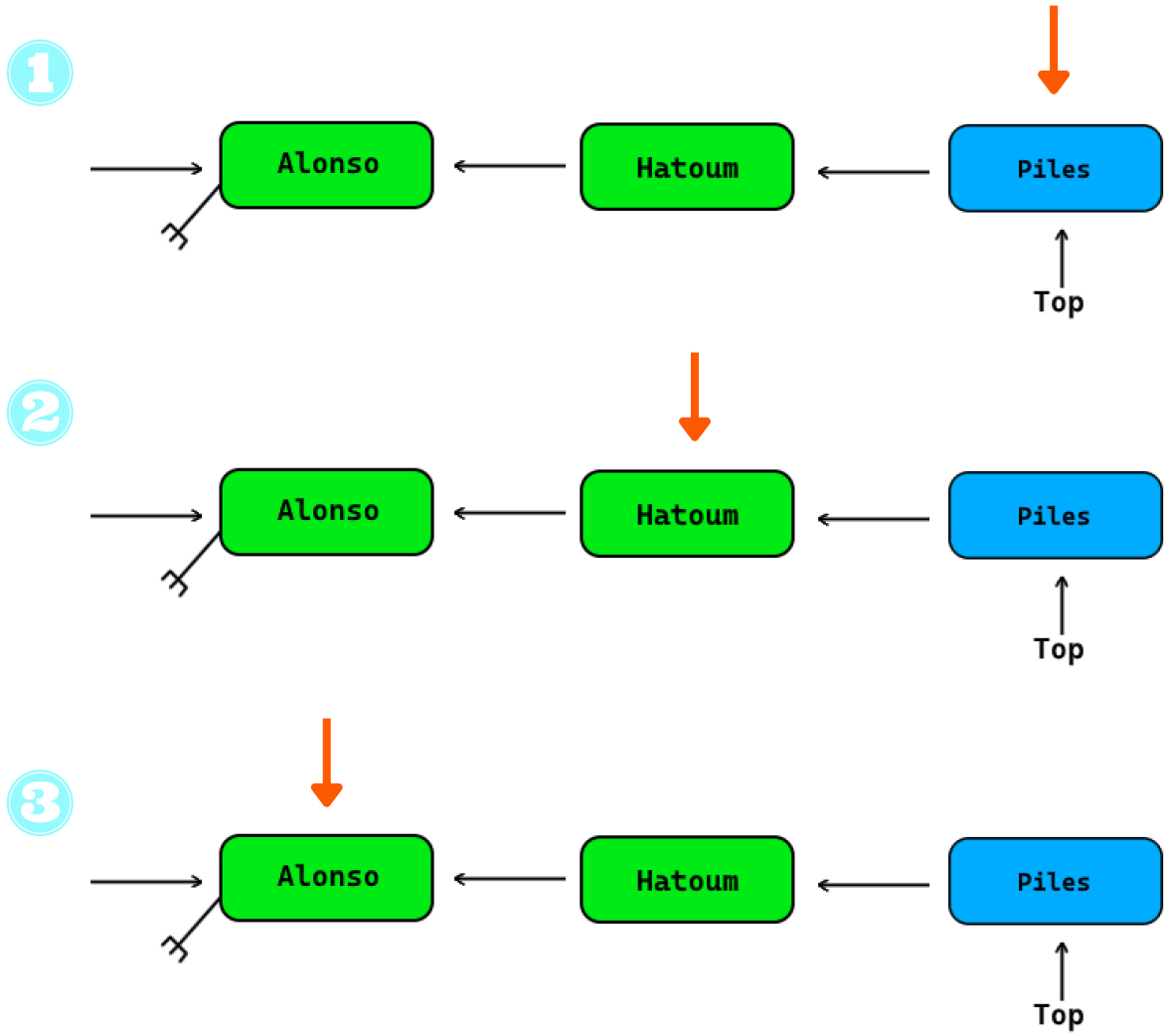


Schéma d'une instance de DynamicStack<String>.

La méthode *display( )* est utilisée pour afficher le contenu de la pile. On parcours la liste chaînée en commençant par le nœud *top*. L'algorithme se déroule comme suit :

- On vérifie si le *top* est différent de null
  - Si c'est le cas,
    - On effectue une boucle affichant chaque valeur de l'élément *de type T* pour chaque nœud de la liste chaînée.
  - Sinon, on lève une exception, indiquant que la pile est vide.



2. Le code réalisé

```java
package TP2.dynamicStack;

import TP2.stack.EmptyStackExceptions;
import TP2.stack.Stack;

/**
 * Implémentation d'une pile dynamique basée sur une liste chaînée.
 *
 * @param <T> Le type des éléments stockés dans la pile.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
1 usage  mrredcoding
public class DynamicStack<T> implements Stack<T> {

    /**
     * Représente un nœud dans la liste chaînée utilisée pour implémenter la pile.
     */
    5 usages  mrredcoding
    private class Node {
        4 usages
        private final T val;
        4 usages
        private Node next;

        /**
         * Construit un nouveau nœud avec la valeur spécifiée.
         *
         * @param value La valeur du nœud.
         */
        1 usage  mrredcoding
        Node(T value) {
            val = value;
            next = null;
        }
    }

    10 usages
    private Node top;

    /**
     * Constructeur de la classe DynamicStack.
     * Initialise une nouvelle pile dynamique vide.
     */
    1 usage  mrredcoding
    public DynamicStack() {
        this.top = null;
    }

    9 usages  mrredcoding
    @Override
    public void push(T value) {
        Node newNode = new Node(value);
        newNode.next = top;
        top = newNode;
    }

    1 usage  mrredcoding
    @Override
    public T pop() throws EmptyStackExceptions {
        if (top != null) {
            T value = top.val;
            top = top.next;
            return value;
        } else {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
    }
```

```
    no usages  mrredcoding
    @Override
    public T top() throws EmptyStackExceptions {
        if (top != null) {
            return top.val;
        } else {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
    }

    1 usage  mrredcoding
    @Override
    public void display() throws EmptyStackExceptions {
        Node current = top;
        if (current == null) {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
        while (current != null) {
            System.out.println(current.val);
            current = current.next;
        }
        System.out.println();
    }
}
```

3. Les tests et résultats de l'implémentation

```java
package TP2.dynamicStack;

import TP2.stack.Stack;
import TP2.utils.Test;

/**
 * Programme de test pour la classe DynamicStack.
 * Crée une pile dynamique, effectue des opérations de push et pop, et affiche les résultats.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
mrredcoding
public class Main {

    /**
     * Méthode principale du programme de test.
     *
     * @param args Les arguments de la ligne de commande (non utilisés dans ce programme).
     */
    mrredcoding
    public static void main(String[] args) {
        // Crée une pile dynamique de chaînes.
        Stack<String> dynamicStack = new DynamicStack<>();

        // Affiche une boîte de titre stylisée avec le nom de la classe de la pile.
        System.out.println(Test.titleBox(dynamicStack.getClass().getSimpleName()));

        // Effectue des opérations de push sur la pile.
        dynamicStack.push( value: "Alonso");
        dynamicStack.push( value: "Hatoum");
        dynamicStack.push( value: "Piles");

        // Affiche le contenu de la pile.
        Test.display(dynamicStack);

        // Effectue une opération de pop et affiche le résultat.
        Test.pop(dynamicStack);

        // Affiche à nouveau le contenu de la pile après la première opération de pop.
        Test.display(dynamicStack);

        // Affiche le sommet de la pile sans le retirer.
        Test.top(dynamicStack);

        // Effectue une deuxième opération de pop et affiche le résultat.
        Test.pop(dynamicStack);

        // Affiche à nouveau le contenu de la pile après la deuxième opération de pop.
        Test.display(dynamicStack);

        // Effectue une troisième opération de pop (pile vide) et affiche le résultat.
        Test.pop(dynamicStack);

        // Affiche à nouveau le contenu de la pile après la troisième opération de pop.
        Test.display(dynamicStack);
    }
}
```

```
┌─────────────┐
│ DynamicStack │
└─────────────┘
DynamicStack Contents :
Piles
Hatoum
Alonso


DynamicStack Popped value = Piles

DynamicStack Contents :
Hatoum
Alonso


DynamicStack Top value = Hatoum

DynamicStack Popped value = Hatoum

DynamicStack Contents :
Alonso


DynamicStack Popped value = Alonso

DynamicStack Contents :
DynamicStack is empty !
```

III. Pile fichier

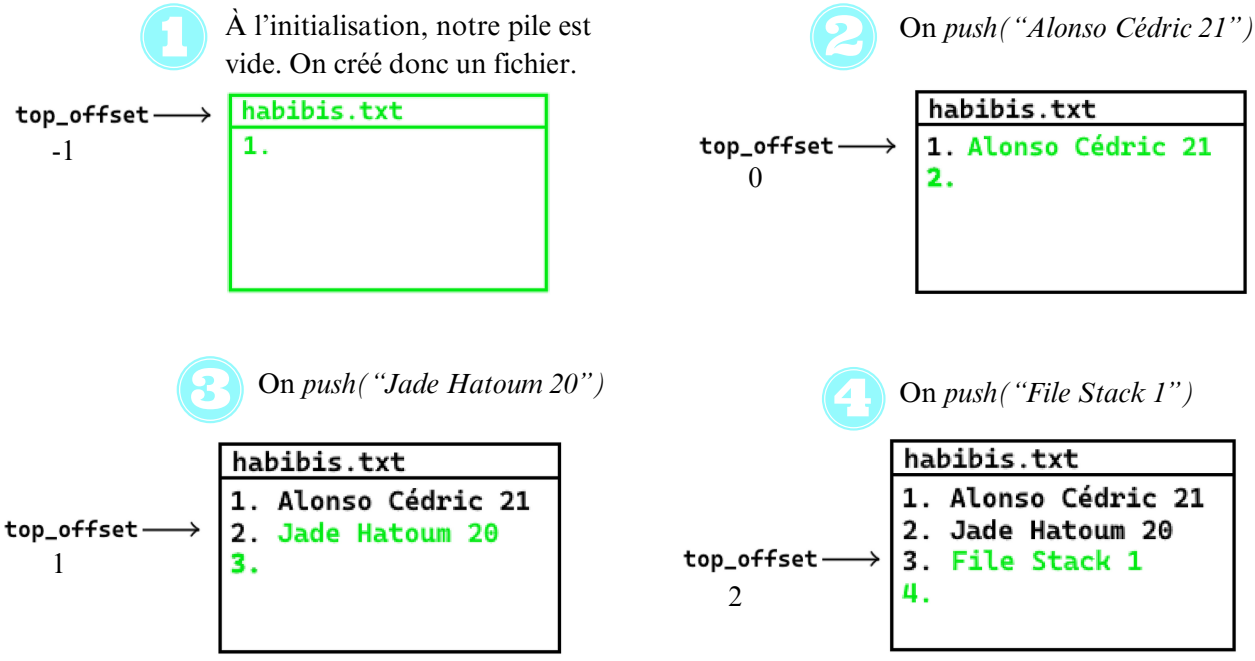
1. Algorithme explicité

Dans le cas de la pile basée sur un fichier, le fonctionnement général diffère de la pile statique classique. Voici une explication détaillée du fonctionnement général de la pile utilisant un fichier :

- La pile est implémentée à l'aide d'un fichier. Chaque élément de la pile est stocké comme une ligne dans ce fichier.
- L'attribut *filename* représente le nom du fichier utilisé pour stocker les éléments de la pile.

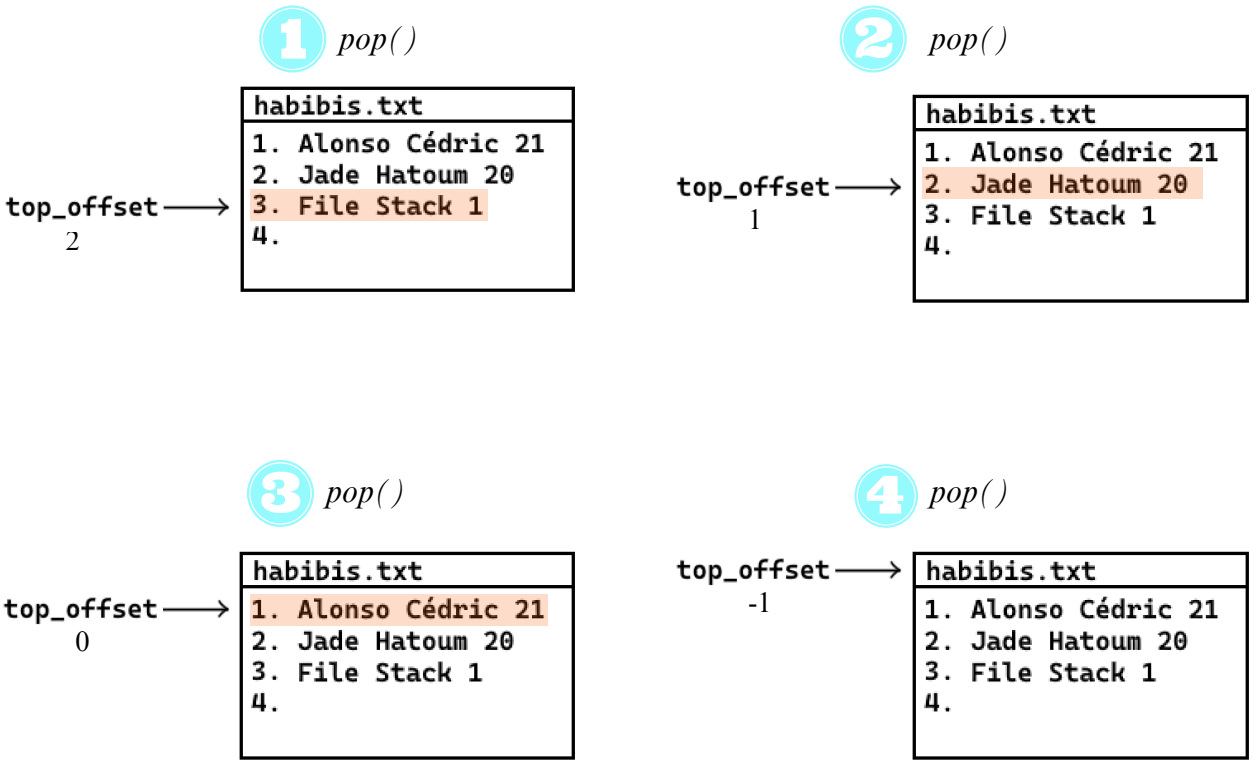
La méthode *push( )* vise à ajouter un nouvel élément de type T au sommet de la pile. L'algorithme se déroule comme suit :

- Ouverture du fichier en écriture :
  - On utilise un *BufferedWriter* pour écrire dans le fichier associé à la pile.
  - Le paramètre *true* dans le constructeur de *FileWriter* indique que l'écriture se fera en mode ajout à la fin du fichier.
- Écriture de la valeur dans le fichier :
  - La valeur de l'élément est convertie en chaîne de caractères avec *value.toString( )*.
  - La valeur est écrite dans une nouvelle ligne du fichier avec *writer.write(value.toString( ))* suivi de *writer.newLine( )*.
- Mise à jour de l'indice du sommet de la pile :
  - L'indice *top_offset* est incrémenté pour indiquer le nouveau sommet de la pile.
- Gestion des erreurs d'entrée/sortie :
  - Les exceptions *IOException* sont capturées et affichent un message d'erreur standard en cas de problème lors de l'écriture dans le fichier.



La méthode *pop( )* vise à retirer l'élément du sommet de la pile et le renvoyer. L'algorithme se déroule comme suit :

- Vérification de la pile vide :
  - On vérifie si l'indice *top_offset* est inférieur à zéro, ce qui signifie que la pile est vide.
  - Si la pile est vide, une exception *EmptyStackExceptions* est levée.
- Lecture des lignes du fichier :
  - On utilise la méthode privée *readFile( )* pour lire toutes les lignes du fichier et les stocker dans une liste de chaînes.
- Récupération de l'élément au sommet de la pile :
  - L'élément au sommet de la pile est obtenu à partir de la liste *lines* à l'indice *top_offset*.
- Mise à jour de l'indice du sommet de la pile (*top_offset*) :
  - L'indice *top_offset* est décrémenté pour indiquer le nouveau sommet de la pile.
- Écriture des lignes mises à jour dans le fichier :
  - La méthode privée *writeFile( )* est utilisée pour réécrire toutes les lignes dans le fichier, excluant ainsi l'élément retiré.
- Retour de l'élément retiré :
  - L'élément retiré est retourné.
- Gestion des erreurs d'exception :
  - L' exception *EmptyStackExceptions* est levée en cas de pile vide.
  - Les exceptions *IOException* sont capturées et affichent un message d'erreur standard en cas de problème lors de la lecture ou de l'écriture dans le fichier.



La méthode *top( )* vise à renvoyer l'élément situé au sommet de la pile sans le retirer. L'algorithme se déroule comme suit :

- Vérification de la pile vide :
  - On vérifie si l'indice *top_offset* est inférieur à zéro, ce qui signifie que la pile est vide.
  - Si la pile est vide, une exception *EmptyStackExceptions* est levée.
- Lecture des lignes du fichier :
  - On utilise la méthode privée *readFile( )* pour lire toutes les lignes du fichier et les stocker dans une liste de chaînes.
- Récupération de l'élément au sommet de la pile :
  - L'élément au sommet de la pile est obtenu à partir de la liste *lines* à l'indice *top_offset*.
- Retour de l'élément au sommet de la pile :
  - L'élément au sommet de la pile est retourné.
- Gestion des erreurs d'exception :
  - L'exception *EmptyStackExceptions* est levée en cas de pile vide.
  - Les exceptions *IOException* sont capturées et affichent un message d'erreur standard en cas de problème lors de la lecture dans le fichier.

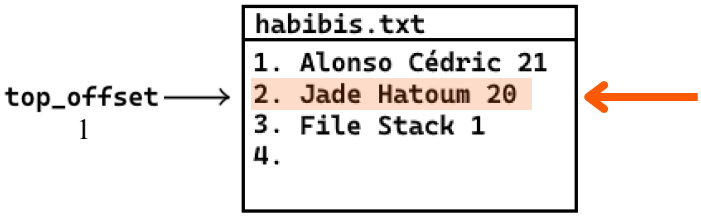
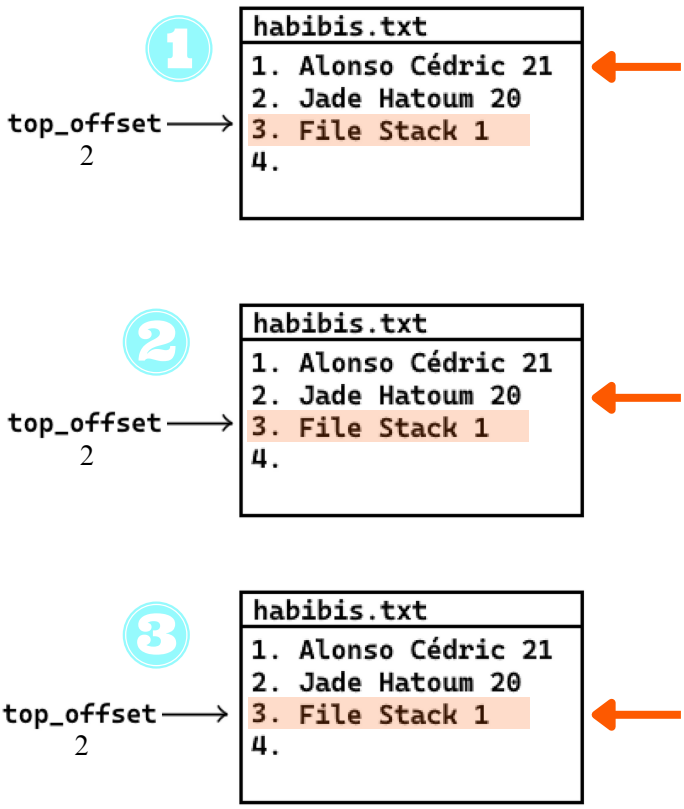


Schéma d'une instance de FileStack<String>.

La méthode *display( )* vise à afficher tous les éléments de la pile. L'algorithme se déroule comme suit :

- Vérification de la pile vide :
  - On vérifie si l'indice *top_offset* est inférieur à zéro, ce qui signifie que la pile est vide.
  - Si la pile est vide, une exception *EmptyStackExceptions* est levée.
- Affichage du nom du fichier associé à la pile :
  - Le nom du fichier est affiché.
- Lecture des lignes du fichier :
  - On utilise la méthode privée *readFile( )* pour lire toutes les lignes du fichier et les stocker dans une liste de chaînes.
- Affichage de chaque élément de la pile :
  - On parcourt la liste *lines* jusqu'à l'indice *top_offset* inclus, et on affiche chaque élément.
- Gestion des erreurs d'exception :
  - L'exception *EmptyStackExceptions* est levée en cas de pile vide.
  - Les exceptions *IOException* sont capturées et affichent un message d'erreur standard en cas de problème lors de la lecture dans le fichier.



La méthode *setFileName( )* vise à modifier le nom du fichier utilisé pour stocker les éléments de la pile et à renommer également le fichier sur le disque. L'algorithme se déroule comme suit :

- Création des objets *File* pour l'ancien et le nouveau fichier :
  - On crée un objet File pour l'ancien fichier *oldFile* et un objet *File* pour le nouveau fichier *newFile* avec le nouveau nom.
- Vérification de l'existence du fichier d'origine :
  - On vérifie si l'ancien fichier existe. Si non, une exception *FileException* est levée indiquant que le fichier d'origine n'existe pas.
- Renommage du fichier sur le disque :
  - On utilise la méthode *renameTo( )* pour renommer l'ancien fichier avec le nouveau nom.
  - Si le renommage est réussi, on met à jour l'attribut *filename* avec le nouveau nom.
  - Sinon, une exception *FileException* est levée indiquant qu'il est impossible de renommer le fichier.

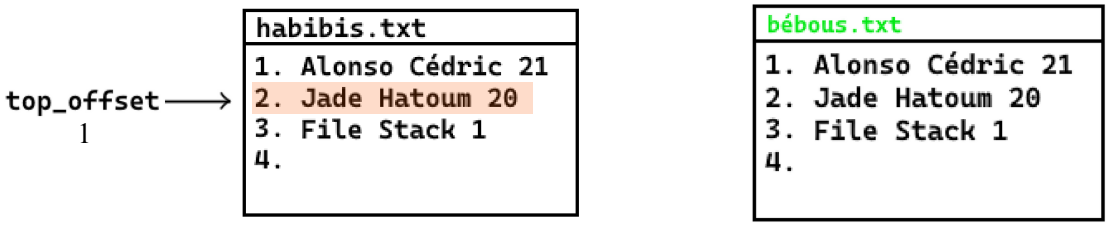


Schéma d'une instance de FileStack<String> où l'on appel *setFileName("bébous.txt").*

2. Le code réalisé

```java
package TP2.fileStack;

import TP2.stack.EmptyStackExceptions;
import TP2.stack.FileException;
import TP2.stack.Stack;

import java.io.*;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

/**
 * Implémentation d'une pile basée sur un fichier.
 *
 * @param <T> Le type des éléments stockés dans la pile.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
2 usages  mrredcoding +1 *
public class FileStack<T> implements Stack<T> {

    7 usages
    private String filename;
    9 usages
    private int top_offset;

    /**
     * Constructeur de la classe FileStack.
     *
     * @param filename Le nom du fichier utilisé pour stocker les éléments de la pile.
     */
    1 usage  mrredcoding
    public FileStack(String filename) {
        this.filename = filename;
        this.top_offset = -1;
    }
    9 usages  mrredcoding
    @Override
    public void push(T value) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename,  append: true))) {
            writer.write(value.toString());
            writer.newLine();
            top_offset++;
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }

    1 usage  mrredcoding
    @Override
    public T pop() throws EmptyStackExceptions {
        if (top_offset < 0) {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
        List<String> lines = readFile();
        assert lines != null;
        T topElement = (T) lines.get(top_offset);
        top_offset--;
        writeFile(lines);
        return topElement;
    }

    1 usage  mrredcoding
    @Override
    public T top() throws EmptyStackExceptions {
        if (top_offset < 0) {
            throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
        }
        return (T) Objects.requireNonNull(readFile()).get(top_offset);
    }
```

```java
1 usage  mrredcoding +1 *
@Override
public void display() throws EmptyStackExceptions {
    if (top_offset < 0) {
        throw new EmptyStackExceptions(this.getClass().getSimpleName() + " is empty !");
    }

    System.out.println("Nom du fichier : " + this.filename);
    List<String> lines = readFile();
    assert lines != null;

    for (int i = 0; i <= top_offset; i++) {
        System.out.println("\t- " + lines.get(i));
    }
}

/**
 * Lit les lignes du fichier associé à la pile.
 *
 * @return Une liste de chaînes représentant les lignes du fichier.
 *         Retourne null en cas d'erreur de lecture.
 */
3 usages  mrredcoding
private List<String> readFile() {
    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
        return reader.lines().collect(Collectors.toList());
    } catch (IOException e) {
        System.err.println(e.getMessage());
        return null;
    }
}
/**
 * Écrit les lignes fournies dans le fichier associé à la pile.
 *
 * @param lines La liste de chaînes à écrire dans le fichier.
 *              Chaque élément de la liste représente une ligne du fichier.
 */
1 usage  mrredcoding
private void writeFile(List<String> lines) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
        for (String line : lines) {
            writer.write(line);
            writer.newLine();
        }
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}


/**
 * Modifie le nom du fichier utilisé pour stocker les éléments de la pile.
 * Renomme également le fichier sur le disque.
 *
 * @param fileName Le nouveau nom du fichier.
 */
1 usage  mrredcoding
public void setFilename(String fileName) throws FileException {
    File oldFile = new File(this.filename);
    File newFile = new File(fileName);

    if (oldFile.exists())
        if (oldFile.renameTo(newFile))
            this.filename = fileName;
        else
            throw new FileException("Impossible de renommer le fichier.");

    else
        throw new FileException("Le fichier d'origine n'existe pas.");

}
}
```

3. Les tests et résultats de l'implémentation

```java
package TP2.fileStack;

import TP2.stack.FileException;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import TP2.utils.Test;

/**
 * Programme de test pour la classe FileStack.
 * Crée une pile basée sur un fichier, effectue des opérations de push et pop, et affiche les résultats.
 * @version 1.0
 * @author Cédric Alonso
 * @author Jade Hatoum
 */
mrredcoding +1
public class Main {

    /**
     * Méthode principale du programme de test.
     *
     * @param args Les arguments de la ligne de commande (non utilisés dans ce programme).
     */
    mrredcoding +1
    public static void main(String[] args) {
        // Nom du fichier utilisé pour stocker les éléments de la pile.
        String fileName = "habibis.txt";

        // Réinitialise le contenu du fichier à une chaîne vide.
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName))) {
            writer.write( str: "");
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }

        // Crée une pile basée sur un fichier avec des chaînes comme éléments.
        FileStack<String> fileStack = new FileStack<>(fileName);

        // Affiche une boîte de titre stylisée avec le nom de la classe de la pile.
        System.out.println(Test.titleBox(fileStack.getClass().getSimpleName()));
        // Effectue des opérations de push sur la pile.
        fileStack.push( value: "Alonso Cédric 21");
        fileStack.push( value: "Jade Hatoum 20");
        fileStack.push( value: "File Stack 1");

        // Affiche le contenu de la pile.
        Test.display(fileStack);

        // Effectue une opération de pop et affiche le résultat.
        Test.pop(fileStack);

        // Affiche à nouveau le contenu de la pile après la première opération de pop.
        Test.display(fileStack);

        // Affiche le sommet de la pile sans le retirer.
        Test.top(fileStack);

        // Effectue une deuxième opération de pop et affiche le résultat.
        Test.pop(fileStack);

        try{
            fileStack.setFilename("bébous.txt");
        } catch (FileException e){
            System.out.println(e.getMessage());
        }

        // Affiche à nouveau le contenu de la pile après la deuxième opération de pop.
        Test.display(fileStack);

        // Effectue une troisième opération de pop (pile vide) et affiche le résultat.
        Test.pop(fileStack);

        // Affiche à nouveau le contenu de la pile après la troisième opération de pop.
        Test.display(fileStack);
    }
}
```

```
| FileStack |

FileStack Contents :
Nom du fichier : habibis.txt
    - Alonso Cédric 21
    - Jade Hatoum 20
    - File Stack 1

FileStack Popped value = File Stack 1

FileStack Contents :
Nom du fichier : habibis.txt
    - Alonso Cédric 21
    - Jade Hatoum 20

FileStack Top value = Jade Hatoum 20

FileStack Popped value = Jade Hatoum 20

FileStack Contents :
Nom du fichier : bébous.txt
    - Alonso Cédric 21

FileStack Popped value = Alonso Cédric 21

FileStack Contents :
FileStack is empty !
```

efrei

PARIS PANTHÉON-ASSAS UNIVERSITÉ

Année académique 2023 - 2024