```java
package util;

/**
 * Classe conteneur permettant de lier un élément avec un potentiel élément suivant
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
class Element {
    Object value;
    Element next;

    /**
     * Constructeur basique d'un élément
     *
     * @param value valeur de l'élément
     * @param next  élément suivant, null si aucun
     */
    Element(Object value, Element next) {
        this.value = value;
        this.next = next;
    }
}
```

```java
package util;

/**
 * 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.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Stack {
    private Element top;
    private int size;

    /**
     * Ajoute un élément au sommet de la pile
     *
     * @param value : valeur du nouvel élément
     */
    public void push(Object value) {
        top = new Element(value, top);
        ++size;
    }

    /**
     * Supprime et retourne l'élément au sommet de la pile
     *
     * @return l'élément supprimé
     * @throws RuntimeException si la pile est vide
     */
    public Object pop() {
        if (top == null)
            throw new RuntimeException(
                    "Impossible de récupérer un élément d'une pile vide.");

        Object value = top.value;
        top = top.next;
        --size;
        return value;
    }
```

```
    /**
     * Retourne la pile sous forme d'un tableau des valeurs contenues
     *
     * @return le tableau de valeurs
     */
    public Object[] state() {
        Object[] result = new Object[size];

        StackIterator i = iterator();
        int index = 0;
        while (i.hasNext()) {
            result[index++] = i.next();
        }
        return result;
    }

    /**
     * Itérateur sur la pile
     *
     * @return un itérateur commençant au sommet de la pile
     */
    public StackIterator iterator() {
        return new StackIterator(top);
    }

    /**
     * Retourne la représentation du contenu de la pile
     *
     * @return la représentation sous forme de chaîne de caractères
     */
    @Override
    public String toString() {
        StackIterator i = iterator();
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        while (i.hasNext()) {
            sb.append(" <");
            sb.append(i.next());
            sb.append("> ");
        }
        sb.append("]");
        return sb.toString();
    }
}


package util;

/**
 * Classe représentant un itérateur pointant sur un élément.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class StackIterator {
    private Element element;

    /**
     * Constructeur de l'itérateur pointant sur le 1er élément
     *
     * @param element 1er élément
     */
    public StackIterator(Element element) {
        this.element = element;
    }

    /**
     * Vérifie que l'élément pointé existe
     *
     * @return true si l'élément existe, false sinon
     */
    public boolean hasNext() {
        return element != null;
    }
```

```
    /**
     * Fait avancer l'élément à l'élément suivant
     *
     * @return la valeur de l'élément courant
     * @throws RuntimeException s'il n'y a pas de prochain élément
     */
    public Object next() {
        if (!hasNext())
            throw new RuntimeException("Il n'y a pas d'élément suivant!");

        Element current = element;
        element = element.next;
        return current.value;
    }
}
```

```
package hanoi;

import util.Stack;

/**
 * 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.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Hanoi {
    private static final int NB_NEEDLES = 3;

    private final Stack[] needles;
    private final int nbDisks;
    private final HanoiDisplayer displayer;

    private int turns;

    /**
     * Constructeur principal. Est appelé dans les deux versions du programme:
     * - La version graphique
     * - La version console
     *
     * @param disk      le nombre de disques sur l'aiguille
     * @param displayer l'affichage choisi (graphique / console)
     * @throws RuntimeException en cas de nombre de disques invalide
     */
    public Hanoi(int disk, HanoiDisplayer displayer) {
        if (disk < 0)
            throw new RuntimeException("Le nombre de disques ne peut pas être " +
                    "négatif.");

        nbDisks = disk;
        this.displayer = displayer;
        this.turns = 0;

        needles = new Stack[NB_NEEDLES];
        for (int i = 0; i < NB_NEEDLES; i++) {
            needles[i] = new Stack();
        }

        // Ajout des disques sur la 1ère aiguille
        for (int i = nbDisks; i > 0; --i) {
            needles[0].push(i);
        }
    }
```

```
    /**
     * Constructeur pour l'affichage de la console
     *
     * @param disk le nombre de disques sur l'aiguille
     */
    public Hanoi(int disk) {
        this(disk, new HanoiDisplayer());
    }

    /**
     * Déplace tous les disques de la première aiguille à la troisième en
     * affichant les états successifs des aiguilles au moyen de l'instance
     * HanoiDisplayer sélectionnée.
     */
    public void solve() {
        this.displayer.display(this);
        this.hanoiAlgorithm(nbDisks, needles[0], needles[1], needles[2]);
    }

    /**
     * Implémentation de l'algorithme d'Hanoi sous forme récursive
     *
     * @param nbDisks      nombre de disques
     * @param start        aiguille de départ
     * @param intermediate aiguille du centre
     * @param finish       aiguille d'arrivée
     */
    private void hanoiAlgorithm(int nbDisks, Stack start, Stack intermediate, Stack finish) {
        if (nbDisks > 0) {
            this.hanoiAlgorithm(nbDisks - 1, start, finish, intermediate);
            this.move(start, finish);
            this.hanoiAlgorithm(nbDisks - 1, intermediate, start, finish);
        }
    }

    /**
     * Déplace le disque supérieur de l'aiguille source à l'aiguille de destination
     *
     * @param from l'aiguille source
     * @param to   l'aiguille de destination
     */
    private void move(Stack from, Stack to) {
        to.push(from.pop());
        ++this.turns;
        this.displayer.display(this);
    }

    /**
     * Rend un tableau de tableaux représentant l'état des aiguilles. Pour un tel
     * tableau t, l'élément t[i][j] correspond à la taille du j-ème disque (en
     * partant du haut) de la i-ème aiguille.
     *
     * @return l'état de chaque aiguille
     */
    public int[][] status() {
        int[][] result = new int[NB_NEEDLES][];

        for (int i = 0; i < NB_NEEDLES; i++) {
            Object[] state = needles[i].state();
            result[i] = new int[state.length];

            for (int j = 0; j < state.length; j++) {
                result[i][j] = (int) state[j];
            }
        }
        return result;
    }
```

```
    /**
     * Permet de vérifier si la solution est atteinte
     *
     * @return true si la solution du problème a été atteinte, false sinon
     */
    public boolean finished() {
        return turns == Math.pow(2, nbDisks) - 1;
    }

    /**
     * Rend le nombre de disques déplacés
     *
     * @return le nombre de disques déplacés
     */
    public int turn() {
        return turns;
    }

    /**
     * Retourne la représentation sous forme de chaîne de caractères des
     * états actuels des aiguilles
     *
     * @return la représentation de l'état actuel
     */
    @Override
    public String toString() {
        StringBuilder state = new StringBuilder();
        state.append("-- Turn: ").append(turn()).append("\n");

        for (int i = 0; i < NB_NEEDLES; i++) {
            state.append(String.format("%-5s", displayer.numberToWord(i)))
                    .append(" : ")
                    .append(needles[i]);

            if (i < NB_NEEDLES - 1)
                state.append("\n");
        }
        return state.toString();
    }
}


package hanoi;

/**
 * Classe permettant l'affichage des étapes de résolution du problème dans la console
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class HanoiDisplayer {
    private static final String[] NUMBERS = {"One", "Two", "Three"};

    /**
     * Transforme un chiffre dans sa représentation en anglais
     *
     * @param number chiffre à transformer
     * @return la représentation du chiffre
     */
    public String numberToWord(int number) {
        if (number >= NUMBERS.length)
            throw new RuntimeException("Index invalide!");

        return NUMBERS[number];
    }
```

```
    /**
     * Affiche l'état des aiguilles de l'instance de la classe Hanoi.
     * Par défaut l'affichage se fait dans la console.
     *
     * @param h instance du programme Hanoi en cours
     */
    public void display(Hanoi h) {
        System.out.println(h);
    }
}


import hanoi.gui.JHanoi;
import util.Stack;
import util.StackIterator;

/**
 * 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.
 *
 * @author Jonathan Friedli
 * @author Valentin Kaelin
 */
public class Hanoi {
    /**
     * Méthode appelée au lancement du programme, teste le nombre de paramètres et la
     * validité de ce dernier.
     * Si paramètre valide, résout la tour d'Hanoï avec le nombre de disques voulu.
     *
     * @param args 0 ou 1 paramètre:
     *             Si 0 paramètre, utilise l'interface graphique.
     *             Si 1 paramètre, utilise l'interface console.
     *             Le paramètre représente le nombre de disques.
     *             Il doit être positif.
     *             Si 2 paramètre ou plus, une exception est levée.
     * @throws Exception en cas d'entrée utilisateur incorrecte.
     */
    public static void main(String[] args) throws Exception {
        if (args.length == 0) {
            new JHanoi();
        } else {
            hanoi.Hanoi hanoi = new hanoi.Hanoi(testArgs(args));
            hanoi.solve();
        }

        // Enlever le commentaire si l'on souhaite tester la Stack.
        // testStack();
    }

    /**
     * Fonction permettant de tester que le paramètre passé par l'utilisateur
     * est correct (un entier > 0).
     *
     * @param args est le tableau d'argument passé par l'utilisateur.
     * @return La valeur passée par l'utilisateur, castée en int.
     * @throws Exception en cas d'entrée utilisateur erronée
     */
    private static int testArgs(String[] args) throws Exception {
        int numberOfDisk;
        if (args.length > 1)
            throw new Exception("Il ne faut qu'un seul argument (exemple: java Hanoi 7)");

        try {
            numberOfDisk = Integer.parseInt(args[0]);

        } catch (Exception e) {
            throw new Exception("L'argument doit être un entier positif.");
        }

        if (numberOfDisk < 0)
            throw new Exception("L'argument doit être un entier positif");

        return numberOfDisk;
    }
```

```
    /**
     * Teste l'implémentation maison de la Stack
     */
    private static void testStack() {
        System.out.println("TEST: Création d'une stack vide.");
        Stack stack = new Stack();
        System.out.println("État de la stack: " + stack);
        System.out.println("\n");

        System.out.println("TEST: Remplissage d'une stack");
        System.out.println("Insertion de la valeur 4");
        stack.push(4);
        System.out.println("État de la stack: " + stack);
        System.out.println("Insertion de la valeur 5");
        stack.push(5);
        System.out.println("État de la stack: " + stack);
        System.out.println("Insertion de la valeur 6");
        stack.push(6);
        System.out.println("État de la stack: " + stack);
        System.out.println("\n");

        System.out.println("TEST: Itérateur fonctionnel");
        StackIterator si = stack.iterator();
        System.out.print("Les valeurs contenues dans la stack sont: ");
        while (si.hasNext()) {
            System.out.print(si.next() + " ");
        }
        System.out.println("\n\n");

        System.out.println("TEST: Récupérer l'état de la stack via state()");
        Object[] stackValues = stack.state();
        System.out.print("Les valeurs contenues dans la stack sont: ");
        for (Object o : stackValues) {
            System.out.print(o + " ");
        }
        System.out.println("\n\n");

        System.out.println("TEST: Récupérer l'état de la stack ne brise pas " +
                "l'encapsulation");
        stackValues[0] = 10;
        System.out.println("État de la stack: " + stack);
        System.out.println("\n");

        System.out.println("TEST: Insertion de String");
        stack.push("Je suis un string");
        System.out.println("État de la stack: " + stack);
        System.out.println("\n");


        System.out.println("TEST: Vidage de la stack");
        for (int i = 0; i < 4; ++i) {
            System.out.println("Valeur supprimée: " + stack.pop());
            System.out.println("État de la stack: " + stack);
        }
        System.out.println("\n");

        System.out.println("TEST: Pop une stack vide");
        try {
            System.out.println("Valeur supprimée: " + stack.pop());
        } catch (RuntimeException e) {
            System.out.println("Exception bien levée.");
        }
    }
}
```