```java
import hanoi.Hanoi;
import hanoi.gui.JHanoi;


/**
 * Permet de lancer le programme en ligne de commande ou avec l'interface graphique
 *
 * @author Bogale Tegest & Jallon Sarah
 */
public class Main {
    /**
     * Lance le programme en ligne de commande ou avec l'interface graphique
     * @param args arguments donnés en ligne de commande
     * @throws RuntimeException si le nombre de disques est < 1 ou si hanoi Displayer est null
     *
     */
    public static void main(String[] args) {
        try{
            if (args.length == 1) {
                Hanoi hanoi = new Hanoi(Integer.parseInt(args[0]));
                hanoi.solve();
            } else if (args.length == 0) {
                new JHanoi();
            } else {
                System.out.println("To use the interface run the program without arguments");
                System.out.println("To use the console run the program with the number of disks");
            }
        }catch (RuntimeException e){
            System.out.println(e.getMessage());
        }

    }
}
```

```java
package test;

import hanoi.Hanoi;
import hanoi.HanoiDisplayer;

import java.util.Arrays;

/**
 * Classe de test mettant en œuvre toutes les fonctionnalités de la classe Hanoi
 *
 * @author Bogale Tegest & Jallon Sarah
 */
public class TestHanoi {
    private static final HanoiDisplayer hanoiDisplayer = new HanoiDisplayer();
    private static Hanoi hanoi;

    public static void main(String[] args) {

        System.out.println("Test the construction of hanoi with a given disks");
        testHanoi();
        System.out.println("Test the algorithme used");
        testSolve();
        System.out.println("Tests the representation of hanoi in an array of 2 dimensions");
        testStatus();
        System.out.println("Hanoi with number of disks less than 1 should raise a RuntimeException: ");
        testWrongDisk();
    }

    /**
     * Test la bonne construction de la tour de hanoi
     */
    private static void testHanoi() {
        hanoi = new Hanoi(4);
        System.out.println("Hanoi : ");
        hanoiDisplayer.display(hanoi);

    }

    /**
     * Construction de la tour de hanoi avec un nombre de disques < 1
     */
    private static void testWrongDisk() {
        try {
            hanoi = new Hanoi(0);
        } catch (RuntimeException e) {
            System.out.println(e.getMessage());
        }

    }

    /**
     * Teste l'algorithme pour la résolution et la méthode finished()
     */
    private static void testSolve() {
        hanoi = new Hanoi(4);
        System.out.println("Solve : ");
        hanoi.solve();
        System.out.println("When solved the value of finished is true : " + hanoi.finished());

    }

    /**
     * Teste la représentation en tableau de tableau de la pile
     */
    private static void testStatus() {
        hanoi = new Hanoi(5);
        System.out.println("Status : " + Arrays.deepToString(hanoi.status()));
    }


}

```

```java
package test;

import util.*;

import java.util.Arrays;
import java.util.EmptyStackException;


/**
 * Classe de test mettant en œuvre toutes les fonctionnalités des classes du package util,
 * soit Stack, Element et Examinator
 * @author Bogale Tegest & Jallon Sarah
 */
public class TestStack {
    private static final String passed = "Passed";
    private static final String failed = "Failed";

    public static void main(String[] args) {
        System.out.println();
        System.out.println("********************** TEST push(Object o) & toString() **********************");
        testPush();
        System.out.println();
        System.out.println("********************** TEST pop() ********************************************");
        testPop();
        System.out.println();
        System.out.println("********************** TEST hasNext() ****************************************");
        testHasNext();
        System.out.println();
        System.out.println("********************** TEST stackToArray() ***********************************");
        testStackToArray();
        System.out.println("********************** TEST next() *******************************************");
        testNext();

    }

    /**
     * Permet de tester l'empilement des valeurs correctes, incorrectes, null ainsi que la représentation sous la
     * forme de chaîne de caractère du contenu de la pile
     */
    private static void testPush() {
        Stack s = creatTestStack(4);
        String correct = "[ <1> <2> <3> <4> ]";
        String wrongStringRep = "[ <1 <2> <3> <4> ]";
        String inCorrect = "[ <2> <3> <4> ]";

        Stack emptyStack = creatTestStack(0);
        System.out.println("The representation of an empty stack is [ ] = " + emptyStack);
        System.out.println("Wrong string representation should fail: "
                + (testResultS(s.toString(), wrongStringRep).equals(failed) ? passed : failed));
        System.out.println("Creation of a stack with correct expected values: "
                + testResultS(s.toString(), correct));
        System.out.println("Creation of a stack with incorrect expected values should not work: "
                + (testResultS(s.toString(), inCorrect).equals(failed) ? passed : failed));
        System.out.println("Creation of a stack with correct values and correct string representation: "
                + testResultS(correct, s.toString()));

        s.push(null);
        System.out.println("Pushing \"null\" object to the stack is possible. Result: " + s);

    }

    /**
     * Permet de tester le désempilement des valeurs correctes, incorrectes et null
     */
    private static void testPop() {
        Stack s = creatTestStack(2);
        s.pop();
        String correct = "[ <2> ]";
        String inCorrect = "[ <1> <2> ]";

        System.out.println("Pop from a non empty stack works: " + testResultS(s.toString(), correct));
        System.out.println("The stack value after a pop is different from the original and is correct: "
                + (testResultS(s.toString(), inCorrect).equals(failed) ? passed : failed));
        System.out.println("Pop on empty stack should throw EmptyStackException (null) : ");
        try {
            s.pop();
            s.pop();
        } catch (EmptyStackException e) {
            System.out.println(e.getMessage());
        }

        Stack nullS = creatTestStack(0);
        nullS.push(null);
        System.out.println("We can pop a null value. Before: " + nullS);
        nullS.pop();
        System.out.println("After pop(): " + nullS);

    }

```

```
90      /**
91       * Test le fonctionnement de hasNext()
92       */
93
94      private static void testHasNext() {
95          Stack s = creatTestStack(3);
96          Examinator e = s.examinator();
97          int cnt = 0;
98          while (e.hasNext()) {
99              e.next();
100             ++cnt;
101         }
102         System.out.println("hasNext iterates until the stack is empty: "
103                 + testResult(cnt, s.size()));
104
105         Examinator e2 = s.examinator();
106         for (int i = 0; i < 10; ++i) {
107             if (e2.hasNext()) {
108                 ++cnt;
109             }
110         }
111         System.out.println("The iterator doesn't overflow : "
112                 + (testResult(cnt, s.size()).equals(failed) ? passed : failed));
113
114     }
115
116     /**
117      * Test le fonctionnement de hasNext()
118      */
119
120     private static void testNext() {
121         Stack s = creatTestStack(3);
122         Examinator e = s.examinator();
123
124         System.out.println("Printing stack values using next() works as expected: ");
125         for (int i = 0; i < s.size(); ++i) {
126             System.out.println(e.next());
127         }
128         System.out.println("If the iterator overflow an EmptyStackException is raised(null): ");
129         try {
130             for (int j = 0; j < s.size() + 3; ++j) {
131                 System.out.println(e.next());
132             }
133         } catch (EmptyStackException ex) {
134             System.out.println(ex.getMessage());
135         }
136
137     }
138
139     /**
140      * Test le fonctionnement de stackToArray
141      */
142     private static void testStackToArray() {
143         Stack s = creatTestStack(4);
144         String correctArray = "[1, 2, 3, 4]";
145         String incorrectArray = "[1 2 3 4]";
146         System.out.println("Check with correct array: " +
147                 testResultS(Arrays.toString(s.stackToArray()), correctArray));
148         System.out.println("Check with incorrect array: " +
149                 (testResultS(Arrays.toString(s.stackToArray()), incorrectArray).equals(failed) ? passed : failed));
150
151         Stack empty = creatTestStack(0);
152         System.out.println("Representation of empty stack to array: " + Arrays.toString(empty.stackToArray()));
153     }
154
155     /**
156      * Permet de créer et remplir une pile
157      */
158     private static Stack creatTestStack(int size) {
159         Stack s = new Stack();
160         for (int i = size; i > 0; --i) {
161             s.push(i);
162         }
163         return s;
164     }
165
166     /**
167      * Permet de comparer la sortie avec le bon affichage
168      */
169     private static String testResultS(String a, String b) {
170         if (a.equals(b)) return passed;
171         else return failed;
172     }
173
174     /**
175      * Permet de comparer si on a le bon nombre de résultats.
176      */
177     private static String testResult(int a, int b) {
178         if (a == b)
```

```
179             return passed;
180         else return failed;
181     }
182
183 }
184
```

```java
package util;

import java.util.EmptyStackException;

/**
 * Classe qui implémente la pile
 *
 * @author Bogale Tegest & Jallon Sarah
 */
public class Stack {
    private Element top = null;
    private int size;

    /**
     * Permet d'empiler un objet sur le sommet de la pile.
     * Il est possible d'empiler un objet null
     *
     * @param o l'objet à empiler
     */
    public void push(Object o) {
        top = new Element(o, top);
        ++size;
    }

    /**
     * Permet de désempiler un objet du sommet de la pile
     *
     * @return retourne l'objet désempilé
     * @throws EmptyStackException si la pile est vide
     */
    public Object pop() {
        if (top == null) {
            throw new EmptyStackException();
        }
        Object o = top.data;
        top = top.next;
        --size;
        return o;
    }

    /**
     * Permet de redéfinir la méthode toString() afin d'obtenir la représentation sous la forme de
     * chaîne de caractères du contenu de la pile,
     *
     * @return rend le contenu de la pile sous forme de chaine de caractère
     */
    @Override
    public String toString() {
        Examinator ex = examinator();
        StringBuilder sb = new StringBuilder();
        sb.append("[ ");
        while (ex.hasNext()) {
            sb.append("<").append(ex.next()).append(">").append(" ");
        }
        sb.append("]");
        return sb.toString();
    }

    /**
     * Permet d'obtenir un tableau d’objets représentant
     * l’état actuel de la pile (l’indice 0 contenant l’élément placé au
     * sommet de la pile)
     *
     * @return le tableau d'objet représentant l'état actuel de la pile
     * @throws EmptyStackException s'il n'y a plus d'élément à itérer
     */
    public Object[] stackToArray() {
        Object[] tab = new Object[size];
        int i = 0;
        Examinator ex = examinator();
        while (ex.hasNext()) {
            tab[i++] = ex.next();
        }
        return tab;
    }

    /**
     * Permet de retourner un itérateur le sur le sommet de la pile
     *
     * @return un itérateur
     */
    public Examinator examinator() {
        return new Examinator(top);
    }

    /**
     * Permet de retourner la taille de la pile
     *
     * @return la taille de la pile
```

```java
     */
    public int size() {
        return size;
    }

    /**
     * Permet de savoir si la stack est vide
     *
     * @return vrai si vide, faux sinon
     */
    public boolean isEmpty() {
        return size == 0;
    }

}
```

```java
package util;

/**
 * Classe qui implémente les éléments à ajouter à la pile
 *
 * @author Bogale Tegest & Jallon Sarah
 */
class Element {
    Object data;
    Element next;

    /**
     * Constructeur avec l'objet stocké et l'élément suivant
     *
     * @param data l'objet stocké dans l'élément
     * @param next élément suivant l'actuel
     */
    Element(Object data, Element next) {
        this.data = data;
        this.next = next;
    }

}

```

```java
package util;

import java.util.EmptyStackException;

/**
 * Classe qui implémente l'itérateur de la pile
 *
 * @author Bogale Tegest & Jallon Sarah
 */
public class Examinator {
    private Element current;

    /**
     * Constructeur, avec l'élément de départ de l'itérateur.
     * À sa création, l'itérateur se réfère au début de la stack (avant le 1er élément).
     *
     * @param el élément sur lequel l'itérateur se réfère
     */
    Examinator(Element el) {
        current = el;
    }

    /**
     * Méthode qui rend l'élément suivant de l'itérateur
     *
     * @return retourne l'Objet stocké dans l'élément suivant de l'itérateur
     * @throws EmptyStackException s'il n'y a plus d'élément à itérer
     */
    public Object next() {
        if (!hasNext()) {
            throw new EmptyStackException();
        }
        Object o = current.data;
        current = current.next;
        return o;
    }

    /**
     * Méthode qui vérifie si l'élément sur lequel l'itérateur se réfère a un élément après
     *
     * @return rend true si l'itérateur n'est pas en fin de la pile, false sinon
     */
    public boolean hasNext() {
        return current != null;
    }
}

```

```java
package hanoi;

import util.Stack;

/**
 * Classe qui implémente les tours de hanoi et ses méthodes pour la résolution ainsi que l'affichage
 *
 * @author Bogale Tegest & Jallon Sarah
 */
public class Hanoi {
    private final Stack[] needles = new Stack[3];
    private final HanoiDisplayer hanoiDisplayer;
    private final int disks;
    private int cntTurn;

    /**
     * Constructeur pour l'affichage dans le GUI
     *
     * @param disks          nombre de disques à déplacer
     * @param hanoiDisplayer instance de hanoi displayer pour l'affichage
     * @throws RuntimeException si le nombre de disques est inférieur à 1 ou si hanoiDisplayer est null
     */
    public Hanoi(int disks, HanoiDisplayer hanoiDisplayer) {
        if (disks < 1) throw new RuntimeException("Error: number of disks should be greater than 0.");
        if (hanoiDisplayer == null) throw new RuntimeException("Error: hanoi displayer must be different from null.");
        this.disks = disks;
        this.hanoiDisplayer = hanoiDisplayer;

        for (int i = 0; i < needles.length; ++i) {
            needles[i] = new Stack();
        }
        for (int i = disks; i > 0; --i) {
            needles[0].push(i);
        }
    }

    /**
     * Constructeur pour l'affichage dans la console
     *
     * @param disks nombre de disques à déplacer
     * @throws RuntimeException si le nombre de disques est inférieur à 1 ou si hanoiDisplayer est null
     */
    public Hanoi(int disks) {
        this(disks, new HanoiDisplayer());
    }

    /**
     * Fonction de transfert récursive qui permet de déplacer n disques d'un baton d'origine à un
     * baton de destination en passant par un baton intermédiaire
     *
     * @param src  baton d'origine
     * @param by   baton intermédiaire
     * @param dest baton de destination
     * @param n    nombre de disques à déplacer
     */
    private void move(Stack src, Stack by, Stack dest, int n) {
        if (n < 1)
            return;
        move(src, dest, by, n - 1);
        dest.push(src.pop());
        ++cntTurn;
        hanoiDisplayer.display(this);
        move(by, src, dest, n - 1);
    }


    /**
     * Algorithme de résolution du problème des tours de Hanoi, repris dans le cours
     * de ASD2022.
     */
    public void solve() {
        hanoiDisplayer.display(this);
        move(needles[0], needles[1], needles[2], disks);
    }

    /**
     * Permet d'obtenir le statut courant des tours
     *
     * @return un tableau de tableaux représentant les 3 batons et leurs contenus
     */
    public int[][] status() {
        Object[][] hanoiStatus = new Object[needles.length][];
        int[][] t = new int[hanoiStatus.length][];
        for (int i = 0; i < t.length; ++i) {
            hanoiStatus[i] = needles[i].stackToArray();
            t[i] = new int[hanoiStatus[i].length];
            for (int j = 0; j < t[i].length; ++j) {
                t[i][j] = (int) hanoiStatus[i][j];
            }
```

```java
        }
        return t;
    }

    /**
     * Permet de savoir la résolution du problème des tours de hanoi est terminée
     *
     * @return vrai si terminé, faux sinon
     */
    public boolean finished() {
        return needles[0].isEmpty() && needles[1].isEmpty() && needles[2].size() == disks;

    }

    /**
     * Permet de connaître le nombre d'étapes déjà réalisées dans la résolution du
     * problème de Hanoi
     *
     * @return le nombre de l'étape courante
     */
    public int turn() {
        return cntTurn;
    }

    /**
     * Permet de redéfinir la méthode toString() afin d'obtenir la représentation sous la forme de
     * chaine de caractère des trois tours de Hanoi utilisant Stack.toString()
     *
     * @return rend le contenu des trois tours sous forme de chaine de caractères
     */
    @Override
    public String toString() {
        return "One:   " + needles[0] +
                "\nTwo:   " + needles[1] +
                "\nThree: " + needles[2];

    }
}

```

```java
package hanoi;

/**
 * Classe qui implémente l'affichage de hanoi sur la console
 *
 * @author Bogale Tegest & Jallon Sarah
 */
public class HanoiDisplayer {

    /**
     * Permet d'afficher l'état courant du problème de Hanoi dans la console
     *
     * @param h instance de Hanoi que nous souhaitons afficher
     */
    public void display(Hanoi h) {
        System.out.println("-- Turn: " + h.turn());
        System.out.println(h);
    }

}

```