```java
package test;

import hanoi.Hanoi;
import hanoi.HanoiDisplayer;
import util.Stack;
import java.util.ArrayList;
import java.util.List;

/**
 * This class is used to perform various tests on the Stack class and the HanoiDisplayer class
 *
 * @author Kevin Farine, Timothee Van Hove
 */
public class ProgramTest {

    /**
     * Number of successful tests
     */
    private static int successNb = 0;

    /**
     *Number of failed tests
     */
    private static int failedNb = 0;

    /**
     * Stack used to perform various tests
     */
    private final static Stack stack = new Stack();

    /**
     * Message to print in case of success
     */
    private final static String FAIL = " |Test failed|";

    /**
     * Message to print in case of failure
     */
    private final static String SUCCESS = " |Test succeeded|";


    public static void main(String[] args) {
        System.out.println("----------Test program----------");
        //Stack class tests
        emptyStackMustBePrintable();
        stackCanContainGenericObjects();
        iteratorMustReferenceTopItem();
        itemsMustBeIterableAndPrintable();
        stackMustBePrintable();
        stackMustReturnCorrectStateArray();
        stackMustBeWellEncapsulated();
        stackCanBeEmptied();
        poppingAnItemFromEmptyStackMustGenerateException();
        nextItemOnEmptyStackMustGenerateException();

        //Hanoi class tests
        multipleArgumentsMustThrowException();
        nonIntegerArgumentsMustThrowException();
        negativeValueArgumentsMustThrowException();
        hanoiAlgorithmMustBeSolvedWithCorrectTurnNumber();
        System.out.println("\nTest program finished :");
        System.out.println("  -" + String.format("%2s", successNb) + " test(s) passed");
        System.out.println("  -" + String.format("%2s", failedNb) + " test(s) failed");


    }

    /**
     * Prints the success message
     */
    private static void success(){
        successNb++;
```

```
        System.out.println(SUCCESS);
    }

    /**
     * Prints the success message with exception message
     * @param e Exception used tu print the additional message
     */
    private static void success(Exception e){
        System.out.print(e.getMessage());
        success();
    }

    /**
     * Prints the failure message
     */
    private static void fail(){
        failedNb ++;
        System.err.println(FAIL);
    }

    /**
     * Tests if an empty stack is printable
     */
    private static void emptyStackMustBePrintable(){
        System.out.print("Printing an empty stack : ");
        try{
            System.out.print(stack);
        }
        catch(Exception e){
            fail();
        }
        success();
    }

    /**
     * Tests if it is possible to push any object in the stack
     */
    private static void stackCanContainGenericObjects() {
        System.out.print("Populating stack with generic objects : ");
        List<Dog> dogs = new ArrayList<>();
        try{
            dogs.add(new Dog("Rex", 7));
            dogs.add(new Dog("Laika", 9));

            stack.push(dogs);
            stack.push(new Dog("Lassie", 4));
            stack.push("Hello world!");
            stack.push(42);
        }
        catch(Exception e){
            fail();
        }
        success();
    }

    /**
     * Tests if the items contained in the stack are printable
     */
    private static void itemsMustBeIterableAndPrintable() {
        System.out.print("Iterating and printing items of the stack : ");
        Stack.StackIterator iterator = stack.getIterator();
        try{
            while (iterator.hasNext()) {
                System.out.print(iterator.next() + " ");
            }
        }
        catch(Exception e){
            fail();
        }
        success();
    }
```

```
    /**
     * Tests if the whole stack is printable
     */
    private static void stackMustBePrintable() {
        System.out.print("Printing the stack : ");
        try {
            if(stack.toString().startsWith("[ <") && stack.toString().endsWith("> ]")){
                System.out.print(stack);
                success();
            }
            else{
                fail();
            }
        }
        catch (Exception e){
            fail();
        }
    }

    /**
     * Tests if the StackIterator given by the stack is referencing the top item of the stack
     */
    private static void iteratorMustReferenceTopItem(){
        System.out.print("Verifying the StackIterator reference : ");
        try {
            Stack.StackIterator iterator = stack.getIterator();
            if ((int) iterator.next() != 42) {
                fail();
                return;
            }
        }
        catch(Exception e){
            fail();
        }
        success();
    }

    /**
     * Tests if the returned current state array has the correct length
     */
    private static void stackMustReturnCorrectStateArray(){
        System.out.print("Verifying the length of the status array : ");
        if(stack.getCurrentState().length != 4){
            fail();
            return;
        }
        success();
    }

    /**
     * Tests if by modifying the objects contained in the current state array it modifies the
     * objects in the stack
     */
    private static void stackMustBeWellEncapsulated(){
        System.out.print("Verifying stack encapsulation by trying to modify the state array : ");
        stack.push(42);
        Object[] state = stack.getCurrentState();
        state[0] = 33;
        if((int)(stack.getIterator().next()) == 33){
            fail();
            return;
        }
        success();
    }

    /**
     * Tests if the stack can be entirely emptied
     */
    private static void stackCanBeEmptied(){
        System.out.print("Emptying the stack entirely : ");
        while(stack.getIterator().hasNext()){
            stack.pop();
```

```
        }
        if(!stack.toString().equals("[ ]")){
            fail();
            return;
        }
        success();
    }

    /**
     * Tests if popping an item from an empty stack throws an exception
     */
    private static void poppingAnItemFromEmptyStackMustGenerateException(){
        System.out.print("Trying to pop an item from an empty stack : ");
        try {
            Stack stack = new Stack();
            stack.pop();
        }
        catch (RuntimeException e){
            success(e);
            return;
        }
        fail();
    }

    /**
     * Tests if calling next() on an iterator referencing an empty stack throws an exception
     */
    private static void nextItemOnEmptyStackMustGenerateException(){
        System.out.print("Trying to reach next item with an iterator on an empty stack : ");
        try {
            Stack stack = new Stack();
            stack.getIterator().next();
        }
        catch (RuntimeException e){
            success(e);
            return;
        }
        fail();
    }

    /**
     * Tests if launching the Hanoi program with multiple argument throws an exception
     */
    private static void multipleArgumentsMustThrowException(){
        System.out.print("Trying to launch the hanoi program with multiple arguments : ");
        try {
            HanoiDisplayer.main(new String[]{"1", "2"});
        }
        catch (RuntimeException e){
            success(e);
            return;
        }
        fail();
    }

    /**
     * Tests if launching the Hanoi program with a non-integer argument throws an exception
     */
    private static void nonIntegerArgumentsMustThrowException(){
        System.out.print("Trying to launch the hanoi program with non integer arguments : ");
        try {
            HanoiDisplayer.main(new String[]{"test"});
        }
        catch (RuntimeException e){
            success(e);
            return;
        }
        fail();
    }

    /**
     * Tests if launching the Hanoi program with a negative integer argument throws an exception
```

```
     */
    private static void negativeValueArgumentsMustThrowException(){
        System.out.print("Trying to launch the hanoi program with negative integer arguments : ");
        try {
            HanoiDisplayer.main(new String[]{"-1"});
        }
        catch (RuntimeException e){
            success(e);
            return;
        }
        fail();
    }

    /**
     * Tests if the problem is solved with the correct number of turns
     */
    private static void hanoiAlgorithmMustBeSolvedWithCorrectTurnNumber(){
        for(int i = 1; i < 5; i++){
            Hanoi hanoi = new Hanoi(i);
            hanoi.solve();
            if(hanoi.turn() != (int)Math.pow(2, i) - 1){
                fail();
                return;
            }
        }
        success();
    }

    /**
     * Abstract class used to construct objects used to test the stack
     */
    abstract static class Pet {
        /**
         * The name of the pet
         */
        private final String name;

        /**
         * Construct a pet with a name
         * @param name The name of the pet
         */
        Pet(String name) {
            this.name = name;
        }

        /**
         * String representation of the pet
         * @return The object in String format
         */
        @Override
        public String toString() {
            return "Name : " + name;
        }
    }

    /**
     * Abstract class used to construct objects used to test the stack
     */
    static class Dog extends Pet {
        /**
         * Age of the dog
         */
        private final int age;

        /**
         * Construct a dog with a name and an age
         * @param name The name of the dog
         * @param age The age of the dog (in human's year)
         */
        Dog(String name, int age) {
            super(name);
            this.age = age;
```

```
        }

        /**
         * String representation of the dog
         * @return The object in String format
         */
        @Override
        public String toString() {
            return super.toString() + ", Age : " + age;
        }
    }
}
```