Lab5

March 20, 2025

1 Lab 5

Deadline: **Week 8** in your respective lab session

1.0.1 Name: Kiril Radev

1.0.2 Student ID: 241005831

You should only use things we learned up to this week (week 5), i.e. ArrayLists, HashMaps, Hashtables, etc., are not allowed. If you are unfamiliar with them, you are not expected to at this stage. Some of them will be introduced in future weeks.

1.1 Question 1 [1 mark]

Write an interface `SquareDrawer` that contains two methods with return types `void`, `drawSquare` and `drawSpecialSquare`.

Write an interface `StairCaseDrawer` which contains one method with return type `void drawStairCase`.

Write a class `Drawer` with three class variables `sideLength`, `filler`, and `special` of type int, char, and char, respectively. This class should also contain a constructor which takes three parameters and sets fields to these values.

Test your code!

Write your answer below:

-#1 interface SquareDrawer-

[1]:
```
public interface SquareDrawer {
        public void drawSquare();
        public void drawSpecialSquare();
}
```

-#2 interface StairCaseDrawer-

[2]:
```
public interface StairCaseDrawer {
        public void drawStairCase();
```

```
}
```

-#3 class Drawer-

[3]:
```
public class Drawer {
        int sideLength;
        char filler;
        char special;

        //constructor
        Drawer (int new_sideLength, char new_filler, char new_special) {
                this.sideLength = new_sideLength;
                this.filler = new_filler;
                this.special = new_special;
        }
}
```

1.2 Question 2 [1 mark]

Write a class `ShapeDrawer1`, a subclass of a Drawer, and implement the interface `SquareDrawer`.

This class should make use of its superclass's constructor.

`drawSquare` should print a square out of character stored inside `filler` with a side specified by `sideLength`.

For example `drawSquare` where `sideLength = 5` and `filler = '#'` should print out the following:

```
#####
#####
#####
#####
#####
```

`drawSpecialSquare` should print a square like `drawSquare`, but now diagonals should be made out of characters stored inside `special`.

For example `drawSpecialSquare` where `sideLength = 5`, `filler = '#'` and `special = 'X'` should print out the following:

```
X###X
#X#X#
##X##
#X#X#
X###X
```

You can assume the `sideLength` is always odd.

Remember to test your code!

Write your answer below:

-#4 class ShapeDrawer1-

[4]:
```
public class ShapeDrawer1 extends Drawer implements SquareDrawer
{
    //constructor
        ShapeDrawer1 (int new_sideLength, char new_filler, char new_special)
        {
                super(new_sideLength,new_filler,new_special);
        }

        public void drawSquare()
        {
                for(int i = 0; i < sideLength; i++)
                {
                        for(int j = 0; j < sideLength; j++)
                        {
                                System.out.print(filler);
                        }
                        System.out.println();
                }
        }//END drawSquare

        public void drawSpecialSquare()
        {
                int mid_row = sideLength / 2;

                upperRows(mid_row);
                middleRow(mid_row);
                lowerRows(mid_row);
        }//END drawSpecialSquare

        private void upperRows(int mid_row)
        {
                for(int i = 0; i < mid_row; i++)
                {
                        for(int left_filler = 0; left_filler < i; left_filler++)
                                System.out.print(filler);

                        System.out.print(special);

                        for(int mid_filler = i + 1; mid_filler < sideLength -␣
 ↪(i + 1); mid_filler++)
                                System.out.print(filler);

                        System.out.print(special);
```

```
                        for(int rigth_filler = sideLength - i; rigth_filler <␣
↪sideLength; rigth_filler++)
                                System.out.print(filler);

                        System.out.println();
                }
        }//END upperRows

        private void middleRow(int mid_row)
        {
                for(int i = 0; i < mid_row; i++)
                        System.out.print(filler);

                System.out.print(special);

                for(int i = 0; i < mid_row; i++)
                        System.out.print(filler);

                System.out.println();
        }//END middleRow

        private void lowerRows(int mid_row)
        {
                for(int i = mid_row; i > 0; i--)
                {
                        for(int left_filler = 0; left_filler < i - 1;␣
↪left_filler++)
                                System.out.print(filler);

                        System.out.print(special);

                        for(int mid_filler = i + 1; mid_filler < sideLength -␣
↪(i - 1); mid_filler++)
                                System.out.print(filler);

                        System.out.print(special);

                        for(int rigth_filler = sideLength - i; rigth_filler <␣
↪sideLength - 1; rigth_filler++)
                                System.out.print(filler);

                        System.out.println();
                }
        }//END lowerRows
}
```

Run your program:

[16]:
```
ShapeDrawer1 square = new ShapeDrawer1(5,'#','X');
//tester
System.out.println("1");
square.drawSquare();
System.out.println("2");
square.drawSpecialSquare();
```

```
1
#####
#####
#####
#####
#####
2
X###X
#X#X#
##X##
#X#X#
X###X
```

1.3 Question 3 [1 mark]

Copy the class from Question 2 and rename `ShapeDrawer1` to `ShapeDrawer2` where appropriate.

`ShapeDrawer2` should implement both `SquareDrawer` and `StaircaseDrawer`.

The `StairCase` drawer should print out the staircase out of `filler` with each step of size `sideLength` in both dimensions with a number of steps specified by `sideLength`.

For example where `sideLength = 5` and `filler = #` should print out the following:

```
#####
#####
#####
#####
#####
##########
##########
##########
##########
##########
###############
###############
###############
###############
###############
####################
####################
```

```
####################
####################
####################
########################
########################
########################
########################
########################
```

Test whether all methods in ShapeDrawer2 work as expected!

Write your answer below:

-#5 class ShapeDrawer2-

[5]:
```
public class ShapeDrawer2 extends Drawer implements StairCaseDrawer
{
    //constructor
        ShapeDrawer2 (int new_sideLength, char new_filler, char new_special)
        {
                super(new_sideLength,new_filler,new_special);
        }

        public void drawStairCase()
        {
                for(int stairs = 1; stairs <= sideLength; stairs++)
                {
                        for(int stair = 0; stair < sideLength; stair++)
                        {
                                for(int characters = 0; characters < stairs *␣
 ↪sideLength; characters++)
                                {
                                        System.out.print(filler);
                                }
                                System.out.println();
                        }
                }
        }//END drawStairCase
}
```

Run your program:

[17]:
```
ShapeDrawer2 stair = new ShapeDrawer2(5,'#','X');
//tester
stair.drawStairCase();
```

```
#####
#####
#####
#####
```

```
#####
##########
##########
##########
##########
##########
###############
###############
###############
###############
###############
###################
###################
###################
###################
###################
#########################
#########################
#########################
#########################
#########################
```

1.4 Question 4 [1 mark]

You are given three methods below, one for each `String`, `Integer`, and `Boolean`, respectively, that takes two objects and checks whether they store the same value.

Write one method that does the same check but works for all types.

You are expected to test your code thoroughly!

Hint: Use generics.

Write your answer below:

-#1 example methods areEqual-

[6]:
```java
public static boolean areEqual(String s1, String s2) {
    return s1.equals(s2);
}

public static boolean areEqual(Integer i1, Integer i2) {
    return i1.equals(i2);
}

public static boolean areEqual(Boolean b1, Boolean b2) {
    return b1.equals(b2);
}
```

-#2 method areEqual-

[7]:
```java
//<T> => of any type
public static <T> boolean areEqual(T obj1, T obj2) {
        return obj1.equals(obj2);
}
```

Run your program:

[3]:
```java
String a = "a";
String b = "b";
String c = "a";

int x = 1;
int y = 0;
int z = 1;

boolean t = true;
boolean f = false;
boolean tt = true;

//compare String values
System.out.println(areEqual(a,b));
System.out.println(areEqual(a,c));
//compare integer values
System.out.println(areEqual(x,y));
System.out.println(areEqual(x,z));
//compare boolean values
System.out.println(areEqual(t,f));
System.out.println(areEqual(t,tt));
```

```
false
true
false
true
false
true
```

1.5 Question 5 [1 mark]

Write a method `isReachable` which takes two `Node` arguments, one for the start node and one for the end node. The method returns `true` if and only if it is possible to go from the node `start` to the node `end` in the directed graph, following the arrows.

You are provided with a class `Node`, which represents a node within a graph, `label` corresponds to the "name" of the node, and the array `outgoing` is all the nodes that are connected to the current node by outgoing arrows.

The image below shows an example directed graph that we also coded below to make testing easier.

We wrote some tests for you, but you are expected to write a few more.

Write your answer below:

-#1 object Node-

[8]:
```
class Node {
    String label;
    Node[] outgoing;

    //constructor
    Node(String label)
    {
        this.label = label;
        this.outgoing = null;
    }

    public void linkTo(Node n)
    {
            //check if no nodes have been connected
        if(outgoing == null)
        {
            outgoing = new Node[1];
            outgoing[0] = n;
            return;
        }
        //check if the node we want to connect has been already connected
        for(Node x : outgoing)
        {
                if(x.label.equals(n.label))
                        return;
        }
        //add the new node if the exceptions do not hold
        Node[] newOutgoing = new Node[outgoing.length+1];
        for(int i = 0; i < outgoing.length; i++)
        {
            newOutgoing[i] = outgoing[i];
        }
        newOutgoing[outgoing.length] = n;
        outgoing = newOutgoing;
    }//END linkTo
}
```

-#2 method isReahcable-

[13]:
```
public static boolean isReachable(Node start, Node end)
{
```

```
        //--1 check if start & end are the same node
        if(start.label.equals(end.label))
                return true;
        //--2 check if start is connected to other nodes
        else if(start.outgoing == null)
                return false;

        //--3 create storage for currently encountered nodes
        //    and ones that are soon to be encountered
        Node[] current_nodes = new Node[] {start};
        Node[] future_nodes = new Node[] {};
        int counted_nodes = 0;

        //--4 because current_nodes & future_nodes will be the sentinel values␣
↪of a while loop
        //    we need to create a base case check before entering it, so␣
↪future_nodes could be
        //    could be passed new values and not be empty

        //--5 check current_nodes
    //    this one was included because the while loop will begin from 0 again
    //    since future_nodes is empty meaning after the equallity future_nodes␣
↪= current_nodes
    //    we will get that current_nodes is empty again and we have to do the␣
↪base case again
        for(Node current_node : current_nodes)
        {
            //--6 check the incoming 'future' nodes
                for(Node outer_node : current_node.outgoing)
                {
                        //--7 check if one of the future nodes = end node
                        if(outer_node.label.equals(end.label))
                                return true;
                        //--8 if not then add the future node into the array
                        future_nodes = updateEncounteredNodes(current_nodes,␣
↪outer_node);
                }
        }

        //--9 create a while loop that check if current_nodes.length !=␣
↪future_nodes.length,
        //    that is because the method updateEncounteredNodes() has an 'if␣
↪statement'
        //    that allows it to add only nodes that haven`t been encountered,
        //    thus every node is unique and if the lengths are equal,
        //    then the node is unreachable and we have entered a cycling loop
```

```
        while(current_nodes.length != future_nodes.length)
        {
                //--10 repeat the same procedure with the exception that
                //     current_nodes has to take on the value of future_nodes
                //     and we create a counter to omit repetitive comparisons
                counted_nodes = current_nodes.length;
                current_nodes = future_nodes;
                for(int i = counted_nodes; i < current_nodes.length; i++)
                {
                        for(Node outer_node : current_nodes[i].outgoing)
                        {
                                if(outer_node.label.equals(end.label))
                                        return true;

                                future_nodes =␣
 ↪updateEncounteredNodes(current_nodes, outer_node);
                        }
                }
        }

        return false;
}//END isReachable
```

-*#3 extra method updateEncounteredNodes*-

[10]:
```
public static Node[] updateEncounteredNodes(Node[] checker, Node new_node)
{
    //check if the node we want to connect has been already connected
    for(Node x : checker)
        if(x.label.equals(new_node.label))
                return checker;

    //create a storage Node array of length +1
    Node[] new_checker = new Node[checker.length+1];
    for(int i = 0; i < checker.length; i++)
        new_checker[i] = checker[i];
    //store the new node and return
    new_checker[checker.length] = new_node;
    return new_checker;
}//END updateEncounteredNodes
```

Run your program:

[15]:
```
Node a = new Node("A");
Node b = new Node("B");
Node c = new Node("C");
Node d = new Node("D");
```

```
Node e = new Node("E");
Node f = new Node("F");

a.linkTo(b);
b.linkTo(c);
c.linkTo(e);
e.linkTo(f);
e.linkTo(d);
d.linkTo(b);

System.out.println(isReachable(a, e)); // true
System.out.println(isReachable(f, a)); // false

// MORE TESTS HERE
```

```
true
false
```

[14]:
```
//nodes creation
Node a = new Node("A");             // node reachable only by itself
Node b = new Node("B");             // cyclic node
Node c = new Node("C");             // cyclic node
Node d = new Node("D");             // cyclic node
Node e = new Node("E");             // cyclic node
Node f = new Node("F");             // dead end situation node
Node epsilon = new Node("Epsilon"); // node not connected to any other

//linking nodes
a.linkTo(b);
b.linkTo(c);
c.linkTo(e);
e.linkTo(f);
e.linkTo(d);
d.linkTo(b);

//testing
Node[] nodes = new Node[] {a,b,c,d,e,f,epsilon};
for(Node n : nodes)
{
        for(Node m : nodes)
                System.out.print(n.label + "->" + m.label + ":" +␣
 ↪isReachable(n,m) + " ");

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

```
A->A:true A->B:true A->C:true A->D:true A->E:true A->F:true A->Epsilon:false
```

```
B->A:false B->B:true B->C:true B->D:true B->E:true B->F:true B->Epsilon:false
C->A:false C->B:true C->C:true C->D:true C->E:true C->F:true C->Epsilon:false
D->A:false D->B:true D->C:true D->D:true D->E:true D->F:true D->Epsilon:false
E->A:false E->B:true E->C:true E->D:true E->E:true E->F:true E->Epsilon:false
F->A:false F->B:false F->C:false F->D:false F->E:false F->F:true
F->Epsilon:false
Epsilon->A:false Epsilon->B:false Epsilon->C:false Epsilon->D:false
Epsilon->E:false Epsilon->F:false Epsilon->Epsilon:true
```

[ ]: