package linkedlist;

public interface IList<T> {

}

package linkedlist;

/\*\*

\* A quick demo of how to implement a set using a singly-linked list

\* @author trao

\*

\*/

public class LinkedSet<T> {

//---------------------------------------------------------------------------------------------------------------

//The body of the list is held in a singly-linked list

private SLList<T> body;

//----------------------------------------------------------------------------------------------------------------

// Constructor creates a new singly-linked list for the body

public LinkedSet()

{

body = new SLList<T>();

}

//----------------------------------------------------------------------------------------------------------------

// member method simply delegates the membership testing to SLList

public boolean member(T elm)

{

return body.member(elm);

}

//----------------------------------------------------------------------------------------------------------------

// Inserts a new element in to the set, if it not already there

public void insert(T str)

{

// Write your code here:

// if str is not a member of this, insert str into this

if (!body.member(str)) {

body.insertAt(str, body.size());

}

}

//----------------------------------------------------------------------------------------------------------------

// Simple printing method, delegates the job to SLList

public void printSet()

{

System.out.println (body.toString());

}

//---------------------------------------------------------------------------------------------------------------- // builds a set by inserting all elements of an array

public void buildSet(T[] elements)

{

// Write your code here

//Insert all elements one by one

for(int i=0; i<elements.length; i++) {

this.body.insertAt(elements[i], i);

}

}

//------------------------------------------------------------------------------------------------------

// Returns the union of this set and the other set without

// modifying this set or the other set

public LinkedSet<T> union(LinkedSet<T> otherSet)

{

LinkedSet<T> result = new LinkedSet<T>();

SLList<T> resultBody = result.body;

SLList<T> otherBody = otherSet.body;

for(int i=0; i<this.body.size(); i++) {

if(!resultBody.member(this.body.elementAt(i))) {

resultBody.insertAt(this.body.elementAt(i), 0);

}

}

for(int i=0; i<otherBody.size(); i++) {

if(!resultBody.member(otherBody.elementAt(i))) {

resultBody.insertAt(otherBody.elementAt(i), 0);

}

}

for(int i=1;i<resultBody.size();i++) {

result.insert(resultBody.elementAt(i));

}

return result;

}

//-------------------------------------------------------------------------------------------------------

// Returns the intersection of this set and the other set

// without modifying this set or the other set

public LinkedSet<T> intersection(LinkedSet<T> otherSet)

{

LinkedSet<T> result = new LinkedSet<T>();

SLList<T> resultBody = result.body;

SLList<T> otherBody = otherSet.body;

for(int i=0; i<this.body.size(); i++) {

for(int j=0; j<otherBody.size(); j++) {

if((this.member(otherBody.elementAt(i))) &&

(!resultBody.member(otherBody.elementAt(i)))) {

resultBody.insertAt(otherBody.elementAt(i), 0);

}

}

}

for(int i=0; i<resultBody.length(); i++) {

result.insert(resultBody.elementAt(i));

}

return result;

}

//-------------------------------------------------------------------------------------------------------

// Returns the difference of this set and the other set

// i.e. thisSet – otherSet (All the elements that are in

// in this set, but not in the other set

// without modifying this set or the other set

public LinkedSet<T> difference(LinkedSet<T> otherSet)

{

LinkedSet<T> result = new LinkedSet<T>();

SLList<T> resultBody = result.body;

SLList<T> otherBody = otherSet.body;

for(int i=0; i<this.body.size(); i++) {

if(!otherBody.member(this.body.elementAt(i))) {

resultBody.insertAt(this.body.elementAt(i), 0);

}

}

for(int i=0; i< resultBody.length(); i++) {

result.insert(resultBody.elementAt(i));

}

return result;

}

}

package linkedlist;

public class SLList<T> implements IList<T> {

//-------------------------------------------------------------------------------------------------------

//List is represented by a reference to

private SLNode<T> head;

private SLNode<T> thisElement = null;

public SLNode<T> getHead()

{

return head;

}

//-------------------------------------------------------------------------------------------------------

public boolean isEmpty()

{

return (head == null);

}

//------------------------------------------------------------------------------------------------------- public void resetThisElement()

{

thisElement = head;

}

//------------------------------------------------------------------------------------------------------ public boolean hasMoreElements()

{

return (thisElement != null);

}

public T nextElement()

{

T thisElement = thisElement.getInfo();

thisElement = thisElement.getNext();

return thisElement;

}

//-------------------------------------------------------------------------------------------------------

public int length()

{

int len = 0;

SLNode<T> thisNode = head;

while (thisNode != null)

{

len++;

thisNode = thisNode.getNext();

}

return len;

}

//-------------------------------------------------------------------------------------------------------

public T elementAt(int loc)

{

int ct = 0;

SLNode<T> thisNode = head;

while (thisNode != null && ct < loc)

{

thisNode = thisNode.getNext();

ct++;

}

if (thisNode != null)

return thisNode.getInfo();

return null;

}

public T first()

{

if (this.isEmpty())

return null;

return elementAt(0);

}

public SLList rest()

{

if (this.isEmpty())

return null;

SLList rest = new SLList();

rest.head = this.head.getNext();

return rest;

}

/\*\*

\* Insert the new info at the location loc

\* @param info

\* @param loc

\* @return

\*/

//-------------------------------------------------------------------------------------------------------

public void insertAt(T info, int loc)

{

//--------------------------------------------------------------------------------------------

// If loc is illegal, return false

if (loc < 0 || loc > size())

return;

//-------------------------------------------------------------------------------------------- // Create a new node for info

SLNode<T> node = new SLNode<T>(info);

//--------------------------------------------------------------------------------------------

// Special case: insert at the beginning

if (loc == 0)

{

node.setNext(head);

head = node;

return;

}

//--------------------------------------------------------------------------------------------

//Find a pointer to the node before the insertion point

SLNode<T> pointer = pointerTo(loc-1);

//-------------------------------------------------------------------------------------------- //Next of the new node should go to the next of the pointer

node.setNext(pointer.getNext());

//--------------------------------------------------------------------------------------------

//Next of pointer should go to new node

pointer.setNext(node);

//--------------------------------------------------------------------------------------------

}

//-----------------------------------------------------------------------------------

/\*\*

\* Delete element at a location loc

\*/

public void deleteNth(int loc)

{

//--------------------------------------------------------------------------------------------

// If loc is illegal, return false

if (loc < 0 || loc > size())

return;

//--------------------------------------------------------------------------------------------

// Special case: delete at the beginning

if (loc == 0)

{

head = head.getNext();

return;

}

//--------------------------------------------------------------------------------------------

//Find a pointer to the node before the insertion point

SLNode<T> pointer = pointerTo(loc-1);

//--------------------------------------------------------------------------------------------

//Make it point to the node after the node to be deleted

pointer.setNext(pointer.getNext().getNext());

}

public boolean member(T info)

{

SLNode<T> thisNode = head;

while (thisNode != null)

{

if (thisNode.getInfo().equals(info))

return true;

thisNode = thisNode.getNext();

}

return false;

}

//-------------------------------------------------------------------------------------------------------

/\*\*

\* Return a pointer to the node whose index is n

\*/

public SLNode<T> pointerTo(int n)

{

if (n < 0 || n > size())

return null;

SLNode<T> pointer = head;

int counter = 0;

while (counter < n)

{

counter++;

pointer = pointer.getNext();

}

return pointer;

}

//-----------------------------------------------------------------------------------

/\*\*

\* Find the length of the list

\*/

public int size()

{

return length();

}

public void deleteFirstOccurrence(T info)

{

SLNode<T> thisNode = head;

SLNode<T> prevNode = null;

while (thisNode != null)

{

if (thisNode.getInfo().equals(info))

{

if (thisNode == head)

head = thisNode.getNext();

else

prevNode.setNext(thisNode.getNext());

return;

}

prevNode = thisNode;

thisNode = thisNode.getNext();

}

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* Delete the first occurrence of this info in a list

\* If this info doesn't occur then nothing is done

\*/

public void deleteAllOccurrences(T info)

{

SLNode<T> thisNode = head;

SLNode<T> prevNode = null;

while (thisNode != null)

{

if (thisNode.getInfo().equals(info))

{

if (thisNode == head)

head = thisNode.getNext();

else

prevNode.setNext(thisNode.getNext());

}

prevNode = thisNode;

thisNode = thisNode.getNext();

}

}

//-----------------------------------------------------------------------------------

/\*\*

\* Create a printable string from the data in this list

\*/

public String toString()

{

String result = "[ ";

SLNode<T> pointer = head;

while (pointer != null)

{

result = result + pointer.getInfo().toString()+", ";

pointer = pointer.getNext();

}

result = result + " ]";

return result;

}

/\*\*

\* Concatenating s2 at the end of this list

\*/

public void concat(SLList<T> s2)

{

// Get a pointer to the last element

SLNode<T> pointer = this.pointerTo(this.length() - 1);

pointer.setNext(s2.getHead());

}

}

package linkedlist;

/\*\*

\* Node Structure in a Singly-linked list

\* @author trao

\*

\*/

public class SLNode <T> {

private T info;

private SLNode<T> next;

//----------------------------------------------------------------------------------------------

public SLNode (T inf)

{

info = inf;

next = null;

}

//----------------------------------------------------------------------------------------------

public T getInfo() {

return info;

}

//----------------------------------------------------------------------------------------------

public void setInfo(T info) {

this.info = info;

}

//----------------------------------------------------------------------------------------------

public SLNode<T> getNext() {

return next;

}

//----------------------------------------------------------------------------------------------

public void setNext(SLNode<T> next) {

this.next = next;

}

}

package linkedlist;

/\*\*

\* DoublyLinkedList: A doubly-linked implementation. An abstract class.

\* Implements the various insert/delete methods on a list of abstract nodes

\* We assume a doubly-linked node.

\* The implementation keeps pointers to the front and rear of the list and

\* maintains its own length.

\*

\* @author (T.M. Rao)

\*

\*/

import java.util.\*;

public class DoublyLinkedList <T> implements Enumeration<T>, IList<T>

{

//Instance Variables

//------------------------------------------------------------------------------------------------------------

protected DoublyLinkedNode<T> firstNode;

//------------------------------------------------------------------------------------------------------------

protected DoublyLinkedNode<T> lastNode;

//------------------------------------------------------------------------------------------------------------

private DoublyLinkedNode<T> thisNode;

//------------------------------------------------------------------------------------------------------------

//Constructor

public DoublyLinkedList()

{

firstNode = null;

thisNode = null;

lastNode = null;

}

//------------------------------------------------------------------------------------------------------------ /\*\*

\* Checks if the list is empty.

\*/

public boolean isEmpty()

{

return (firstNode == null && lastNode == null);

}

//------------------------------------------------------------------------------------------------------------ public DoublyLinkedNode<T> getFirst()

{

return firstNode;

}

//------------------------------------------------------------------------------------------------------------

public DoublyLinkedNode<T> getLast()

{

return lastNode;

}

//------------------------------------------------------------------------------------------------------------ public DoublyLinkedNode<T> getCurrent()

{

return thisNode;

}

//------------------------------------------------------------------------------------------------------------ /\*\*

\* Retriever method: returns length

\*/

public int length()

{

//return length; This is a O(N) operation

int len = 0;

DoublyLinkedNode<T> dln = firstNode;

while (dln != null)

{

len++;

dln = dln.getNextNode();

}

return len;

}

//------------------------------------------------------------------------------------------------------------ /\*\*

\* Resets the thisNode cursor to the beginning of the

\* list. This should be done before we scan the list for

\* any reason

\*/

public void resetThisElement()

{

thisNode = firstNode;

}

//------------------------------------------------------------------------------------------------------------

public void resetThisElementToLast()

{

thisNode = lastNode;

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* Creates a brand new copy of this list in completely different

\* memory area.

\*/

DoublyLinkedList<T> deepCopy()

{

DoublyLinkedList<T> copy = new DoublyLinkedList<T>();

DoublyLinkedNode<T> dln = firstNode;

while (dln != null)

{

copy.insertAtEnd(dln.getInfo());

dln = dln.getNextNode();

}

return copy;

}

//------------------------------------------------------------------------------------------------------------ /\*\*

\* Determines if this list is actually a set (i.e. it has no repeated

\* elements

\*/

public boolean isSet()

{

DoublyLinkedNode<T> dln1 = firstNode;

if (isEmpty()) return true;

if (firstNode == lastNode) return true;

DoublyLinkedNode<T> lastButOneNode = lastNode.getPrevNode();

while (dln1 != lastButOneNode)

{

DoublyLinkedNode<T> dln2 = dln1.getNextNode();

while (dln1 != lastNode)

{

if (dln1.getInfo().equals(dln2.getInfo()))

return false;

dln2 = dln2.getNextNode();

}

dln1 = dln1.getNextNode();

}

return false;

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* hasMoreElements() and nextElement() methods provide a

\* way of scanning the elements of a list from first node

\* to last node.

\* Typical usage:

\* resetThisElement();

\* while (hasMoreElements())

\* {

\* nextEle = nextElement();

\* ... process the nextEle ...

\* }

\*/

public boolean hasMoreElements()

{

return (thisNode != null);

}

//------------------------------------------------------------------------------------------------------------ public T nextElement()

{

T info = thisNode.getInfo();

thisNode = thisNode.getNextNode();

return info;

}

//------------------------------------------------------------------------------------------------------------

public T prevElement()

{

T info = thisNode.getInfo();

thisNode = thisNode.getPrevNode();

return info;

}

//------------------------------------------------------------------------------------------------------------

public T getThisElement()

{

T info = thisNode.getInfo();

return info;

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* Returns a reference to the n-th node of the list

\*/

public DoublyLinkedNode<T> nodeAt(int index)

{

if (index < 0 || index >= length())

{

//First filter out the illegal values

System.out.println("Error in elementAt"+

"index is negative or too large; null returned");

return null;

}

//If the list is empty, failure

if (isEmpty())

{

System.out.println("Error in elementAt"+

"List is empty; null returned");

return null;

}

//First element

else if (index == 0)

{

return firstNode;

}

//Last element

else if (index == length()-1)

{

return lastNode;

}

else

//Somewhere in the middle

{

DoublyLinkedNode<T> node = firstNode;

//count up to index-1 starting at 0

for (int count = 0; count < index; count++)

{

node = node.getNextNode();

}

return node;

}

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* Returns a reference to the n-th element of the list

\*/

public T elementAt(int index)

{

DoublyLinkedNode<T> node = nodeAt(index);

return node.getInfo();

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* Special case: insertion at the beginning

\* (like push in a stack)

\*/

public void insertAtBeginning(DoublyLinkedNode<T> node)

{

node.setNextNode(firstNode);

node.setPrevNode(null);

if (isEmpty())

{

lastNode = node;

}

else

{

firstNode.setPrevNode(node);

}

firstNode = node;

}

//----------------------------------------------------------------------------------------------------------------

//Anothe Version (with just Info)

public void insertAtBeginning(T info)

{

insertAtBeginning(new DoublyLinkedNode<T>(null, info, null));

}

//----------------------------------------------------------------------------------------------------------------

/\*\*

\* Special case: insertion at the end

\* (like insert in a queue)

\*/

public void insertAtEnd(DoublyLinkedNode<T> node)

{

node.setPrevNode(lastNode);

node.setNextNode(null);

if (isEmpty())

{

firstNode = node;

}

else

{

lastNode.setNextNode(node);

}

lastNode = node;

}

//----------------------------------------------------------------------------------------------------------------

//Another version

public void insertAtEnd(T info)

{

insertAtEnd(new DoublyLinkedNode<T>(null, info, null));

}

//--------------------------------------------------------------------------------------------------------------

/\*\*

\* General case: insertion at the n-th location

\*

\*/

public void insertAt(DoublyLinkedNode<T> node, int location)

{

//Check for illegal values of index

if ((location < 0)||(location > length()))

{

System.out.println("Error in insertAt"+

" location is too large or too small; No Action taken");

}

//Spacial Case: Insert At beginning

else if (location == 0)

{

insertAtBeginning(node);

}

//Spacial Case: Insert At end

else if (location == length())

{

insertAtEnd(node);

}

else

{

//Get a pointer to n-th element

DoublyLinkedNode<T> thisNode = nodeAt(location);

//Adjust the pointers correctly to effect the insertion

DoublyLinkedNode<T> prevNode = thisNode.getPrevNode();

node.setNextNode(thisNode);

node.setPrevNode(prevNode);

prevNode.setNextNode(node);

thisNode.setPrevNode(node);

}

}

//------------------------------------------------------------------------------------------------------------

//Another Version

public void insertAt(T info, int location)

{

insertAt(new DoublyLinkedNode<T>(null, info, null), location);

}

//-------------------------------------------------------------------------------------------------------------

/\*\*

\* Delete the first element of the list

\*/

public void deleteFirst()

{

//Can't delete from an empty list

if (isEmpty())

{

System.out.println("Error in AbstractList.deleteFirst() "+

"Attempted delete from Empty List ");

}

//Special case: list has only one element

else if (length()==1)

{

firstNode = null;

lastNode = null;

}

//All othe lists: Only first pointer changes. Last doesn't

else

{

DoublyLinkedNode<T> nextNode = firstNode.getNextNode();

nextNode.setPrevNode(null);

firstNode = nextNode;

}

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* Delete last element of a list

\*/

public void deleteLast()

{

//Can't delete from an empty list

if (isEmpty())

{

System.out.println("Error in AbstractList.deleteLast() "+

"Attempted delete from Empty List ");

}

//Special case: list has only one element

else if (length()==1)

{

firstNode = null;

lastNode = null;

//length = 0;

}

//All othe lists: Only last pointer changes. First doesn't

else

{

DoublyLinkedNode<T> prevNode = lastNode.getPrevNode();

prevNode.setNextNode(null);

lastNode = prevNode;

}

}

//----------------------------------------------------------------------------------------------------------------

/\*\*

\* Delete the n-th element of a list

\*/

public void deleteNth(int location)

{

//Check for illegal values of index

if ((location < 0)||(location >= length()))

{

System.out.println("Error in deleteNth"+

" location is too large or too small; No Action taken");

}

//Spacial Case: Delete At beginning

else if (location == 0)

{

deleteFirst();

}

//Spacial Case: Delete At end

else if (location == length()-1)

{

deleteLast();

}

else

{

//Get a pointer to the n-th element

DoublyLinkedNode<T> thisNode = nodeAt(location);

//Get pointers to its prev and next nodes

DoublyLinkedNode<T> prevNode = thisNode.getPrevNode();

DoublyLinkedNode<T> nextNode = thisNode.getNextNode();

//Adjust pointers

prevNode.setNextNode(nextNode);

nextNode.setPrevNode(prevNode);

}

}

//--------------------------------------------------------------------------------------------------------------

/\*\*

\* Membership in a list: Is this object present in the list?

\*/

public boolean member(T info)

{

DoublyLinkedNode<T> thisNode = firstNode;

while (thisNode != null)

{

T nodeInfo = thisNode.getInfo();

if (nodeInfo.equals(info))

return true;

thisNode = thisNode.getNextNode();

}

return false;

}

//--------------------------------------------------------------------------------------------------------------

/\*\*

\* Delete the first occurrence of this info in a list

\* If this info doesn't occur then nothing is done

\*/

public void deleteFirstOccurrence(T info)

{

DoublyLinkedNode<T> thisNode = firstNode;

int location = -1;

while (thisNode != null)

{

location++;

Object nodeInfo = thisNode.getInfo();

if (nodeInfo.equals(info))

{

if (location == 0)

deleteFirst();

else if (location == length()-1)

deleteLast();

else

{

//Get pointers to its prev and next nodes

DoublyLinkedNode<T> prevNode = thisNode.getPrevNode();

DoublyLinkedNode<T> nextNode = thisNode.getNextNode();

//Adjust pointers

prevNode.setNextNode(nextNode);

nextNode.setPrevNode(prevNode);

}

break;

}

//Increment the pointer to next node

thisNode = thisNode.getNextNode();

}

}

//----------------------------------------------------------------------------------------------------------------

/\*\*

\* Deletes the first occurrence of this info in a list

\* If this info doesn't occur then nothing is done

\*/

public void deleteAllOccurrences(T info)

{

DoublyLinkedNode<T> thisNode = firstNode;

while (thisNode != null)

{

T nodeInfo = thisNode.getInfo();

if (nodeInfo.equals(info))

{

//Get pointers to its prev and next nodes

DoublyLinkedNode<T> prevNode = thisNode.getPrevNode();

DoublyLinkedNode<T> nextNode = thisNode.getNextNode();

//Adjust pointers

if (prevNode == null && nextNode == null)

{

//There is only one element in the list

deleteFirst();

return;

}

if (prevNode != null)

prevNode.setNextNode(nextNode);

else

{

firstNode = nextNode;

nextNode.setPrevNode(null);

}

if (nextNode != null)

nextNode.setPrevNode(prevNode);

else

{

lastNode = prevNode;

prevNode.setNextNode(null);

}

}

thisNode = thisNode.getNextNode();

}

}

//------------------------------------------------------------

/\*\*

\* Convert list contents into a printable String

\*/

public String toString()

{

String result = "[ ";

DoublyLinkedNode<T> nextNode = firstNode;

while (nextNode != null)

{

result = result + nextNode.toString() + " ";

nextNode = nextNode.getNextNode();

}

result = result+" ]";

return result;

}

//------------------------------------------------------------------------------------------------------------

/\*\*

\* Reverse the list using only a constant amount of extra memory

\*/

public void reverseInPlace()

{

DoublyLinkedNode<T> node = firstNode;

while (node != null)

{

DoublyLinkedNode<T> prev = node.getPrevNode();

DoublyLinkedNode<T> next = node.getNextNode();

node.setPrevNode(next);

node.setNextNode(prev);

node = next;

}

node = firstNode;

firstNode = lastNode;

lastNode = node;

}

}

package linkedlist;

/\*\*

\* class DoublyLinkedNode: A doubly-linked node implementation.

\* This represents a node in a doubly linked list. We have a

\* forward pointer and a back pointer

\*

\* @author (T.M. Rao)

\*

\*/

public class DoublyLinkedNode <T>

{

//DoublyLinkedNode structure

//----------------------------------------------------------------------------------------------------------------

private DoublyLinkedNode<T> nextNode;

//----------------------------------------------------------------------------------------------------------------

private DoublyLinkedNode<T> prevNode;

//----------------------------------------------------------------------------------------------------------------

protected T info;

/\*\*

\* A constructor

\*/

public DoublyLinkedNode(T inf)

{

prevNode = null;

info = inf;

nextNode = null;

}

/\*\*

\* Another constructor

\*/

public DoublyLinkedNode (DoublyLinkedNode<T> prev, T inf,

DoublyLinkedNode<T> next)

{

prevNode = prev;

info = inf;

nextNode = next;

}

//Retriever methods------------------------------------------------------------------------------------------

public DoublyLinkedNode <T> getNextNode()

{

return nextNode;

}

public DoublyLinkedNode <T> getPrevNode()

{

return prevNode;

}

public T getInfo()

{

return info;

}

//---------------------------------------------------------------------------------------------------------------

//Mutator methods-------------------------------------------------------------------------------------------

public void setNextNode(DoublyLinkedNode<T> node)

{

nextNode = node;

}

public void setPrevNode(DoublyLinkedNode<T> node)

{

prevNode = node;

}

public void setInfo(T inf)

{

info = inf;

}

//----------------------------------------------------------------------------------------------------------------

public String toString()

{

return info.toString();

}

}

package linkedlist;

import java.io.File;

import java.io.FilenameFilter;

public class MyFilter implements FilenameFilter

{

public boolean accept(File dir, String name)

{

name = name.toUpperCase();

if (name.endsWith(".JPG") || name.endsWith(".JPEG"))

{

return true;

}

if (name.endsWith(".PNG"))

{

return true;

}

if (name.endsWith(".TIF"))

{

return true;

}

return false;

}

}

package linkedlist;

public class LinkedSetTester {

public static void main(String[] args) {

String[] s1 = {"abc", "defg", "abc", "ijkl"};

// As an array, repeated elements are OK

LinkedSet<String> set1 = new LinkedSet<String>();

set1.buildSet(s1); // but no repeated elements in Set

System.out.println("Set1");

System.out.println("---------------");

set1.printSet();

System.out.println("---------------");

String[] s2 = {"xyz", "defg", "abc", "pqr", "xyz"};

LinkedSet<String> set2 = new LinkedSet<String>();

set2.buildSet(s2);

System.out.println("Set2");

System.out.println("---------------");

set2.printSet();

System.out.println("---------------");

System.out.println("Set1 union Set2");

System.out.println("---------------");

set1.union(set2).printSet();

System.out.println("---------------");

System.out.println("Set1 intersection Set2");

System.out.println("---------------");

set1.intersection(set2).printSet();

System.out.println("---------------");

System.out.println("Set1 - Set2");

System.out.println("---------------");

set1.difference(set2).printSet();

System.out.println("---------------");

}

}

---------------------------------------------------------------------------------------------------------------------

package JavaFXGUIDemo;

//import statements----------------------------------------------------------------------------------------------

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Label;

import javafx.scene.control.Button;

import javafx.scene.control.ComboBox;

import javafx.scene.image.Image;

import javafx.scene.image.ImageView;

import javafx.scene.layout.BorderPane;

import javafx.scene.layout.GridPane;

import javafx.stage.Stage;

import linkedlist.\*;

import javafx.event.EventHandler;

import javafx.geometry.Pos;

import javafx.event.ActionEvent;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

//-------------------------------------------------------------------------------------------------------------------

import javax.swing.JOptionPane;

//-------------------------------------------------------------------------------------------------------------------

public class BorderPaneDemo extends Application {

private BorderPane bPane;

private Label picLabel;

private ComboBox<String> directoryNamesBox;

private Scene scene;

private GridPane navPanel;

private DoublyLinkedList<String> myList;

private int currentIndex;

private Label showCountLabel;

private Button begin;

private Button end;

private Button forward;

private Button backword;

//-----------------------------------------------------------------------------------------------------------------

public void createGUIComponents()

{

// Instantiates GUI Components--------------------------------------------------------------------------

picLabel = new Label("");

showCountLabel = new Label("Count");

directoryNamesBox = new ComboBox<String>();

begin = new Button("<<");

backword = new Button ("<");

forward = new Button (">");

end = new Button (">>");

navPanel = new GridPane ();

navPanel.add(begin, 0, 0);

navPanel.add(backword, 1, 0);

navPanel.add(showCountLabel, 2, 0);

navPanel.add(forward, 3, 0);

navPanel.add(end, 4, 0);

navPanel.setAlignment(Pos.TOP\_CENTER);

bPane = new BorderPane(picLabel,navPanel,null,directoryNamesBox,null);

bPane.setTop(navPanel);

bPane.setCenter(picLabel);

bPane.setBottom(directoryNamesBox);

scene = new Scene(bPane);

//building the GUI------------------------------------------------------------------------------------------

//--------------------------------------------------------------------------------------------------------------

directoryNamesBox.getItems().addAll("P:\\Fall 2016\\CSC 205\\Pictures1",

"P:\\Fall 2016\\CSC 205\\Pictures2","P:\\Fall 2016\\CSC 205\\Homework");

}

//-----------------------------------------------------------------------------------------------------------------

public void addPictures(String directoryName)

{

try

{

//-------------------------------------------------------------------------------------------------------------

// creates a new empty doubly-linked-list

//-------------------------------------------------------------------------------------------------------------

myList = new DoublyLinkedList<String>();

//-------------------------------------------------------------------------------------------------------

// Gets the File object from the directory name

//-------------------------------------------------------------------------------------------------------

File rootDirectory = new File((java.lang.String)directoryName);

//-------------------------------------------------------------------------------------------------------

// If it is not a directory then it reports an error to the user

//-------------------------------------------------------------------------------------------------------

if (rootDirectory.isDirectory() == false)

{

JOptionPane.showMessageDialog(null,

"ERROR: You must specify a directory from where to load pictures");

System.out.println(

"ERROR: You must specify a directory from where to load pictures");

}

//-------------------------------------------------------------------------------------------------------

// Otherwise look at the contents individually

//-------------------------------------------------------------------------------------------------------

else

{

try{

// Make an array of files in this directory

// MyFilter object is used select only files that

// have a .jpg, .jpeg, .pnj, or .tif extensions

//------------------------------------------------------------------------------------------------------------------

String[] filesInDirectory =

(String [])rootDirectory.list(new MyFilter());

//------------------------------------------------------------------------------------------------------

// Goes thru files one by one and adds them to the doubly linked list

//----------------------------------------------------------------------------------------------------

for (int cnt = 0; cnt < filesInDirectory.length; cnt++)

{

String nextFile = filesInDirectory[cnt];

String fullFileName = (String)(rootDirectory.getCanonicalPath()

+ File.separator + nextFile);

myList.insertAtEnd(fullFileName);

}

//-------------------------------------------------------------------------------------------------------

// if any pictures are found

//----------------------------------------------------------------------------------------------------

if (filesInDirectory.length > 0)

{

myList.resetThisElement();

currentIndex = 1;

showCountLabel.setText(currentIndex+" of " + myList.length());

displayImage(picLabel, myList.getThisElement());

}

else

JOptionPane.showMessageDialog(null,

"No Pictures in that folder." + "Choose another folder");

} catch(IOException ioe)

{

System.out.println("ERROR: IOException");

}

}

}

catch (SecurityException ex)

{

System.out.println("ERROR: Security Exception: " + ex.toString());

}

}

//-------------------------------------------------------------------------------------------------------------------

public void attachHandlers()

{

//-------------------------------------------------------------------------------------------------------------------

directoryNamesBox.setOnAction(new EventHandler<ActionEvent>()

{

public void handle(ActionEvent e)

{

String imgDirectName = directoryNamesBox.getSelectionModel().getSelectedItem();

addPictures(imgDirectName);

}

});

begin.setOnAction(new EventHandler<ActionEvent>()

{

public void handle(ActionEvent e)

{

myList.resetThisElement();

currentIndex = 1;

showCountLabel.setText(currentIndex+" of " + myList.length());

displayImage(picLabel, myList.getThisElement());

}

});

forward.setOnAction(new EventHandler<ActionEvent>()

{

public void handle(ActionEvent e)

{

// todo error handling

if ( currentIndex < myList.length() ) {

currentIndex++;

myList.nextElement();

showCountLabel.setText(currentIndex+" of " + myList.length());

displayImage(picLabel, myList.getThisElement());

}

}

});

backword.setOnAction(new EventHandler<ActionEvent>()

{

public void handle(ActionEvent e)

{

if (currentIndex > 1 ) {

currentIndex--;

myList.prevElement();

showCountLabel.setText(currentIndex+" of " + myList.length());

displayImage(picLabel, myList.getThisElement());

}

}

});

end.setOnAction(new EventHandler<ActionEvent>()

{

public void handle(ActionEvent e)

{

currentIndex = myList.length();

myList.resetThisElementToLast();

showCountLabel.setText(currentIndex+" of " + myList.length());

displayImage(picLabel, myList.getThisElement());

}

});

}

//-------------------------------------------------------------------------------------------------------------------

public void displayImage(Label label, String imgFileName)

{

// Creates the Image object from the image

Image image;

try {

image = new Image(new FileInputStream(imgFileName));

//-----------------------------------------------------------------------------------------------------------------

// setGraphic to the centerLabel-----------------------------------------------------------------------------

//-----------------------------------------------------------------------------------------------------------------

label.setGraphic(new ImageView(image));

} catch (FileNotFoundException excep)

{

excep.printStackTrace();

}

}

//-----------------------------------------------------------------------------------------------------------------

public void start(Stage primaryStage)

{

createGUIComponents();

attachHandlers();

//-------------------------------------------------------------------------------------------------------------------

//Creates the given scene and places it on the stage------------------------------------------------------

//-----------------------------------------------------------------------------------------------------------------

primaryStage.setTitle("Slide show");

primaryStage.setScene(scene);

primaryStage.setWidth(600);

primaryStage.setHeight(700);

primaryStage.show();

}

//-----------------------------------------------------------------------------------------------------------------

public static void main(String[] args)

{

Application.launch(args);

}

}