/\*\*

\*

\* @author Steven Glasford

\* @version 1.00 2-28-19

\*/

public class Client {

/\*\*

\* @param args none

\*/

public static void main(String[] args) {

//

LuckyNumberList lucky = new LuckyNumberList();

//These are some of my names, notice how foreign some are

LuckyNumber name1 = new LuckyNumber("Jules");

LuckyNumber name2 = new LuckyNumber("Patty");

LuckyNumber name3 = new LuckyNumber("Ciao");

LuckyNumber name4 = new LuckyNumber("Glove");

LuckyNumber name5 = new LuckyNumber("Dumb");

LuckyNumber name6 = new LuckyNumber("Bri");

LuckyNumber name7 = new LuckyNumber("Table");

LuckyNumber name8 = new LuckyNumber("Steven");

LuckyNumber name9 = new LuckyNumber("Pharell");

LuckyNumber name0 = new LuckyNumber("Pitbull");

//add the names to the list

lucky.addLuckyNumber(name0);

lucky.addLuckyNumber(name1);

lucky.addLuckyNumber(name2);

lucky.addLuckyNumber(name3);

lucky.addLuckyNumber(name4);

lucky.addLuckyNumber(name5);

lucky.addLuckyNumber(name6);

lucky.addLuckyNumber(name7);

lucky.addLuckyNumber(name8);

lucky.addLuckyNumber(name9);

//create a default list iterator

Iterator<Position> luckyListIterator = lucky.positions().iterator();

//create a prime list iterator

Iterator<Position<LuckyNumber>> primeListIterator =

lucky.primePositions().iterator();

//create a

Iterator<Position<LuckyNumber>> evenListIterator =

lucky.evenPositions().iterator();

System.out.println("Print out all of the Bitches.");

String defaultList = "";

String evenList = "";

String primeList = "";

//create the default list

while (luckyListIterator.hasNext()){

LuckyNumber temp =

(LuckyNumber) luckyListIterator.next().getElement();

defaultList = defaultList + temp.toString() + "\t\t";

//if the thing is even print out an even string

if (temp.isEven())

defaultList += "Even\t\t";

//if it isn't even print out an odd

else

defaultList += "Odd\t\t";

if(temp.isPrime())

defaultList += "Prime\n";

else

defaultList += "Not Prime\n";

}

//print out the default

System.out.println(defaultList);

//create the prime list

System.out.println("\nUsing the PrimeListIterator");

while (primeListIterator.hasNext()){

LuckyNumber temp =

(LuckyNumber) primeListIterator.next().getElement();

primeList = primeList + temp.toString() + "\t\t";

if (temp.isEven())

primeList += "Even\t\t";

else

primeList += "Odd\t\t";

if(temp.isPrime())

primeList += "Prime\n";

else

primeList += "Not Prime\n";

}

//print out the prime list

System.out.println(primeList);

//create the even list stuff

System.out.println("\nUsing the EvenListIterator");

while (evenListIterator.hasNext()){

LuckyNumber temp =

(LuckyNumber) evenListIterator.next().getElement();

evenList = evenList + temp.toString() + "\t\t";

if (temp.isEven())

evenList += "Even\t\t";

else

evenList += "Odd\t\t";

if(temp.isPrime())

evenList += "Prime\n";

else

evenList += "Not Prime\n";

}

//print out the even list

System.out.println(evenList);

}

}

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Section 7.4.1

\*/

public interface Iterable<E> {

Iterator<E> iterator( ); // Returns an iterator of the elements in the collection

}

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Section 7.4

\*/

public interface Iterator<E> {

boolean hasNext( ); // Returns true if there is at least one additional

// element in the sequence, and false otherwise.

E next( ); // Returns the next element in the sequence.

void remove( ) throws IllegalStateException;

// Removes from the collection the element returned by

// the most recent call to next( ). Throws an

// IllegalStateException if next has not yet been called,

// or if remove was already called since the most recent

// call to next.

}

import java.util.NoSuchElementException;

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragements 7.9, 7.10, 7.11, 7.12 & 7.14

\*

\* toString method added by Latimer

\*/

/\*\* Implementation of a positional list stored as a doubly linked list. \*/

public class LinkedPositionalList<E> implements PositionalList<E> {

//----- nested Node class -----

private static class Node<E> implements Position<E> {

private E element; // reference to the element stored at this node

private Node<E> prev; // reference to the prevous node in the list

private Node<E> next; // reference to the subsequent node in the list

public Node( E e, Node<E> p, Node<E> n ){

element = e;

prev = p;

next = n;

}

@Override

public E getElement( ) throws IllegalStateException

{

if ( next == null )

throw new IllegalStateException( "Position no longer valid." );

return element;

}

public Node<E> getPrev( )

{

return prev;

}

public Node<E> getNext( )

{

return next;

}

public void setElemetn( E e )

{

element = e;

}

public void setPrev( Node<E> p )

{

prev = p;

}

public void setNext( Node<E> n )

{

next = n;

}

} //----- end of nested Node class -----

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragement 7.14

\*/

//----- nested PositionIterator class -----

private class PositionIterator implements Iterator<Position<E>>{

private Position<E> cursor = first(); // position of the next element to report

private Position<E> recent = null; // position of last reported element

/\*\* Tests whether the iterator has a next object. \*/

@Override

public boolean hasNext( ) { return ( cursor != null ); }

/\*\* Returns the next position in the iterator. \*/

@Override

public Position<E> next( ) throws NoSuchElementException {

if ( cursor == null ) throw new NoSuchElementException( "nothing left " );

recent = cursor;

cursor = after( cursor );

return recent;

}

/\*\* Removes the element returned by most recent call to next. \*/

@Override

public void remove( ) throws IllegalStateException {

if ( recent == null ) throw new IllegalStateException( "nothing to remove" );

LinkedPositionalList.this.remove( recent ); // remove from outer list

recent = null; // do not allow remove again until next is called

}

} //----- end of nested PositionIterator class -----

//----- nested PositionIterable class -----

private class PositionIterable implements Iterable<Position<E>>{

@Override

public Iterator<Position<E>> iterator( ) { return new PositionIterator( ); }

} //----- end of nested PositionIterable class -----

/\*\* Returns an iterable representation of the list's positions.

\* @return \*/

public Iterable<Position<E>> positions( ) {

return new PositionIterable( ); // create a new instance of the inner class

}

//----- nested ElementIterator class -----

/\* This class adapts the iteration produced by positions( ) to return elements. \*/

private class ElementIterator implements Iterator<E> {

Iterator<Position<E>> posIterator = new PositionIterator( );

@Override

public boolean hasNext( ) { return posIterator.hasNext( ); }

@Override

public E next( ) { return posIterator.next( ).getElement( ); } // return element

@Override

public void remove( ) { posIterator.remove( ); }

}

/\*\* Returns an iterator of the elements stored in the list \*/

public Iterator<E> iterator( ) { return new ElementIterator( ); }

// instance variables of the LinkedPositionalList

private Node<E> header; // header sentinel

private Node<E> trailer; // trailer sentinel

private int size = 0; // number of elements in the list

public LinkedPositionalList( ){

header = new Node<>( null, null, null ); // create header

trailer = new Node<>( null, header, null ); // create trailer is preceded by header

header.setNext(trailer); // header is followed by trailer

}

// private utilities

/\*\*

\* @param p position to validate

\* @return node if position is valid

\* @throws IllegalArgumentException if p no longer in list or p is not a position

\*/

private Node<E> validate( Position<E> p ) throws IllegalArgumentException {

if( !(p instanceof Node )) throw new IllegalArgumentException( "Invalid p" );

Node<E> node = ( Node<E> ) p; // safe cast

if ( node.getNext() == null )

throw new IllegalArgumentException( "p is no longer in the list" );

return node;

}

/\*\*

\* @param node to be returned as position if not header or trailer

\* @return position of node

\*/

private Position<E> position( Node<E> node ){

if ( node == header || node == trailer )

return null;

return node;

}

// public accessor methods

/\*\*

\* @return number of elements in linked list

\*/

@Override

public int size( ){

return size;

}

/\*\*

\* @return true if list is empty, false other wise

\*/

@Override

public boolean isEmpty( ){

return ( size == 0 );

}

/\*\*

\* @return the first position in linked list (null if empty).

\*/

@Override

public Position<E> first( ){

return position( header.getNext( ) );

}

/\*\*

\* @return the last position in linked list (null if empty).

\*/

@Override

public Position<E> last( ){

return position( trailer.getPrev( ) );

}

/\*\*

\* @param p position to get position immediately before

\* @return position before p

\* @throws IllegalArgumentException if p not valid

\*/

@Override

public Position<E> before( Position<E> p ) throws IllegalArgumentException{

Node<E> node = validate( p );

return position( node.getPrev( ) );

}

/\*\*

\* @param p position to get immediately after

\* @return position after p

\* @throws IllegalArgumentException if p not valid

\*/

@Override

public Position<E> after( Position<E> p ) throws IllegalArgumentException{

Node<E> node = validate( p );

return position( node.getNext( ) );

}

// private utilities

/\*\*

\* @param e element to be added

\* @param pred node to add element after

\* @param succ node to add element before

\* @return position of newly added element

\*/

private Position<E> addBetween(E e, Node<E> pred, Node<E> succ ){

Node<E> newest = new Node<>(e, pred, succ); // create and link new node

pred.setNext(newest);

succ.setPrev(newest);

size++;

return newest;

}

// public update methods

/\*\*

\* @param e element to be added just after header

\* @return position of newly added element

\*/

@Override

public Position<E> addFirst(E e) {

return addBetween( e, header, header.getNext() );

}

/\*\*

\* @param e element to be added just before trailer

\* @return position of newly added element

\*/

@Override

public Position<E> addLast( E e ){

return addBetween(e, trailer.getPrev( ), trailer );

}

/\*\*

\*

\* @param p position to add element before

\* @param e element to be added

\* @return position of newly added element

\* @throws IllegalArgumentException if p is not valid

\*/

@Override

public Position<E> addBefore( Position<E> p, E e ) throws IllegalArgumentException {

Node<E> node = validate( p );

return addBetween(e, node.getPrev( ), node );

}

/\*\*

\* @param p position to add element after

\* @param e element to be added

\* @return position of newly added element

\* @throws IllegalArgumentException if p is not valid

\*/

@Override

public Position<E> addAfter( Position<E> p, E e ) throws IllegalArgumentException {

Node<E> node = validate( p );

return addBetween(e, node, node.getNext( ) );

}

/\*\*

\* @param p position of node to update

\* @param e new element for node

\* @return old element in node before update

\* @throws IllegalArgumentException if p not valid

\*/

@Override

public E set( Position<E> p, E e ) throws IllegalArgumentException {

Node<E> node = validate( p );

E answer = node.getElement( );

node.setElemetn( e );

return answer;

}

/\*\*

\* @param p position to be removed

\* @return element that was removed

\* @throws IllegalArgumentException if p not valid

\*/

public E remove( Position<E> p ) throws IllegalArgumentException {

Node<E> node = validate( p );

Node<E> predecessor = node.getPrev();

Node<E> successor = node.getNext();

predecessor.setNext( successor );

successor.setPrev( predecessor );

size--;

E answer = node.getElement( );

node.setElemetn( null );

node.setNext( null );

node.setPrev( null );

return answer;

}

}

/\*\*

\* A class of the form LuckyNumber.

\* @author Steven Glasford

\* @version 2-26-2019

\*/

public class LuckyNumber {

String name = null;

int luckyNumber;

/\*\*

\* Constructor of the piece of shit.

\* @param name A name to be inserted

\*/

LuckyNumber(String name) {

//get the name from the constructor

this.name = name;

//randomly get a number between 0 and 9 inclusively

luckyNumber = (int) (Math.random() \* 10);

}

/\*\*

\* Get the name out of the class.

\* @return the name of the person.

\*/

public String getName() {

return name;

}

/\*\*

\* Get the luckyNumber out of the class.

\* @return The Lucky Number

\*/

public int getLuckyNumber(){

return luckyNumber;

}

/\*\*

\* Convert the information in the class to a string.

\* @return the string of information contained in the luckyNumber class.

\*/

public String toString(){

return getName() + "\t\t" + getLuckyNumber();

}

/\*\*

\* Determine if the number contained in the LuckyNumber class is a prime.

\* @return Whether the number is a prime number.

\*/

public boolean isPrime() {

return ((luckyNumber == 2) ||

(luckyNumber == 3) ||

(luckyNumber == 5) ||

(luckyNumber == 7));

}

/\*\*

\* Determine if the number is a prime.

\* @return Whether or not the number is even

\*/

public boolean isEven() {

return ((luckyNumber % 2) == 0);

}

}

import java.util.NoSuchElementException;

/\*\*

\* A list of lucky Numbers.

\* @author Steven Glasford

\* @version 1.00 2-26-2019

\*/

public class LuckyNumberList {

private LinkedPositionalList luckyNumber = null;

/\*\*

\* Constructor builds an empty LinkedPositionalList.

\*/

public LuckyNumberList (){

luckyNumber = new LinkedPositionalList();

}

/\*\*

\* Add a number to the list.

\* @param item The item you want to add.

\*/

public void addLuckyNumber(LuckyNumber item){

luckyNumber.addLast(item);

}

/\*\*

\* Determine if the number is even.

\* @param item the number you want to determine if the lucky number

\* is even

\* @return Whether the item is even

\*/

public boolean isEven(LuckyNumber item){

return ((item.getLuckyNumber() % 2) == 0);

}

/\*\*

\* Determine if the number is a prime. This will only work for numbers

\* between 0 and 9, which shouldn't be an issue in this program, since the

\* value never exceeds 9, or 0.

\* @param item The item you want to determine its primeness

\* @return Whether the item is a prime

\*/

public boolean isPrime(LuckyNumber item){

return ((item.getLuckyNumber() == 2) ||

(item.getLuckyNumber() == 3) ||

(item.getLuckyNumber() == 5) ||

(item.getLuckyNumber() == 7));

}

/\*\*

\* Convert the thing to a string.

\* @return a string.

\*/

public String toString(){

String returnString = "";

Iterator listIterator = luckyNumber.iterator();

while (listIterator.hasNext()){

returnString += listIterator.next() + "\n";

}

return returnString;

}

//

// The following classes are the nested Iterator classes from

// Code Fragment 7.14

//

// Only the classes for the Position Itertor have bee included.

//

// These fragements have been modified so that they are specific to the

// Alphabet class.

//

// It is necessary to put the iterator code here since we want to create

// iterators specifically for the Alphabet class which is a concrete class

// based on the generic ADT LinkedPositionalList.

//

// Our code needs to have knowledge of Letter.

//

// Generally the Generic placeholders <E> have been replaced with

// concret references <Letter>

// AND

// Call to LinkedPositionalList methods have be replaced by calls using

// the instance reference alphabet

// e.g.

// private Position<Letter> cursor = first();

// became

// private Position<Letter> cursor = alphabet.first();

//

//----- nested PositionIterator class -----

private class PositionIterator implements Iterator<Position>{

// position of the next element to report

private Position cursor = luckyNumber.first();

// position of last reported element

private Position recent = null;

/\*\* Tests whether the iterator has a next object. \*/

@Override

public boolean hasNext( ) { return ( cursor != null ); }

/\*\* Returns the next position in the iterator. \*/

@Override

public Position next( ) throws NoSuchElementException {

if ( cursor == null ) throw new NoSuchElementException(

"nothing left " );

recent = cursor;

cursor = luckyNumber.after( cursor );

return recent;

}

/\*\* Removes the element returned by most recent call to next. \*/

@Override

public void remove( ) throws IllegalStateException {

if ( recent == null ) throw new IllegalStateException(

"nothing to remove" );

// remove from outer list

luckyNumber.remove( recent );

// do not allow remove again until next is called

recent = null;

}

} //----- end of nested PositionIterator class -----

//----- nested PositionIterable class -----

private class PositionIterable implements Iterable<Position>{

@Override

public Iterator<Position> iterator( ) { return new

PositionIterator( ); }

} //----- end of nested PositionIterable class -----

/\*\* Returns an iterable representation of the list's positions.

\* @return \*/

public Iterable<Position> positions( ) {

// create a new instace of the inner class

return new PositionIterable( );

}

//----- nested PositionIterator class -----

private class EvenPositionIterator implements

Iterator<Position<LuckyNumber>>{

// position of the next element to report

private Position<LuckyNumber> cursor = luckyNumber.first();

// position of last reported element

private Position<LuckyNumber> recent = null;

/\*\* Tests whether the iterator has a next object. \*/

@Override

public boolean hasNext( ) { return ( cursor != null ); }

/\*\* Returns the next position in the iterator. \*/

@Override

public Position<LuckyNumber> next( ) throws NoSuchElementException {

//<<< new code

// On the first call to next (i.e. when recent == null) you need to

//<<< new code

// advance recent until it is pointing to a vowel element.

//<<< new code

if ( recent == null )

//<<< new code

{

//determine if the thing is even

while ( cursor != null && !isEven(cursor.getElement()) )

//<<< new code

cursor = luckyNumber.after( cursor );

//<<< new code

}

if ( cursor == null ) throw new NoSuchElementException(

"nothing left " );

recent = cursor;

cursor = luckyNumber.after( cursor );

// advance cursor to the next vowel

while ( cursor != null && !isEven(cursor.getElement()) )

cursor = luckyNumber.after( cursor );

return recent;

}

/\*\* Removes the element returned by most recent call to next. \*/

@Override

public void remove( ) throws IllegalStateException {

if ( recent == null ) throw new IllegalStateException(

"nothing to remove" );

luckyNumber.remove( recent ); // remove from outer list

// do not allow remove again until next is called

recent = null;

}

public boolean isEven(LuckyNumber item) {

return ((item.getLuckyNumber() % 2) == 0);

}

}

//----- end of nested PositionIterator class -----

private class EvenPositionIterable implements Iterable<Position<LuckyNumber>>{

@Override

public Iterator<Position<LuckyNumber>> iterator( ) { return new

EvenPositionIterator( );

}

}

//----- end of nested PositionIterable class -----

/\*\* Returns an iterable representation of the list's positions.

\* @return \*/

public Iterable<Position<LuckyNumber>> evenPositions( ) {

// create a new instace of the inner class

return new EvenPositionIterable( );

}

//----- nested PositionIterator class -----

private class PrimePositionIterator implements

Iterator<Position<LuckyNumber>>{

// position of the next element to report

private Position<LuckyNumber> cursor = luckyNumber.first();

// position of last reported element

private Position<LuckyNumber> recent = null;

/\*\* Tests whether the iterator has a next object. \*/

@Override

public boolean hasNext( ) { return ( cursor != null ); }

/\*\* Returns the next position in the iterator. \*/

@Override

public Position<LuckyNumber> next( ) throws NoSuchElementException {

//<<< new code

// On the first call to next (i.e. when recent == null) you need to

//<<< new code

// advance recent until it is pointing to a vowel element.

//<<< new code

if ( recent == null )

//<<< new code

{

//<<< new code

while ( cursor != null && !isPrime(cursor.getElement()) )

//<<< new code

cursor = luckyNumber.after( cursor );

//<<< new code

}

if ( cursor == null ) throw new NoSuchElementException(

"nothing left " );

recent = cursor;

cursor = luckyNumber.after( cursor );

// advance cursor to the next vowel

while ( cursor != null && !isPrime(cursor.getElement()) )

cursor = luckyNumber.after( cursor );

return recent;

}

/\*\* Removes the element returned by most recent call to next. \*/

@Override

public void remove( ) throws IllegalStateException {

if ( recent == null ) throw new IllegalStateException(

"nothing to remove" );

luckyNumber.remove( recent ); // remove from outer list

// do not allow remove again until next is called

recent = null;

}

public boolean isPrime(LuckyNumber item) {

return ((item.getLuckyNumber() == 2) ||

(item.getLuckyNumber() == 3) ||

(item.getLuckyNumber() == 5) ||

(item.getLuckyNumber() == 7));

}

}

//----- end of nested PositionIterator class -----

private class PrimePositionIterable implements Iterable<Position<LuckyNumber>>{

@Override

public Iterator<Position<LuckyNumber>> iterator( ) { return new

PrimePositionIterator( );

}

}

//----- end of nested PositionIterable class -----

/\*\* Returns an iterable representation of the list's positions.

\* @return \*/

public Iterable<Position<LuckyNumber>> primePositions( ) {

// create a new instace of the inner class

return new PrimePositionIterable( );

}

}

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragement 7.8

\*/

/\*\*

\* An interface for positional lists.

\*/

public interface PositionalList<E> {

/\*\*

\* @return the number of elements in the list.

\*/

int size( );

/\*\*

\* @return true if the list is empty.

\*/

boolean isEmpty( );

/\*\*

\* @return the first Position in the list ( or null, if empty ).

\*/

Position<E> first( );

/\*\*

\* @return the last Position in the list ( or null, if empty ).

\*/

Position<E> last( );

/\*\*

\* @param p a position in the list,

\* @return position immediately before p ( or null if p is first ).

\* @throws IllegalArgumentException if p is not in list.

\*/

Position<E> before( Position<E> p ) throws IllegalArgumentException;

/\*\*

\* @param p a position in the list,

\* @return position immediately after p ( or null if p is last ).

\* @throws IllegalArgumentException if p is not in list.

\*/

Position<E> after( Position<E> p ) throws IllegalArgumentException;

/\*\*

\* @param e element to be inserted at front of list

\* @return position of inserted element

\*/

Position<E> addFirst( E e );

/\*\*

\* @param e element to be inserted at back of list

\* @return position of inserted element

\*/

Position<E> addLast( E e );

/\*\*

\* @param p position to be inserted before

\* @param e element to be inserted before position p

\* @return position of e

\* @throws IllegalArgumentException if p not in list

\*/

Position<E> addBefore( Position<E> p, E e ) throws IllegalArgumentException;

/\*\*

\* @param p position to be inserted after

\* @param e element to be inserted after position p

\* @return position of e

\* @throws IllegalArgumentException if p not in list

\*/

Position<E> addAfter( Position<E> p, E e ) throws IllegalArgumentException;

/\*\*

\* @param p position to store element at

\* @param e element to be stored at p

\* @return the element that is replaced

\* @throws IllegalArgumentException if p is not in list

\*/

E set( Position<E> p, E e ) throws IllegalArgumentException;

/\*\*

\* @param p position of element to be removed

\* @return removed element

\* @throws IllegalArgumentException if p not in list

\*/

E remove( Position<E> p ) throws IllegalArgumentException;

}

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragement 7.7

\*/

public interface Position<E> {

/\*\*

\* Returns the element stored at this position.

\*

\* @return the stored element

\* @thorws IllegalStateExceptoin if position no longer valid

\*/

E getElement( ) throws IllegalStateException;

}