

```

// The Game of Adjacency - Lab 6
// Sample Solution by Anthony Cox

class Adjacency {
    static final int SIZE = 8; // The size of the board
    static final int CORNERS = 2; // The size of initial starting blocks
    static final char BLANK = '_'; // The char for blank squares

    // Make a new game board and put on it the initial pieces
    // Returns a new board of type char[SIZE][SIZE]
    // public static char[][] setupBoard ()
    {
        // Create new board
        char[][] board = new char[SIZE][SIZE];

        // Initialize the board
        int row, col;
        for (row = 0; row < SIZE; row++) {
            for (col = 0; col < SIZE; col++) {
                board[row][col] = BLANK;
            }
        }

        // Put tokens on opposite CORNER sized areas
        for (row = 0; row < CORNERS; row++) {
            for (col = 0; col < CORNERS; col++) {
                board[row][col] = 'X';
                board[(SIZE-1)-row][(SIZE-1)-col] = 'O';
            }
        }

        return (board);
    }

    // Display the game board as unicode chars
    // board = the game board to display
    // public static void printBoard (char[][] board)
    {
        // Start with the top row and work down the board
        for (int row = SIZE-1; row >= 0; row--) {
            for (int col = 0; col < SIZE; col++) {
                System.out.print (board[row][col]);
            }
            System.out.println ( " " + (row+1) );
        }
        for (int col = 0; col < SIZE; col++) {
            System.out.print ( (col+1) );
        }
        System.out.println();
    }

    // Update the game board if player has chosen coord (x,y)
    // (x,y) = (col,row) of selected board square
    // player = the current player, either 'X' or 'O'
    // board = the game board
    // Returns the updated game board
    // public static void updateBoard (char[][] board, char player, int x, int y)
    {
        board[y][x] = player;
        for ( int row = y - 1; row <= y + 1; row++ ) {
            for ( int col = x - 1; col <= x + 1; col++ ) {
                if ( isValid(board, row, col) )
                    if ( ((player == 'X') && (board[row][col] == 'O')) ||
                        ((player == 'O') && (board[row][col] == 'X')) )
                    board[row][col] = player;
            }
        }

        // Are coordinates (x,y) valid for the current board?
        // Returns true if valid, false otherwise
        // static boolean isValid (char[][] board, int x, int y)
        {
            return ( (x >= 0) && (x < SIZE) && (y >= 0) && (y < SIZE));
        }

        // Are coordinates (x,y) on the current board empty?
        // Returns true if empty, false otherwise
        // static boolean isEmpty (char[][] board, int x, int y)
        {
            // protect array access with lazy evaluation
            return isValid(board,x,y) && (board[y][x] == BLANK);
        }

        // Determine the final score, display it and announce a winner
        // board = the game board in its final configuration
        // static void printWinner (char[][] board)
        {
            int xScore = 0, // player X's score
                oScore = 0; // player O's score

            for (int row = 0; row < SIZE; row++) {
                for (int col = 0; col < SIZE; col++) {
                    if (board[row][col] == 'X') {
                        xScore++;
                    } else if (board[row][col] == 'O') {
                        oScore++;
                    }
                }
            }

            System.out.println ("");
            System.out.println ("Final Score");
            System.out.println ("Player X: " + xScore);
            System.out.println ("Player O: " + oScore);
            System.out.println ("");

            if (xScore > oScore) {
                System.out.println ("Player X wins!");
            } else if (oScore > xScore) {
                System.out.println ("Player O wins!");
            } else {
                System.out.println ("Its a tie!");
            }
        }

        // Mainline
        // public static void main (String[] s)
        {
            char[][] board = setupBoard (); // Initialized game board
            char player = 'X'; // Current player
            int x, y; // Co-ordinates of a move

```

```
int maxTurns = SIZE * SIZE - (2 * (CORNERS * CORNERS));
boolean endOfGame = false;

System.out.println ("Initial Board");
printBoard (board);

// Labelled break avoids using flags to indicate completion
for (int turn = 1; turn <= maxTurns; turn++) {

    System.out.println ("");
    System.out.println ("Turn #" + turn + " (Player " + player + ")");

    // Get the next location
    do {
        // Input before println so it looks nice with piped input
        x = Stdin.getInt (); // column of new piece
        System.out.println ("What is the column? " + x);
        y = Stdin.getInt (); // row of new piece
        System.out.println ("What is the row? " + y);
        if ((x == 0) && (y == 0)) {
            endOfGame = true;
            break;
        }
        x--; y--; // Adjust for indexing from 0
    } while( !isEmpty( board, x, y ) );

    if (endOfGame)
        break;

    // Update the board and display it
    System.out.println ("");
    updateBoard(board, player, x, y);
    printBoard (board);

    // Change players
    if(player == 'X') {
        player = 'O';
    } else {
        player = 'X';
    }
} // go back and ask for next move

    printWinner (board);
}

} // end of class Adjacent
```

```
//
// Filled in skeleton for a Year object
//
class Year
{
    private int year;

    // Constructor
    // - Prevents an invalid year object from being created
    //
    Year (int y)
    {
        if (y < 0) {
            System.out.println ("Error: Year class cannot represent negative years");
            System.out.println ("    Using 0 instead of " + y);
            y = 0;
        }
        year = y;
    }

    // Does the object represent a leap year? (returns true or false)
    //
    public boolean isLeapYear ()
    {
        if (year > 1582) {
            if ((year % 400) == 0) return (true);
            else if ((year % 100) == 0) return (false);
            else if ((year % 4) == 0) return (true);
        } else {
            if ((year % 4) == 0) return (true);
            return (false);
        }
    }

    // What century is this object temporally located in?
    //
    public int century ()
    {
        return ((year / 100) + 1);
    }
}

// Container class for mainline
//
class Demo
{
    // Trivial mainline to use a Year object
    //
    public static void main (String[] args)
    {
        System.out.print ("Enter an positive integer representing a year: ");
        int y = StdIn.getInt ();

        Year year = new Year (y);
        if (year.isLeapYear ()) {
            System.out.println (y + " is a leap year.");
        } else {
            System.out.println (y + " is not a leap year.");
        }
        System.out.println (y + " is in century " + year.century() + ".");
    }
}
```

```

// The Bank Manager: Version 2 Using Objects
// Sample Solution by Anthony Cox

//
// Class to represent a bank account
//
class Account
{
    private int balance; // Current account balance
    private char type; // (S)avings or (C)hequing
    private int interestRate; // Interest rate to use
    private int minBalance; // Balance since last interest

    // Constructor
    // t = type of account: (S)avings or (C)hequing
    // ir = interest rate as an integer percentage
    //
    Account (char t, int ir) {
        balance = 0;
        minBalance = 0;
        type = t;
        interestRate = ir;
    }

    //
    // Get a monetary value >= 0 by prompting for dollars and cents
    // Returns the amount in cents
    //
    private static int getAmount ()
    {
        int dollars, cents;

        dollars = Bank.promptInt ("How many dollars? ");
        while (dollars < 0) {
            System.out.println ("Error: Invalid dollar amount");
            dollars = Bank.promptInt ("How many dollars? ");
        }
        cents = Bank.promptInt ("How many cents? ");
        while ((cents < 0) || (cents > 99)) {
            System.out.println ("Error: Invalid cent amount.");
            cents = Bank.promptInt ("How many cents? ");
        }
        return (100 * dollars + cents);
    }

    //
    // Print a correctly formatted monetary amount
    //
    private static void printAmount (int amount)
    {
        int cents = amount % 100;
        System.out.print (" $" + (amount / 100) + ".");
        if (cents == 0) {
            System.out.print ("00");
        } else if (cents < 10) {
            System.out.print ("0" + cents);
        } else {
            System.out.print (cents);
        }
    }

    //
    // Get a deposit amount and add it to the current balance
    //
    public void deposit ()
    {
        balance += getAmount ();
    }

    //
    // Get a withdrawal amount and subtract it from the current balance
    //
    public void withdraw ()
    {
        int amount = getAmount ();
        while (amount > balance) {
            System.out.println ("Error: Amount exceeds available funds.");
            amount = getAmount ();
        }
        balance -= amount;
        if (balance < minBalance )
            minBalance = balance;
        return (amount);
    }

    //
    // Print out the current account balance
    //
    public void balance ()
    {
        if ( type == 'S' )
            System.out.println ("Savings account balance: ");
        else
            System.out.println ("Chequing account balance: ");
        printAmount (balance);
        System.out.println ("");
    }

    //
    // Calculate interest for this account
    //
    public void interest ()
    {
        balance += (interestRate * minBalance) / 100;
        minBalance = balance;
    }

    //
    // Transfer an amount from this account to another account
    //
    public void transferFrom (Account source)
    {
        balance += source.withdraw ();
    }

    //
    // End of class Account
    //
}

//
// Class to represent a small bank
//
class Bank
{
    private static final int MAX_ACCOUNTS = 100; // Max. number of accounts
    // Interest rates for new accounts (with default initialization values)
    private static int savingRate= 5;
    private static int chequingRate = 3;

    // System security password
    private static final String password = "";

    // Index of the next new account and total number of open accounts

```

```

private static int openAccounts = 0;
// Array of accounts held at this bank
private static Account[] accounts = new Account[MAX_ACCOUNTS];

//
// Return an integer value using the supplied prompting string
// prompt = prompting string
//
public static int promptInt (String prompt)
{
    System.out.print (prompt);
    return (Stdin.getInt ());
}

//
// Verify password and select the operation to perform
// Returns an uppercase operation identification character
// Note: does not verify if this is a valid operation
// Returns '*' if password not correctly entered
//
private static char getTransaction ()
{
    System.out.print ("Enter the password: ");
    String pw = Stdin.getString ();
    if (pw.compareTo (password) != 0) {
        return (*);
    }
    System.out.println ("(D)eposit, (W)ithdraw, (B)alance, (T)ransfer, apply (I)nteres
t,");
    System.out.println ("apply (I)nterest, (S)et interest rates, (N)ew account, (O)uit
");
    System.out.print ("Operation? ");
    return Stdin.getString().toUpperCase().charAt(0);
}

//
// Select a valid account to operate on
// Returns the index of the account or -1 on error
//
private static int getAccount ()
{
    if (openAccounts == 0) {
        System.out.println ("Error: No accounts exist.");
        return (-1);
    }
    int number = promptInt ("Account number? ");
    if ((number < 0) || (number >= openAccounts)) {
        System.out.println ("Error: Invalid account number.");
        return (-1);
    }
    return (number);
}

//
// Ask for a new interest rate.
//
private static int getInterestRates( String accountType )
{
    // get the new interest rate
    int rate = promptInt ("What is the new " + accountType + " interest rate? ");
    while (rate < 0) {
        System.out.println ("Error: Negative interest rate entered. ");
        rate = promptInt ("What is the new " + accountType + " interest rate? ");
    }
}

return rate;
}

//
// Open a new account
//
private static void openNewAccount()
{
    if (openAccounts >= MAX_ACCOUNTS) {
        System.out.println ("Error: Maximum number of accounts exceeded.");
    }
    else {
        System.out.print ("(S)avings or (C)hequing? ");
        char type = Stdin.getString().toUpperCase().charAt(0);
        if (type == 'S') {
            accounts[openAccounts] = new Account (type, savingRate);
            System.out.print ("Created new savings account number: ");
            System.out.println (openAccounts++);
        }
        else if (type == 'C') {
            accounts[openAccounts] = new Account (type, chequingRate);
            System.out.print ("Created new chequing account number: ");
            System.out.println (openAccounts++);
        }
        else {
            System.out.println ("Error: Invalid account type entered.");
        }
    }
}

//
// Do the appropriate transaction.
//
private static void doTransaction( char command )
{
    // DEPOSIT, WITHDRAW, BALANCE, OR TRANSFER
    if (command == 'D' || command == 'W'
        || command == 'B' || command == 'T' )
    {
        // modify one (maybe two) account(s)
        int accountNumber = getAccount();
        if( accountNumber >= 0 ) {
            if (command == 'D')
                accounts[accountNumber].deposit ();
            else if (command == 'W')
                accounts[accountNumber].withdraw ();
            else if (command == 'B')
                accounts[accountNumber].balance ();
            else if (command == 'T') {
                System.out.print ("What is the Source ");
                int source = getAccount ();
                if (source != -1) {
                    accounts[accountNumber].transferFrom (accounts[source]);
                }
            }
        }
    }
    // APPLY INTEREST TO ALL OPEN ACCOUNTS
    else if (command == 'I') {
}

```

```
for (int i = 0; i < openAccounts; i++) {
    accounts[i].interest ();
}

// CHANGE THE INTEREST RATES
} else if (command == 'S') {
    savingRate = getInterestRates( "saving" );
    chequingRate = getInterestRates( "chequing" );
}

// OPEN A NEW ACCOUNT
} else if (command == 'N') {
    openNewAccount();
}

// QUIT
} else if (command == 'Q') {
    System.out.println ( "Program terminating." );
}

// ERROR, BAD PASSWORD
} else if (command == '**') {
    System.out.println ( "Error: Invalid password entered." );
}

// ERROR, UNKNOWN TRANSACTION
} else {
    System.out.println ( "Error: Unknown transaction requested." );
}
}

// Mainline
//
// public static void main (String[] s)
{
    char command;
    int accountNumber;

    System.out.println ( "Bank Account Management Program: Version 2\n" );

    // Loop to continuously perform commands until a 'quit' is requested
    do {
        // Get command also does password verification
        command = getTransaction ();
        doTransaction( command );

        // Put a newline between operations
        System.out.println ( "" );
    } while (command != 'Q');
} // end of mainline
} // end of class Bank
```

```

class Sorts {
//
// Mainline
//
static public void main (String[] args)
{
    double[] items, copy;

    // Loop over the different array sizes
    for (int n = 1000; n < 10001; n+= 1000) {
        // Indicate the size of the array being used
        System.out.println ("Array size = " + n);

        // Create a random array
        items = randomArray (n);

        // Selection sort the array
        copy = arrayCopy (items);
        int s = selectionSort (copy);
        System.out.println (" selection sort: " + s + " comparisons.");

        // Quick sort the array
        copy = arrayCopy (items);
        int q = quicksort (copy, 0, n - 1);
        System.out.println (" quicksort: " + q + " comparisons.");

    } // end of loop to size arrays 1000..10000
} // end of mainline

//
// Return a random array of <size> numbers of type double
//
static public double[] randomArray (int size)
{
    double[] array = new double[size];
    for (int i = 0; i < size; i++) {
        array[i] = Math.random ();
    }
    return (array);
}

//
// Return a copy of an array
//
static public double[] arrayCopy (double[] in)
{
    double[] out = new double[in.length];
    for (int i = 0; i < in.length; i++) {
        out[i] = in[i];
    }
    return (out);
}

//
// Swap the entries at <index1> and <index2> in the array of items
//
static void swap (double[] items, int index1, int index2)
{
    double tmp = items[index1];
    items[index1] = items[index2];
    items[index2] = tmp;
}

//
// Selection sort the array of items
static int selectionSort (double[] items)
{
    // Measure the "work" done
    int comparisons = 0;

    // For all elements of the array (but the last)
    for (int i = 0; i < (items.length - 1); i++) {

        // Find the index of the smallest unselected value
        int minIndex = i;
        for (int j = i; j < items.length; j++) {
            if (items[j] < items[minIndex]) {
                minIndex = j;
            }
            comparisons++;
        }

        // If necessary, put smallest remaining value in current array index
        if (items[i] > items[minIndex]) {
            swap (items, i, minIndex);
        }
        comparisons++;
    }
    return (comparisons);
}

//
// Quicksort the array of items between indices left and right inclusive
static int quicksort (double[] items, int left, int right)
{
    // Measure the "work" done
    int comparisons = 0;

    // Exit if 0 or 1 elements to be sorted
    if ((right - left) < 1) return (comparisons);
    // Exit if sorting range is invalid
    if (left >= right) return (comparisons);

    // Select a pivot value, and set swapping indicies
    double pivot = items[left];
    int l = left + 1;
    int r = right;

    // Partition the array into {< pivot}, pivot, {> pivot}
    while (l <= r) {
        // Find the first item > pivot
        while ((l <= r) && (items[l] <= pivot)) {
            l++;
        }
        comparisons++;

        // Find the last item < pivot
        while ((r >= l) && (items[r] >= pivot)) {
            r--;
        }
        comparisons++;

        // Swap the items on the wrong side of the pivot
        if (l < r) {
            swap (items, l, r);
        }
    }

    // Put pivot between < and > subarrays
    swap (items, left, r);
    comparisons++;
}

```

```
// Recurse and quicksort the subarrays
comparisons += quicksort (items, left, r - 1);
comparisons += quicksort (items, r + 1, right);
return (comparisons);
}
} // end of class Sorts
```

```

import java.net.*;
import java.io.*;
import java.util.*;

// An object to store a phone book entry
class Entry {
    private String surname;
    private String firstname;
    private String phonenumber;

    // Constructor to set instance variables
    Entry (String sn, String fn, String pn)
    {
        surname = sn;
        firstname = fn;
        phonenumber = pn;
    }

    // Display on System.out the formatted contents of an entry
    public void printEntry ()
    {
        System.out.println (surname + ", " + firstname + ": " + phonenumber);
    }

    // Return the surname for printing during binary search
    public String surnameOf ()
    {
        return (surname);
    }

    // Actually implements: less than or equal to
    // Assumes that surnames are a unique key
    public boolean lessthan (Entry e)
    {
        if (surname.compareTo (e.surname) <= 0)
            return (true);
        else
            return (false);
    }

    // Actually implements: greater than or equal to
    // Assumes that surnames are a unique key
    public boolean greaterthan (Entry e)
    {
        if (surname.compareTo (e.surname) >= 0)
            return (true);
        else
            return (false);
    }

    // Compare the entries surname with the parameter
    // - same return values as String.compareTo ()
    public int surnameCompare (String sn)
    {
        return (surname.compareTo (sn));
    }
}

// end of class Entry

// A simple telephone book with binary search based lookup
class PhoneBook
{
    // Maximum number of entries in the phone book
    private static final int NUM_ENTRIES = 50;
    // Array of the entries
    private static Entry[] entry;

    // Mainline
    //
    public static void main (String[] args) {
        System.out.println ("*** Phone Number Locator ***\n");
        entry = getData ();
        quickSortEntries (entry, 0, NUM_ENTRIES-1);

        // Loop to process requests
        String surname = getSurname ();
        while (surname.compareTo ("Exit") != 0) {
            int index = find (entry, surname, 0, NUM_ENTRIES-1);
            if (index >= 0) {
                entry[index].printEntry ();
            } else {
                System.out.println ("Not found.");
            }
            System.out.println ("");
            surname = getSurname ();
        }
        System.out.println ("Program terminating.");
    }

    // Supplied method to create a populated data array
    private static Entry[] getData ()
    {
        Entry[] data = new Entry[NUM_ENTRIES];
        try {
            FileReader fr = new FileReader ("phonedata");
            BufferedReader br = new BufferedReader (fr);
            StringTokenizer st;
            for (int i = 0; i < NUM_ENTRIES; i++) {
                String line = br.readLine();
                st = new StringTokenizer (line);
                data[i] = new Entry (st.nextToken(), st.nextToken());
            }
            fr.read ();
            fr.close ();
        } catch (IOException ioe) {
            System.err.println ("IO Error: " + ioe);
            System.exit(1);
        }
    }
}

```

```

    }
    return (data);
}

//
// Method to prompt for and return a surname to lookup
//
private static String getSurname ()
{
    System.out.print ("Surname to lookup? ");
    return (Stdin.getString ());
}

//
// Swap the entries at index1 and index2 in the array of items
//
static private void swap (Entry[] items, int index1, int index2)
{
    Entry tmp = items[index1];
    items[index1] = items[index2];
    items[index2] = tmp;
}

//
// Quicksort the array of items between indices left and right inclusive
//
static private void quickSortEntries (Entry[] items, int left, int right)
{
    // Exit if 0 or 1 elements to be sorted
    if ((right - left) < 1) return;
    if (left >= right) return;

    // Select a pivot value, and set swapping indicies
    Entry pivot = items[left];
    int l = left + 1;
    int r = right;

    while (l <= r) {
        while ((l <= r) && items[l].lessThan (pivot)) {
            l++;
        }
        while ((r >= l) && items[r].greaterThan (pivot)) {
            r--;
        }
        if (l < r) {
            swap (items, l, r);
        }
    }

    // Put pivot between < and > subarrays
    swap (items, left, r);
    // Recurse and quicksort the subarrays
    quickSortEntries (items, left, r - 1);
    quickSortEntries (items, r + 1, right);
}

//
// Recursive implementation of binary search
// - returns the index in <items> of the entry with <surname>
// - max and min are the high and low indices of the range to search
//
static private int find (Entry[] items, String surname, int min, int max)
{
    if (min > max) {
        // Not present in array, exit indicating failure
        return (-1);
    } else {
        // Locate the midpoint and compare its key against the search key
        int mid = (int) Math.floor ((min + max) / 2);
        int cmp = items[mid].surnameCompare (surname);
        System.out.println ("Checking " + items[mid].surnameOf ());

        if (cmp == 0) {
            // Keys match, found the desired entry
            return (mid);
        } else if (cmp > 0) {
            // Discard the upper half of the range
            return (find (items, surname, min, mid - 1));
        } else {
            // Discard the lower half of the range
            return (find (items, surname, mid + 1, max));
        }
    }
} // end of class Phonebook

```

```

// Lab 9: Ordered Sets using linked-lists
// Sample Solution by Anthony Cox, Oct. 99

// A simple class for linked list elements
class Element
{
    public Element next;
    public char value;

    Element (char c, Element e)
    {
        value = c;
        next = e;
    }
}

// Class to represent an ordered set
// - implemented using a linked list
class Ordered
{
    // Instance variable to hold the linked list
    private Element list;

    // Constructor for an empty set
    Ordered ()
    {
        list = null;
    }

    // Constructor for a singleton set
    Ordered (char c)
    {
        list = new Element (c, null);
    }

    // Constructor for a populated set
    // - Assumes that elements of s are already in order!
    // - Ignores whitespace characters in the string
    Ordered (String s)
    {
        list = null;
        if (s != null && s.length () > 0) {
            for (int i = s.length () - 1; 0 <= i; i--) {
                if (s.charAt (i) != ' ') {
                    list = new Element (s.charAt (i), list);
                }
            }
        }
    }

    // Recursive Implementation, to demonstrate how the sets functionality
    // can be located in the Element class instead of the Ordered Class.
    public String toString ()
    {
        Element e = list; /* Current element in traversal */
        String s = "";
        while (e != null) {
            s = s + String.valueOf (e.value) + " ";
            e = e.next;
        }
        return (s.trim ());
    }

    // Add c to set, putting it in the correct location
    // - Mutates the original set by adding a new element
    public void add (char c)
    {
        Element e = list; /* Current element in traversal */
        Element p = list; /* Previous element in traversal */
        if (list == null) {
            // Case 1: Adding to the empty list
            list = new Element (c, null);
        } else if (c < list.value) {
            // Case 2: Adding to the front of the list
            list = new Element (c, list);
        } else {
            // Case 3: Adding anywhere else in the list
            while ((e != null) && (c > e.value)) {
                p = e;
                e = e.next;
            }
            e = new Element (c, e);
            p.next = e;
        }
    }

    // Is the set empty?
    public boolean isEmpty ()
    {
        return (list == null);
    }

    // Simple linear search of set
    // - returns true if c in set, false otherwise
    public boolean isMember (char c)
    {
        Element e = list;
        while (e != null) {
            if (e.value == c) {
                return (true);
            }
            e = e.next;
        }
        return (false);
    }

    // Return a copy of the set
    public Ordered copy ()
    {
        // This is why we filter out whitespace in the constructor
        return (new Ordered (toString ()));
    }

    // Return a new set describing the union of two sets
    // - note: since sets ordered more efficient implementation is possible
    public Ordered union (Ordered set)
    {
        Ordered result = set.copy ();
        Element e = list;
        while (e != null) {
            if (! result.isMember (e.value)) {
                result.add (e.value);
            }
            e = e.next;
        }
        return (result);
    }

    // Return a new set describing the intersection of two sets
    // - note: since sets ordered more efficient implementation is possible
    public Ordered intersection (Ordered set)
    {
        Ordered result = new Ordered ();
        Element e = list;
        while (e != null) {
            // Return a new set describing the intersection of two sets
            // - note: since sets ordered more efficient implementation is possible
            Element p = list; /* Previous element in traversal */
            if (list == null) {
                // Case 1: Adding to the empty list
                list = new Element (c, null);
            } else if (c < list.value) {
                // Case 2: Adding to the front of the list
                list = new Element (c, list);
            } else {
                // Case 3: Adding anywhere else in the list
                while ((e != null) && (c > e.value)) {
                    p = e;
                    e = e.next;
                }
                e = new Element (c, e);
                p.next = e;
            }
        }
    }
}

```

```
if (set.isMember (e.value)) {
    result.add (e.value);
}
e = e.next;
}
return (result);
}

// Return a new set describing the subtraction of <set> from this set
// - note: since sets ordered more efficient implementation is possible
public Ordered difference (Ordered set)
{
    Ordered result = new Ordered ();
    Element e = list;
    while (e != null) {
        if (!set.isMember (e.value)) {
            result.add (e.value);
        }
        e = e.next;
    }
    return (result);
}

public Element getList()
{
    return list;
}

// Not part of the marked implementation
public static void main (String[] args)
{
    Ordered o1 = new Ordered ("abcdefg");
    Ordered o2 = new Ordered ("hijklm");
    Ordered o3 = new Ordered ("acegikm");
    System.out.println ("Set1: " + o1);
    System.out.println ("Set2: " + o2);
    System.out.println ("Set3: " + o3);
    o2.add ('a');
    System.out.println ("Set2 + a: " + o2);
    o2.add ('z');
    System.out.println ("Set2 + z: " + o2);
    o2.add ('x');
    System.out.println ("Set2 + x: " + o2);
    System.out.println ("Copy of Set3: " + o3.copy ());
    System.out.println ("o1 union o2: " + o1.union (o2));
    System.out.println ("o1 union o3: " + o1.union (o3));
    System.out.println ("o1 isect o2: " + o1.intersection (o2));
    System.out.println ("o1 isect o3: " + o1.intersection (o3));
}
}
```