```
.root {
    -fx-font-family: monospace;
    -fx-font-size: 11pt;
    -fx-background-color:  blue;
}

.label {
    -fx-text-fill: black;
}

.text-area {
    -fx-background-color:  green;
}
```

30/30 points for Program 5

As always, an “above-and-beyond” solution.
Well done.

```
/*******************************************************************************************
 *  This class handles the processing of a time object
 *
 *  It stores the fields of hours, minutes and seconds and also will format the the date in several ways
 *  It allows comparisons between times
 *  --Corrected bug causing hour to display 0 in 12 hour time setting
 *
 *  CST 283 Programming Assignment 1
 *  @author Michael Clinesmith
 *******************************************************************************************/

import java.util.NoSuchElementException;
import java.util.StringTokenizer;

public class Times
{
    private int hour, minute;
    private double second;

    /**
     * No-argument constructor to create a time object
     * Sets time to 00:00:00.0
     */

    public Times()
    {
        hour = 0;
        minute = 0;
        second = 0.0;
    }

    /**
     * Constructor to create a time object given hours, minutes and seconds
     * @param h int: the hour for the time
     * @param m int: the minutes for the time
     * @param s double: the seconds for the time (including fractional part)
     */
    public Times(int h, int m, double s)
    {
        hour = h;
        minute = m;
        second = s;
    }

    /**
     * Constructor to create a time object given hours, minutes and seconds
     * @param h int: the hour for the time
     * @param m int: the minutes for the time
     * @param s int: the seconds for the time
     */
    public Times(int h, int m, int s)
    {
        hour = h;
        minute = m;
        second = s;
    }

    /**
     * Constructor to create a time object from a string in the format of HH:MM:SS.SSS
     * @param stringTime String: the time in the format of HH:MM:SS.SSS
```

I liked this class for the time stamp management. Definitely one that can be re-used another day.

```
 */
public Times(String stringTime)
{
    StringTokenizer timeTokens;
    try
    {
        timeTokens = new StringTokenizer( stringTime, ":" );
        hour = Integer.parseInt( timeTokens.nextToken() );
        minute = Integer.parseInt( timeTokens.nextToken() );
        second = Double.parseDouble( timeTokens.nextToken() );
    }
    catch (NumberFormatException | NoSuchElementException e)    // if exceptions in running out of tokens or bad format
    {
        // set to default time
        hour = 0;
        minute = 0;
        second = 0.0;
    }
}

/**
 * Constructor that will take a Times object and make a deep copy of it
 * @param times
 */
public Times(Times times)
{
    hour = times.getHour();
    minute = times.getMinute();
    second = times.getSecond();
}

/**
 * Accessor method to get the hours
 * @return int: hour in the time object
 */
public int getHour()
{
    return hour;
}

/**
 * Accessor method to get the minutes
 * @return int: minutes in the time object
 */
public int getMinute()
{
    return minute;
}

/**
 * Accessor method to get the seconds
 * @return double: seconds in the time object, including decimal part
 */
public double getSecond()
{
    return second;
}

/**
 * Method to get the seconds but only the integer part
```

```
 * @return int: the seconds in the time object (truncated)
 */
public int getSecondInt()
{
    return (int) (second);
}

/**
 * Method to get the hour using standard 12 hour format 12:00 AM - 11:59 PM
 * @return int: the hour going by standard 12 hour format (1 to 12), returns -1 if invalid
 */
public int getHour12()
{
    int hr = -1;
    if(isValid())
    {
        hr = (hour+11) % 12 + 1;        // corrected bug
    }
    return hr;
}

/**
 * Method to get if the time is AM or PM using standard 12 hour format
 * @return String: "AM" or "PM", or "NA" if invalid
 */
public String getAMPM()
{
    String AMPM="NA";

    if (isValid())
    {
        if (hour<12)
        {
            AMPM = "AM";
        }
        else
        {
            AMPM = "PM";
        }
    }

    return AMPM;
}

/**
 * Mutator method to set the hours in the time object
 * @param hour int: the hour to set in the time object
 */
public void setHour( int hour )
{
    this.hour = hour;
}

/**
 * Mutator method to set the minutes in the time object
 * @param minute int: the minutes to set in the time object
 */
public void setMinute( int minute )
{
    this.minute = minute;
```

```
}

/**
 * Mutator method to set the seconds in the time object
 * @param second double: the seconds to set in the time object (can include decimal part)
 */
public void setSecond( double second )
{
    this.second = second;
}

/**
 * Method to determine if the values stored in the time object are valid
 * @return boolean: true if the values in the time object are all valid, false otherwise
 */
public boolean isValid()
{
    boolean valid = true;

    if ( hour<0 || hour>23 || minute<0 || minute>59 || second<0.0 || second>=60.0)
    {
        valid = false;
    }
    return valid;
}

/**
 * Method to return the values in the time object as a string object in
 * @return String: the time in the format hH:MM:SS XM with the h optional, "Invalid time" if the values were not valid
 */
@Override
public String toString()
{
    String str = "Invalid time";
    if (isValid())
    {
        str = "" + getHour12() + ":" + MinutetoStringNR() + ":" + SecondtoStringNR() + " " + getAMPM();
    }
    return str;
}

/**
 * Method to return the values in the time object as a string object in the form requested by the instructor
 * @return String: the time in the format HHMMz
 */
public String toModString()
{
    String str = HourtoString() + MinutetoString() + "z";
    return str;
}

/**
 * Method to return true if when rounding to the nearest second, the minutes should be rounded up
 * @return boolean: true if seconds are being rounded from 59 to 00, false if not
 */
public boolean roundMinuteUp()
{
    boolean roundUp = false;
    if (second >= 59.5)
    {
```

```
        roundUp = true;
    }
    return roundUp;
}

/**
 * Method to return true if when rounding to the nearest second, the hours should be rounded up
 * @return boolean: true if rounding minutes and seconds from 59:59 to 00:00, false if not
 */
public boolean roundHourUp()
{
    boolean roundUp = false;
    if (second >= 59.5 && minute == 59)
    {
        roundUp = true;
    }
    return roundUp;
}

/**
 * Method to return true if when rounding to the nearest second, the day should be rounded up
 * @return boolean: true if rounding hours, minutes and seconds from 23:59:59 to 00:00:00, false if not
 */
public boolean roundDayUp()
{
    boolean roundUp = false;
    if (second >= 59.5 && minute == 59 && hour == 23)
    {
        roundUp = true;
    }
    return roundUp;
}

/**
 * Method to convert minutes to a String to assist in printing the time in the format requested by the instructor
 * It rounds to the nearest minute, with the exception that it will not round 23:59 to 00:00, causing the
 * date to switch
 * @return String: A string representing the minutes in the time, rounded if necessary; NA is returned if time invalid
 */
public String MinutetoString()
{
    String str = "NA";                  // code for invalid time
    int tempMin = minute;
    if (isValid())
    {
        if( second>=30.0 )             // check if closer to next minute
        {
            tempMin++;
        }

        if ( tempMin==60 )              // check if rounding caused overflow to next hour
        {
            tempMin = 0;
            if (hour==23)
            {
                tempMin = 59;           // do not roll over minute if it would cause day to roll over
            }

        }
```

```
        if ( tempMin>9 )                // minutes is two digits
        {
            str = Integer.toString( tempMin );
        }
        else                            // add "0" to beginning of minutes
        {
            str = "0" + Integer.toString( tempMin );
        }
    }
    return str;
}

/**
 * Method to convert minutes to a String
 * This method does No Rounding - NR
 * @return String: A string representing the minutes in the time, NA is returned if time invalid
 */
public String MinutetoStringNR()
{
    String str = "NA";                  // code for invalid time
    if (isValid())
    {
        if ( minute>9 )                // minutes is two digits
        {
            str = Integer.toString( minute );
        }
        else                            // add "0" to beginning of minutes
        {
            str = "0" + Integer.toString( minute );
        }
    }
    return str;
}
/**
 * Method to convert seconds to a String
 * This method does No Rounding - NR
 * @return String: A string representing the seconds in the time, NA is returned if time invalid
 */
public String SecondtoStringNR()
{
    String str = "NA";                  // code for invalid time
    int sec = getSecondInt();           // get integer value for second
    if (isValid())
    {
        if ( sec>9 )                    // sec is two digits
        {
            str = Integer.toString( sec );
        }
        else                            // sec is one digit so add "0" to beginning of sec
        {
            str = "0" + Integer.toString( sec );
        }
    }
    return str;
}


/**
 * Method to convert hours to a String to assist in printing the time in the format requested by the instructor
 * It rounds to the nearest minute, with the exception that it will not round 23:59 to 00:00, causing the
```

```
 * date to switch
 * @return String: A string representing the hours in the time, rounded if necessary; NA is returned if time invalid
 */
public String HourtoString()
{
    String str = "NA";                  // code for invalid time
    int tempHours = hour;
    if (isValid())
    {
        // check if needs to round to next hour, but do not round if it will round to next day
        if( minute == 59 && second>=30.0 && hour != 23 )
        {
            tempHours++;
        }

        if ( tempHours>9 )                  // minutes is two digits
        {
            str = Integer.toString( tempHours );
        }
        else                            // add "0" to minutes
        {
            str = "0" + Integer.toString( tempHours );
        }
    }
    return str;
}

/**
 * Method to check if two Times objects are equal
 * @param time Times: A Times object that is being compared to this object
 * @return boolean: true if the times are the same, false if not
 */
public boolean isEqual(Times time)
{
    boolean equal = false;
    if (hour == time.getHour() && minute == time.getMinute() && second == time.getSecond())
    {
        equal = true;
    }
    return equal;
}

/**
 * Method to compare two Times objects
 * @param time Times: A Times object that is being compared to this object
 * @return int: returns 1 if this object comes after the Times object parameter, -1 if it comes before, and 0 if they are equal
 */
public int compareTo(Times time)
{
    int compare = 0;                            // if none of if statements apply, then equal

    // set to 1 if this object is later, set to -1 if parameter object is later
    if (hour > time.getHour())                // compare hour first
    {
        compare = 1;
    }
    else if ( hour < time.getHour())
    {
        compare = -1;
    }
```

```
        else if( minute > time.getMinute())       // if hours the same check minutes
        {
            compare = 1;
        }
        else if ( minute < time.getMinute())
        {
            compare = -1;
        }
        else if ( second > time.getSecond())      // if hours and minutes the same, check seconds
        {
            compare = 1;
        }
        else if (second < time.getSecond())
        {
            compare = -1;
        }
        return compare;
    }

}
```

```
/*******************************************************************************************
 *  This class handles the processing of a date object
 *
 *  It stores the fields of a month, day and year and also will format the the date is several ways
 *  It allows comparisons between dates
 *
 *  CST 283 Programming Assignment 1
 *  @author Michael Clinesmith
 *******************************************************************************************/

import java.util.NoSuchElementException;
import java.util.StringTokenizer;

public class Dates
{
    // date fields
    private int month, day, year;

    // String to save the month codes
    private final String[] MONTH_CODE = {"NA ", "JAN", "FEB", "MAR", "APR", "MAY", "JUN", "JUL",
            "AUG", "SEP", "OCT", "NOV", "DEC" };
    private final String[] MONTH = {"Not Valid", "January", "February", "March", "April", "May", "June", "July",
            "August", "September", "October", "November", "December"};
    /**
     * No-argument Constructor for a date object
     * Sets the date to be January 1, 1900
     */
    public Dates()
    {
        month = 1;
        day = 1;
        year = 1900;
    }

    /**
     * Constructor to set the date of a date object
     * @param m int: value representing the month
     * @param d int: value representing the day
     * @param y int: value representing the year
     */
    public Dates(int m, int d, int y)
    {
        month = m;
        day = d;
        year = y;
    }

    /**
     * Constructor that stores a date of the format YYYY-MM-DD
     * @param stringDate
     */
    public Dates(String stringDate)
    {
        stringDate = stringDate.trim();             // eliminates any extra whitespace
        StringTokenizer dateTokens;
        try
        {
            dateTokens = new StringTokenizer( stringDate, "-" );
            year = Integer.parseInt( dateTokens.nextToken() );
            month = Integer.parseInt( dateTokens.nextToken() );
```

```
            day = Integer.parseInt( dateTokens.nextToken() );
        }
        catch (NumberFormatException | NoSuchElementException e)
        {
            year = 1900;
            month = 1;
            day = 1;
        }
        // System.out.println( toModString() );

    }

    /**
     * Constructor that takes a Date object and makes a deep copy of it
     * @param dates
     */
    public Dates ( Dates dates)
    {
        year = dates.getYear();
        month = dates.getMonth();
        day = dates.getDay();
    }

    /**
     * Accessor method to get the date of a date object
     * @return int: value representing the day
     */
    public int getDay()
    {
        return day;
    }

    /**
     * Accessor method to get the month of a date object
     * @return int: value representing the month
     */
    public int getMonth()
    {
        return month;
    }

    /**
     * Accessor method to get the year of a date object
     * @return int: value representing the year
     */
    public int getYear()
    {
        return year;
    }

    /**
     * Method to get the 3-letter string for a month
     * @return String: a three letter code representing the month stored in date object
     */
    public String getMonthCode()
    {
        String code = MONTH_CODE[0];            // default invalid month code
        if ( month>0 && month < 13 )            // if month valid get code
        {
            code = MONTH_CODE[month];
```

```
    }
    return code;
}


/**
 * Method to get the 3-letter string for a month
 * @return String: a three letter code representing the month stored in date object
 */
public String getMonthName()
{
    String code = MONTH[0];            // default invalid month code
    if ( month>0 && month < 13 )           // if month valid get code
    {
        code = MONTH[month];
    }
    return code;
}

/**
 * Mutator method to set the date of a date object
 * @param day int: a value to set the day of the object
 */
public void setDay( int day )
{
    this.day = day;
}
/**
 * Mutator method to set the month of a date object
 * @param month int: a value to set the month of the object
 */

public void setMonth( int month )
{
    this.month = month;
}
/**
 * Mutator method to set the year of a date object
 * @param year int: a value to set the year of the object
 */

public void setYear( int year )
{
    this.year = year;
}

/**
 * Method checks to see if a saved day is a valid day
 *
 * @return boolean: true if the day is valid, false if it is not
 */
public boolean isValid()
{
    boolean valid = true;
    if ( year < 1)                      // must be a valid AD year
    {
        valid = false;
    }
    else if( month<1 || month>12 )      // must be a valid month
    {
```

```
        valid = false;
    }
    else if( day<1 || day>31 )          // days must be from 1 to 31
    {
        valid = false;
    }
    else if ( day == 31 && (month == 2 || month == 4 || month == 6 || month == 9 || month == 11)) // day cannot be 31 for these months
    {
        valid = false;
    }
    else if (day == 30 && month == 2)   // day cannot be 30 in February
    {
        valid = false;
    }
    else if (day == 29 && month ==2 && !isLeapYear() )  // day cannot be 29 in February if it is not a leap year
    {
        valid = false;
    }

    return valid;
}

/**
 * Method checks if a year is a leap year (for years greater than 0)
 * if the year is greater than 0
 * is divisible by 4
 * and is not divisible by 100 unless it is also divisible by 400
 * then they year is a leap year
 *
 * @return boolean: true if the year is a leap year, false otherwise
 */
public boolean isLeapYear()
{
    boolean leapYear = false;

    if ( year > 0 && year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
    {
        leapYear = true;
    }
    return leapYear;
}

/**
 * Method to convert a date object to a string
 * @return String: a string representing the date object
 */
@Override
public String toString()
{
    String string = getMonthName() + " " + getDay() + ", " + getYear();

    return string;
}

/**
 * Method to convert a date object to a string in the format requested by the instructor
 * @return String: a string representing the date object in the form DD MMM YY
 */
public String toModString()
{
```

```
    String string = dayToString() + " " + getMonthCode() + " " + yearToString();

    return string;
}

/**
 * Method to convert a day to a two character String
 * @return String: a two character String for the day or "NA" if the date was not volid
 */
public String dayToString()
{
    String string="NA";
    if (isValid())                              // if date object is valid
    {
        if ( day > 9 )                          // if 2 digits in day convert to string
        {
            string = Integer.toString( day );

        }
        else                                    // add 0 in front of day
        {
            string = "0" + Integer.toString( day );
        }
    }
    return string;
}

/**
 * Method to convert a year to a two character String
 * @return String: a two character String for the year or "NA" if the date was not valid
 */
public String yearToString()
{
    String string="NA";
    int shortYear = year % 100;             // shortYear is from 0 to 99

    if (isValid())
    {
        if (shortYear<10)                   // shortYear has only 1 digit
        {
            string = "0" + Integer.toString( shortYear );
        }
        else                                // shortYear has 2 digits
        {
            string = Integer.toString( shortYear );
        }
    }

    return string;
}

/**
 * Method to check if two Dates objects are equal
 * @param date Dates: A Dates object that is being compared to this object
 * @return boolean: true if the dates are the same, false if not
 */
public boolean isEqual(Dates date)
{
    boolean equal = false;
    if (year == date.getYear() && month == date.getMonth() && day == date.getDay())
```

```
        {
            equal = true;
        }
        return equal;
    }

    /**
     * Method to compare two Dates objects
     * @param date Dates: A Dates object that is being compared to this object
     * @return int: returns 1 if this object comes after the Dates object parameter, -1 if it comes before, and 0 if they are equal
     */
    public int compareTo(Dates date)
    {
        int compare = 0;                        // if none of if statements apply, then equal

        // set to 1 if this object is later, set to -1 if parameter object is later
        if (year > date.getYear())              // compare year first
        {
            compare = 1;
        }
        else if ( year < date.getYear())
        {
            compare = -1;
        }
        else if( month > date.getMonth())       // if year the same check month
        {
            compare = 1;
        }
        else if ( month < date.getMonth())
        {
            compare = -1;
        }
        else if ( day > date.getDay())          // if year and month the same, check day
        {
            compare = 1;
        }
        else if (day < date.getDay())
        {
            compare = -1;
        }
        return compare;
    }

}
```

```
/*********************************************************************************************
 *  This class stores an array of county FIPS code, state and population data
 *
 *  It allows storage of state and national jurisdiction codes and lists of subjurisdictions
 *  of those entities
 *
 *  CST 283 Programming Assignment 5
 *  Modified from CountyData.java - CST 183 Programming Assignment 7
 *  @author Michael Clinesmith
 *********************************************************************************************/

import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;

import java.util.Optional;
import java.util.Scanner;
import java.io.*;
import java.util.StringTokenizer;

public class CountyList    √
{
    // constants usable by entire class
    private final static int NUM_OF_COUNTIES = 5000;

    private int numOfElements=0;
    private County[] countyList = new County[NUM_OF_COUNTIES];    √

    /**
     * No argument constructor
     */
    public CountyList()
    {
        numOfElements=0;
    }

    /**
     * Constructor loads County FIPS code and populations data from codeFilename and popFilename
     * @param codeFilename String: The file that has County FIPS code and name information
     * @param popFilename String: The file that has County FIPS code and population information
     */
    public CountyList(String codeFilename, String popFilename)
    {
        String message;
        File codeData;                // file that holds the population data
        Scanner inputFile;                  // used to get data from file
        int i = 0, j = 0, numElems = 0;
        String inputLine;                   // String used to get a line of file input

        setupListWithCodeData(codeFilename);

        setupListWithPopData(popFilename);

        calculateSuperPopData();

    }

    /**
     * Accessor method that gets the number of elements in the CountyArray
     * @return int: The number of elements in the County Array
     */
```

```
public int getNumOfElements()
{
    return numOfElements;
}

/**
 * Method that gets the index position of the country record with the given code
 * @param code String: A String code representing a county jursidiction
 * @return int: The index position of the county, or -1 if not found
 */
public int getIndex(String code)
{
    boolean isFound = false;
    int index = -1;
    for (int i=0; i<numOfElements && !isFound; i++)    // search array and exit if element found
    {
        if (countyList[i].getFIPScode().equals( code ))
        {
            index = i;
            isFound = true;
        }
    }
    return index;
}

/**
 * Method that get the name of a county with a given FIPS code
 * @param code String: A FIPS code for a county
 * @return String: The name of the county, or Not Found if not found
 */
public String getName(String code)
{
    String name = "Not Found";
    int index = getIndex( code );
    if (index>=0)
    {
        name = countyList[index].getCountyName();
    }
    return name;
}

/**
 * Method that gets the state abbreviation for a county
 * @param code String: The FIPS code of a county
 * @return String: The State abbreviation for a county or "Not Found" if not found
 */
public String getState(String code)
{
    String state = "Not Found";
    int index = getIndex( code );
    if (index>=0)
    {
        state = countyList[index].getStateCode();
    }
    return state;
}

/**
 * Method that gets the population for a county given its FIPS code
 * @param code String: The FIPS code of a county
```

```
 * @return int: The county's population, or -1 if it was not found
 */
public int getPopulation(String code)
{
    int pop = -1;
    int index = getIndex( code );
    if (index>=0)
    {
        pop = countyList[index].getPopulation();
    }
    return pop;
}

/**
 * Method that begins the creation of the county list, loading FIPS code and name information
 *
 * THE USER MAY END THE PROGRAM IN THE METHOD IF THERE IS PROBLEMS LOADING THE DATA
 *
 * @param filename String: A file that contains the FIPS code and name information for counties
 */
private void setupListWithCodeData(String filename)
{
    File codeData;
    String message;
    Scanner inputFile;
    int i=0;
    int commaLoc = 0;
    int USPos = 0;
    int currentStatePos = 0;

    String inputLine;

    try
    {
        String fips, name, state;                    // Work variables

        // Build list of county objects
        codeData = new File(filename);

        if(!codeData.exists())  // file not found
        {
            message = "The file " + filename + " containing fips data was not found.\n" +
                    "Do you want to end the program?";

            quitOption( message );  // give user option to exit program
            return;                 // end method since file not found
        }

        inputFile = new Scanner(codeData);

        // Read input file while more data exist
        // Read one line at a time (assuming each line contains one username)
        i = 0;          // used to work through array elements
        while (inputFile.hasNext())
        {
            inputLine = inputFile.nextLine();

            // Read all data on one line
            fips      = inputLine.substring( 0, 5 );
            commaLoc = inputLine.indexOf( "," );
```

```
            if (commaLoc>6)
            {
                name = inputLine.substring( 6, commaLoc);
                state = inputLine.substring( commaLoc+2 );
            }
            else
            {
                name = inputLine.substring( 6 );
                state = "NA";
            }

            countyList[i] = new County(fips,name,state, 0);

            // add county or state to appropriate superjursidiction
            if (commaLoc<0)     // new State
            {
                countyList[i].setStateEntity( true );
                if (!name.equals( "00000" )) // a State, but not UNITED STATES
                {
                    countyList[USPos].addToSubEntityList( fips ); // sub to US
                    currentStatePos = i;                          // new State, set position marker
                }
            }
            else            // County within a state
            {
                countyList[currentStatePos].addToSubEntityList( fips );
            }

            i++;
        }
        numOfElements = i;    // Capture number of elements

        inputFile.close();
    }
    catch (IOException e)  // if error loading data, give error message and end program
    {
        message = "There was an error processing the file " + filename + ".\n" +
                "Do you wish to end the program?";

        quitOption( message );      // give user option to quit
        return;                     // error processing, so return without displaying confirmation message
    }

    message = "The jurisdiction data from the file " + filename +
            "\nis now uploaded into memory.";

    Alert alert = new Alert( Alert.AlertType.INFORMATION );
    alert.setTitle( "DATA LOADED" );
    alert.setContentText( message );
    alert.showAndWait();

    return;
}

/**
 * Method that populates the county list with population data
 *
 * THE USER MAY END THE PROGRAM IN THE METHOD IF THERE IS PROBLEMS LOADING THE DATA
 *
 * @param filename String: A file that contains the FIPS code and name information for counties
```

```
 */
private void setupListWithPopData(String filename)
{
    File popData;
    String message;
    Scanner inputFile;
    String inputLine;
    int pop=0;
    int index;

    try
    {
        String fips, name, state;                    // Work variables

        StringTokenizer lineTokens;     //  used to get tokens from data input

        // Build list of county objects
        popData = new File(filename);

        if(!popData.exists())  // file not found
        {
            message = "The file " + filename + " containing fips data was not found.\n" +
                    "Do you wish to end the program?";

            quitOption( message );  // give user option to quit
            return;                 // exit method since file was not found
        }

        inputFile = new Scanner(popData);

        // Read input file while more data exist
        // Read one line at a time (assuming each line contains one username)
        index = 0;          // used to work through array elements
        while (inputFile.hasNext())
        {
            inputLine = inputFile.nextLine();

            // Read all data on one line
            fips      = inputLine.substring( 0, 5 );
            try
            {
                pop = Integer.parseInt( inputLine.substring( 6 ));
                index = getIndex(fips);

                countyList[index].setPopulation( pop );
            }
            catch (NumberFormatException ex)
            {
                message = "There was an error reading a population data record.\n" +
                        "Do you wish to exit this program?  If not, this record will be skipped.";

                quitOption( message );  // give user option to quit
            }
            catch (ArrayIndexOutOfBoundsException ex)
            {
                message = "A data record was found but that record was not in the county file.\n" +
                        "Do you wish to exit this program?  If not, this record will be skipped.";

                quitOption( message );  // give user option to quit
            }
```

```
        }

        inputFile.close();
    }
    catch (IOException e)  // if error loading data, give error message and end program
    {
        message = "There was an error processing the file " + filename + ".\n" +
                "Do you wish to exit this program?";

        quitOption( message );  // give user option to quit
        return;                 // exit method without confirmation message
    }

    message = "The population data from the file " + filename +
            "\nis now uploaded into memory.";

    Alert alert = new Alert( Alert.AlertType.INFORMATION );
    alert.setTitle( "DATA LOADED" );
    alert.setContentText( message );
    alert.showAndWait();

    return;
}

/**
 * Method that fills in the population values for all superjurisdictions (above the counties)
 */
private void calculateSuperPopData()
{
    String[] list;
    int listLength;
    int USIndex = getIndex( "00000" );
    int pop;

    for(int i = 0; i<numOfElements; i++)
    {
        if(countyList[i].isStateEntity() && i !=USIndex) // skip US to the end
        {
            list = countyList[i].getSubEntityList();
            listLength = countyList[i].getSubEntityCount();
            pop = 0;

            for(int j=0; j<listLength; j++)
            {
                // gets the population of the county with code list[j] and adds it to the total
                pop += countyList[getIndex( list[j] )].getPopulation();
            }
            countyList[i].setPopulation( pop );
        }
    }
    if (USIndex>=0)     // US index exists
    {
        // calculate US pop
        list = countyList[USIndex].getSubEntityList();
        listLength = countyList[USIndex].getSubEntityCount();
        pop = 0;
        for (int j = 0; j < listLength; j++)
        {
            // gets the population of the county with code list[j] and adds it to the total
```

```
                pop += countyList[getIndex( list[j] )].getPopulation();
            }
            countyList[USIndex].setPopulation( pop );
        }
    }

    /**
     * Method that asks if the user wants to quit because of an error in processing
     * @param message String: The question to ask the user
     *
     * THIS METHOD WILL END THE PROGRAM IF THE USER SELECTS OK
     */
    private void quitOption(String message)
    {
        Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
        alert.setTitle( "Quit?" );
        alert.setContentText( message );
        Optional<ButtonType> result = alert.showAndWait();
        if (result.get() == ButtonType.OK)
        {
            System.exit( 0 );
        }
        return;
    }

}
```

```
/*********************************************************************************************
 *  This class stores county FIPS code, state and population data.
 *
 *  It allows storage of state and national jurisdiction codes, names, populations and lists of subjurisdictions
 *  of those entities (for states and national codes)
 *
 *  CST 283 Programming Assignment 5
 *  Modified from CountyData.java - CST 183 Programming Assignment 7
 *  @author Michael Clinesmith
 *********************************************************************************************/

public class County
{
    // class fields
    private String FIPScode;
    private String countyName, stateCode;

    private int population;
    private boolean stateEntity;
    private String[] subEntityList;
    private int subEntityCount;

    /**
     * No parameter constructor
     */
    public County()
    {
        FIPScode = "99999";
        countyName = "None";
        stateCode = "NA";
        population = 0;
        stateEntity = false;
        subEntityList = null;
        subEntityCount = 0;
    }

    /**
     * Constructor with parameters
     * @param code String: County FIPS code
     * @param county String: County name
     * @param state String: State code
     * @param popData int array: contains population data
     */
    public County(String code, String county, String state, int popData)
    {
        FIPScode = code;
        countyName = county;
        stateCode = state;
        population = popData;
        stateEntity = false;
        subEntityList = null;
        subEntityCount = 0;
    }

    /**
     * County copy constructor
     * @param county County: A County object to make a deep copy of
     */
    public County(County county)
    {
```

```
    FIPScode = county.getFIPScode();
    countyName = county.getCountyName();
    stateCode = county.getStateCode();
    population = county.getPopulation();
    stateEntity = county.isStateEntity();
    subEntityList = county.getSubEntityList();
    subEntityCount = county.getSubEntityCount();
}

/**
 * Mutator method to set the FIPS code
 * @param FIPScode String: FIPS code of a county
 */
public void setFIPScode(String FIPScode)
{
    this.FIPScode = FIPScode;
}

/**
 * Mutator method to set the county name
 * @param countyName String: A county name
 */
public void setCountyName(String countyName)
{
    this.countyName = countyName;
}

/**
 * Mutator method to set the state code
 * @param stateCode String: A state code representing the state a county is in
 */
public void setStateCode(String stateCode)
{
    this.stateCode = stateCode;
}

/**
 * Mutator method to set the county population
 * @param pop int: The population in a particular year
 */
public void setPopulation(  int pop)
{
    population = pop;
}

/**
 * Mutator method to set the StateEntity status
 * @param status boolean: A true or false value used to set stateEntity
 */
public void setStateEntity( boolean status)
{
    stateEntity = status;
}

/**
 * Mutator method to set a subEntityList
 * Setting a non null list will set the stateEntity status to true if neither stateEntity or nationEntity are true
 *
 * @param list String[]: A list used to set a state's or nation's subjurisdictions - it may be null
 * @param count int: The number of elements in the list
```

```
 */
public void setSubEntityList( String[] list, int count)
{
    if( count == 0)     // no elements in list - clue from user to remove it
    {
        subEntityList=null;
    }
    else
    {
        if (subEntityList == null || subEntityList.length < count)  // need to create a suitable sized list
        {
            if (count < 100)
            {
                subEntityList = new String[100];
            } else
            {
                subEntityList = new String[count + 50];
            }
        }

        for (int i = 0; i < count; i++)                             // create the list in the object
        {
            subEntityList[i] = list[i];
        }
        subEntityCount = count;
        if (!stateEntity)      // since a list now exists, set state flag
        {
            stateEntity = true;
        }
    }
}

/**
 * Method that adds a code to a jurisdiction's subEntityList
 * @param code String: A String representing a jurisdiction code
 */
public void addToSubEntityList(String code)
{

    if( subEntityCount==0 || subEntityCount==subEntityList.length)      // need to make a new list
    {
        String[] newList = new String[subEntityCount+20];
        // transfer elements to new list
        for(int i=0; i<subEntityCount; i++)
        {
            newList[i] = subEntityList[i];
        }
        newList[subEntityCount] = code;                                 // add new code
        subEntityCount++;
        subEntityList = newList;                                        // save new list as subEntityList
    }
    else
    {
        subEntityList[subEntityCount] = code;                           // add new code
        subEntityCount++;
    }
}

/**
 * Method that subtracts or removes a code from a jurisdiction's SubEntityList
```

```
 * @param code String: The code to remove from the list
 * @return boolean: true if the code was removed, false if not (i.e. it was not in the list)
 */
public boolean subtractFromSubEntityList(String code)
{
    boolean codeFound=false;
    for(int i=0; i<subEntityCount && !codeFound; i++)
    {
        if (subEntityList[i].equals( code ))
        {
            codeFound=true;
            subEntityCount--;
            subEntityList[i]=subEntityList[subEntityCount];
        }
    }
    return codeFound;
}

/**
 * Accessor method to get a county's FIPS code
 * @return String: The FIPS code of a county
 */
public String getFIPScode()
{
    return FIPScode;
}

/**
 * Accessor method to get a county's name
 * @return String: The name of a county
 */
public String getCountyName()
{
    return countyName;
}

/**
 * Accessor method to get a county's state code
 * @return String: The state code of a county
 */
public String getStateCode()
{
    return stateCode;
}

/**
 * Accessor method to get a county's population
 * @return int: the county's population in that year
 */
public int getPopulation()
{
    return population;
}

/**
 * Accessor method to get it an object has been set as a stateEntity
 * @return boolean: true if it is a stateEntity, false if not
 */
public boolean isStateEntity()
{
```

```
        return stateEntity;
    }

    /**
     * Accessor method to get the subEntityList
     * @return String[]: The subEntityList array or null if none exists
     */
    public String[] getSubEntityList()
    {
        String[] list = null;
        if(subEntityCount>0)
        {
            list = subEntityList.clone();
        }
        return list;
    }

    /**
     * Accessor method to get the number of elements in a subEntityList
     * @return int: The number of elements in the subEntityList
     */
    public int getSubEntityCount()
    {
        return subEntityCount;
    }

    /**
     * Method to return a string representing the data stored in a CountyData obuect
     * @return String: Contains the FIPS code, county name, state code, and population data of a CountyData object
     */
    @Override
    public String toString()
    {
        String data;

        if(stateEntity)
        {
            data = "FIPS Code: " + FIPScode +
                    "\nState Name: " + countyName +
                    "\nSubjurisdiction IDs:";
            for (int i=0; i<subEntityCount; i++)
            {
                data += "\n" + subEntityList[i];
            }
        }
        else
        {
            data = "FIPS Code: " + FIPScode +
                    "\nCounty Name: " + countyName +
                    "\nState: " + stateCode +
                    "\nPopulation: " + population;
        }
        return data;
    }

}
```

```java
/*********************************************************************************************
 *  This class stores information regarding security and weather alerts.
 *
 *  It also loads alert definitions into memory
 *
 *  It will create an alert message using the toString method which also uses data contained in
 *  a CountyList object
 *
 *  CST 283 Programming Assignment 5
 *  Modified from CountyData.java - CST 183 Programming Assignment 7
 *  @author Michael Clinesmith
 *********************************************************************************************/

import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import java.io.File;
import java.io.IOException;
import java.util.Optional;
import java.util.Scanner;

public class Alerts     √
{

    // static fields concerning alert meaning
    private static final int MAX_CODES=50, MAX_SECURITY=10;
    private static final String ALERT_FILE = "warningList.txt";  // This file should have the alert definitions in it
    private static int numCodes, numSeverity, numSecurity;
    private static String[] weatherWarningCodes = new String[MAX_CODES];            // 2 letter weather codes
    private static String[] weatherWarningType = new String[MAX_CODES];             // definition of those codes
    private static String[] weatherWarningSeverity = new String[MAX_SECURITY];      // 1 letter weather severity code
    private static String[] weatherWarningSeverityType = new String[MAX_SECURITY];  // definition of those codes
    private static String[] securityWarningCode = new String[MAX_SECURITY];         // color national security code
    private static String[] securityWarningType = new String[MAX_SECURITY];         // definition of those codes


    private static boolean isLoaded=false;      // used to indicate codes needing to be loaded (so only happens once)
    private static CountyList countyList = null; // used to allow access to the county list (and only save once)

    // single alert fields
    private String FIPSCode, warningCode;
    private Dates startDate, endDate;
    private Times startTime, endTime;

    /**
     * No argument constructor, sets default values and loads the codes if necessary
     */
    public Alerts()
    {
        FIPSCode="99999";
        warningCode="---";
        startDate = new Dates();
        startTime = new Times();
        endDate = new Dates();
        endTime = new Times();
        if(!isLoaded)
        {
            loadCodes();
        }
    }
```

```
/**
 * Constructor with arguments
 * @param fCode String: FIPS codes representing a county (or state or US)
 * @param startDateTime String: A String representing the starting date and time
 * @param endDateTime String: A String representing the ending date and time
 * @param wCode String: The warning code
 */
public Alerts(String fCode, String startDateTime, String endDateTime, String wCode )
{

    FIPSCode = fCode;
    warningCode = wCode;
    try
    {
        // parse out parts of string from format YYYYMMDD to MM, DD, YYYY and convert to ints
        startDate = new Dates(Integer.parseInt( startDateTime.substring( 4,6 )),
                        Integer.parseInt( startDateTime.substring( 6, 8 )),
                        Integer.parseInt( startDateTime.substring( 0, 4 )));

        // parse out parts of string from format HHMM to HH, MM, SS and convert to ints
        startTime = new Times(Integer.parseInt( startDateTime.substring( 8, 10) ),
                        Integer.parseInt( startDateTime.substring( 10, 12 ) ),
                        0);

        // parse out parts of string from format YYYYMMDD to MM, DD, YYYY and convert to ints
        endDate = new Dates(Integer.parseInt( endDateTime.substring( 4,6 )),
                Integer.parseInt( endDateTime.substring( 6, 8 )),
                Integer.parseInt( endDateTime.substring( 0, 4 )));

        // parse out parts of string from format HHMM to HH, MM, SS=0 and convert to ints
        endTime = new Times(Integer.parseInt( endDateTime.substring( 8, 10) ),
                Integer.parseInt( endDateTime.substring( 10, 12 ) ),
                0);

    }
    catch(NumberFormatException | StringIndexOutOfBoundsException e ) // give default values if error in processing
    {
        startDate = new Dates();
        startTime = new Times();
        endDate = new Dates();
        endTime = new Times();
    }
    if(!isLoaded)
    {
        loadCodes();
    }
}

/**
 * Copy constructor - makes a deep copy of an Alerts
 * @param anAlert Alerts: The Alerts object to make a copy of
 */
public Alerts(Alerts anAlert)
{
    FIPSCode = anAlert.getFIPSCode();
    startDate = anAlert.getStartDate();
    startTime = anAlert.getStartTime();
    endDate = anAlert.getEndDate();
    endTime = anAlert.getEndTime();
    warningCode = anAlert.getWarningCode();
```

```
}

/**
 * Accessor method to get the County FIPS code
 * @return String: A County FIPS code
 */
public String getFIPSCode()
{
    return FIPSCode;
}

/**
 * Accessor method to get the startDate of an alert
 * @return Dates: A deep copy of the startDate of an alert
 */
public Dates getStartDate()
{
    return new Dates(startDate);
}

/**
 * Accessor method to get the endDate of an alert
 * @return Dates: A deep copy of the endDate of an alert
 */
public Dates getEndDate()
{
    return new Dates(endDate);
}

/**
 * Accessor method to get the startTime of an alert
 * @return Times: A deep copy of the startTime of an alert
 */
public Times getStartTime()
{
    return new Times(startTime);
}

/**
 * Accessor method to get the endTime of an alert
 * @return Times: A deep copy of the endTime of an alert
 */
public Times getEndTime()
{
    return new Times(endTime);
}

/**
 * Accessor method to get the warningCode of an alert
 * @return String: The warningCode of an alert
 */
public String getWarningCode()
{
    return warningCode;
}

/**
 * Method that gets the current CountyList being used regarding alrets
 * This method only returns a copy of the current address being used
 * It DOES NOT make a deep copy of the list
```

```
 * @return CountyList : The current CountyList stored in memory being used.
 */
public static CountyList getCountyList()
{
    return countyList;
}

/**
 * Mutator method to set the FIPSCode of an alert
 * @param FIPSCode String: A County (or State or US) FIPS code
 */
public void setFIPSCode( String FIPSCode )
{
    this.FIPSCode = FIPSCode;
}

/**
 * Mutator method to set the startDate of an alert - makes a deep copy
 * @param startDate Dates: A Dates object representing a starting date for an alert
 */
public void setStartDate( Dates startDate )
{
    this.startDate = new Dates(startDate);
}

/**
 * Mutator method to set the startTime of an alert - makes a deep copy
 * @param startTime Times: A Times object representing the starting time for an alert
 */
public void setStartTime( Times startTime )
{
    this.startTime = new Times(startTime);
}

/**
 * Mutator method to set the endDate for an alert - makes a deep copy
 * @param endDate Dates: A Dates object representing the ending date for an alert
 */
public void setEndDate( Dates endDate )
{
    this.endDate = new Dates(endDate);
}

/**
 * Mutator method to set the endTime for an alert - makes a deep copy
 * @param endTime Times: A Times object representing the ending time for an alert
 */
public void setEndTime( Times endTime )
{
    this.endTime = new Times(endTime);
}

/**
 * Mutator method to set the warningCode for an alert
 * @param warningCode String: A String representing the warning code
 */
public void setWarningCode( String warningCode )
{
    this.warningCode = warningCode;
}
```

```
/**
 * Static Mutator method to set the countyList object to be able to reference county population data
 * @param list CountyList: a CountyList object containing a list of county populations
 */
public static void setCountyList( CountyList list )
{
    countyList = list;
}

/**
 * Method that loads the warning labels from the file stored in ALERT_FILE into the class.
 *
 * The warning file is assumed to have the 2 character codes listed first, 1 character codes listed
 *      next, and color security codes last with appropriate headers
 *
 * There are a number of possible error messages that give the user the option to exit the program
 *      if the data is not formatted correctly
 *
 * THIS METHOD ALLOWS THE USER TO EXIT THE PROGRAM IF THERE ARE PROBLEMS LOADING THE DATA FILE
 *
 */
public void loadCodes()
{
    File warningData;
    String message;
    Scanner inputFile;
    int i=0;

    boolean fileEndEarly = false;
    boolean fileEnd = false;
    boolean continueLoop = true;

    String inputLine;

    try
    {
        String fips, name, state;                    // Work variables

        // Build list of county objects
        warningData = new File( ALERT_FILE );

        if (!warningData.exists())  // file not found
        {
            message = "The file " + ALERT_FILE + " containing warning definitions was not found.\n" +
                    "Do you wish to exit the program?";

            quitOption( message ); // gives user choice to quit
            return;                // exit method since no file found

        }

        inputFile = new Scanner( warningData );

        // Read one line at a time
        // skip through junk until "WEATHER AND NATURAL DISASTER" reached or at file end
        fileEnd = !(skipUntil( inputFile, "WEATHER AND NATURAL DISASTER" ));

        continueLoop = true;
        i = 0;          // used to work through array elements
```

```
// load warning codes one line at a time
while (continueLoop && !fileEndEarly && i<MAX_CODES)
{
    // keep loading codes until try attempt fails, then proceed
    try
    {
        if (inputFile.hasNext())
        {
            inputLine = inputFile.nextLine();
            weatherWarningCodes[i] = inputLine.substring( 1, 3 );
            weatherWarningType[i] = inputLine.substring( 4 );
            i++;
        } else
        {
            fileEndEarly = true;
        }
    } catch (StringIndexOutOfBoundsException e)
    {
        continueLoop = false;
        numCodes = i;
    }
}

if (i==MAX_CODES)       // should not happen unless file modified
{
    message = "The maximum number of weather warning codes has been uploaded.\n" +
            "Some codes may not have been uploaded.\n" +
            "Do you wish to exit?";

    quitOption( message );  // give user option to quit

}

// skip through junk until "where" reached or at file end
fileEndEarly = !(skipUntil( inputFile, "where" ));

continueLoop = true;
i = 0;          // used to work through array elements
// load warning severity codes one line at a time
while (continueLoop && !fileEndEarly && i<MAX_SECURITY)
{
    // keep loading codes until try attempt fails, then proceed
    try
    {
        if (inputFile.hasNext())
        {
            inputLine = inputFile.nextLine();
            weatherWarningSeverity[i] = inputLine.substring( 0, 1 );
            weatherWarningSeverityType[i] = inputLine.substring( 2 );
            i++;
        } else
        {
            fileEndEarly = true;
        }
    } catch (StringIndexOutOfBoundsException e)
    {
        continueLoop = false;
        numSeverity = i;
    }
}
```

```
if (i==MAX_SECURITY)        // should not happen unless file modified
{
    message = "The maximum number of weather warning severities has been uploaded.\n" +
            "Some codes severities may not have been uploaded.\n" +
            "Do you wish to exit the program?";

    quitOption( message );  // give user option to quit
}

// skip through junk until "NATIONAL SECURITY" reached or at file end
fileEndEarly = !(skipUntil( inputFile, "NATIONAL SECURITY" ));

continueLoop = true;
i = 0;          // used to work through array elements
// load warning security codes one line at a time
while (continueLoop && inputFile.hasNext())
{
    int spacePos;

    // keep loading codes until try attempt fails, then proceed
    try
    {
        inputLine = inputFile.nextLine();
        spacePos = inputLine.indexOf( ' ' );

        securityWarningCode[i] = inputLine.substring( 0, spacePos );
        securityWarningType[i] = inputLine.substring( spacePos ).trim(); // also removes leading whitespace
        i++;

    }
    catch (StringIndexOutOfBoundsException e)
    {
        continueLoop = false;
        numSecurity = i;
    }
}

if (i==MAX_SECURITY)        // should not happen unless file modified
{
    message = "The maximum number of national security advisories has been uploaded.\n" +
            "Some security levels may not have been uploaded." +
            "Do you wish to end the program?";

   quitOption( message );   // give user option to quit
}


// end program if data not all loaded properly
if (fileEndEarly || numSeverity==0)
{
    message = "Not all the required definitions were found in the file " + ALERT_FILE + ".\n" +
            "Do you wish to end the program?";

    quitOption( message );  // give user option to quit
}

message = "The security and warning message definitions from the file " + ALERT_FILE +
        "\nare now uploaded into memory.";
```

```
        Alert alert = new Alert( Alert.AlertType.INFORMATION );
        alert.setTitle( "DATA LOADED" );
        alert.setContentText( message );
        alert.showAndWait();

        isLoaded = true;


    }
    catch (IOException e)  // if error loading data, give error message and end program
    {
        message = "There was an error processing the file " + ALERT_FILE + ".\n" +
                "Do you wish to end the program?";

        quitOption( message );  // give user option to quit
    }

    return;
}

/**
 * This method takes an open file, and a string flag, and will advance the file stream until
 * the flag is found
 * @param openFile  Scanner: An input file stream to advance until the flag is found
 * @param stringFlag String: An string at the beginning of a line that the file is searching for
 * @return boolean: true if the flag was found, and false if not
 */
public boolean skipUntil(Scanner openFile, String stringFlag)
{
    boolean isFound = false;
    String inputLine = "";
    while (openFile.hasNext() && !isFound)
    {
        inputLine = openFile.nextLine();
        if (inputLine.startsWith( stringFlag ))
        {
            isFound = true;
        }

    }
    return isFound;
}

/**
 * Method that asks if the user wants to quit because of an error in processing
 * @param message String: The question to ask the user
 *
 * THIS METHOD WILL END THE PROGRAM IF THE USER SELECTS OK
 */
private void quitOption(String message)
{
    Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
    alert.setTitle( "Quit?" );
    alert.setContentText( message );
    Optional<ButtonType> result = alert.showAndWait();
    if (result.get() == ButtonType.OK)
    {
        System.exit( 0 );
    }
    return;
```

```
}

/**
 * Method that saves all the alerts that have been stored into memory to a String
 * @return String: A readable String containing the contents of all the stored warnings in memory
 */
public static String possibleAlertsString()
{

    String alertString="NATIONAL SECURITY WARNINGS:\n";

    // add national security warnings
    for (int i=0; i<numSecurity; i++)
    {
        alertString += securityWarningCode[i] + " " + securityWarningType[i] + "\n";

    }

    // add first letter warning codes
    alertString +="\nFIRST LETTER WEATHER SEVERITY CODES:\n";

    for (int i=0; i<numSeverity; i++)
    {
        alertString += weatherWarningSeverity[i] + " " + weatherWarningSeverityType[i] + "\n";
    }

    // add second and third letter warning codes

    alertString +="\nSECOND AND THIRD LETTER WEATHER SEVERITY CODES:\n";
    for (int i=0; i<numCodes; i++)
    {
        alertString += weatherWarningCodes[i] + " " + weatherWarningType[i] + "\n";

    }


    return alertString;
}

/**
 * Method that identifies and returns the security warning or weather alert description based on its code
 * @return String: A String storing the type of security or weather alert
 */
public String getWarningString()
{
    boolean isFound = false;
    String str = "Not Found";
    for (int i=0; i<numSecurity && !isFound; i++)           // check if code is in security codes
    {
        if (warningCode.equals( securityWarningCode[i] ))
        {
            str = securityWarningType[i];
            isFound = true;
        }
    }

    if (!isFound && warningCode.length()==3)                // if not found, check if code is in weather codes
    {
        isFound=false;
        for (int i=0; i<numCodes; i++)
```

```
        {
            // check if ith code matches the last two letters of warningCode
            if (weatherWarningCodes[i].equals( warningCode.substring( 1 ) ))
            {
                str = weatherWarningType[i] + " ";
                for(int j=0; j<numSeverity && !isFound; j++)
                {
                    // check if jth severity code matches the first letter of warningCode
                    if(weatherWarningSeverity[j].equals( warningCode.substring( 0,1 ) ))
                    {
                        isFound = true;
                        str +=weatherWarningSeverityType[j];
                    }
                }

                if (!isFound) // no matches for first letter
                {
                    // end i loop, because no possible match on first character
                    str = "Not Found";
                    i=numCodes;
                }
            }
        }
    }
return str;
}

/**
 * Method that creates a priority level based on the type of alert
 * This is used to help sort the alerts
 * 0 - RED
 * 1 - ORANGE
 * 2 - YELLOW
 * 3 - BLUE
 * 4 - GREEN
 * 5 - Warning
 * 6 - Watch
 * 7 - Advisory
 * 8 - Other (unidentified)
 *
 * @return int : A priority level as listed above based on the warningCode
 */
public int getAlertPriority()
{
    int alertPriority = numSecurity + numSeverity;
    boolean isFound = false;
    for (int i=0; i<numSecurity && !isFound; i++)           // check if code is in security codes
    {
        if (warningCode.equals( securityWarningCode[i] ))
        {
            alertPriority = i;
            isFound = true;
        }
    }

    if (!isFound && warningCode.length()>0)                // if not found, check if code is in weather codes
    {
        isFound = false;

        for (int j = 0; j < numSeverity && !isFound; j++)
```

Clever approach to the sorting.

```
        {
            // check if jth severity code matches the first letter of warningCode
            if (weatherWarningSeverity[j].equals( warningCode.substring( 0, 1 ) ))
            {
                isFound = true;
                alertPriority = j + numSecurity;
            }
        }
    }
    return alertPriority;
}

/**
 * Method that checks if two alerts are equal
 * @param anAlert Alerts: Alerts object to compare to this one
 * @return boolean: true if the alerts are equal, false if they are not
 */
public boolean isEqual(Alerts anAlert)
{
    boolean equal = false;
    if (FIPSCode.equals( anAlert.getFIPSCode()) &&
            warningCode.equals( anAlert.getWarningCode() ) &&
            startDate.isEqual( anAlert.getStartDate() ) &&
            endDate.isEqual( anAlert.getEndDate() ) &&
            startTime.isEqual( anAlert.getStartTime() ) &&
            endTime.isEqual( anAlert.getEndTime() ) )
    {
        equal = true;
    }
    return equal;
}


/**
 * Method to compare two Alerts objects
 * The method compares the alert priority and if equal, compares the population (if it exists)
 *
 * The method will return 1 if
 * 1) "this" ALerts has a higher priority (lower value) than the anAlert parameter
 * 2) "this" Alerts has the same priority as the anAlert parameter and "this" Alerts has a greater population
 *
 * The method will return 0 if
 * "this" Alerts has the same priority and population as the anAlert parameter
 *
 * The method will return -1 if
 * 1) "this" Alerts has a lower priority (higher value) than the anAlert parameter
 * 2) "this" Alerts has the same priority as the anAlert parameter and "this" Alerts has a lower population
 *
 * @param anAlert Alerts: An Alert object that is being compared to this object
 * @return int: See Comment
 */
public int compareTo(Alerts anAlert)
{
    int value = 0;
    // compare priorities
    if (getAlertPriority()<anAlert.getAlertPriority())
    {
        value = 1;
    }
    else if(getAlertPriority()>anAlert.getAlertPriority())
```

```
    {
        value = -1;
    }
    else // alert level same so compare populations if exist
    {
        if(countyList!=null)
        {
            if (countyList.getPopulation( FIPSCode )>countyList.getPopulation( anAlert.getFIPSCode() ))
            {
                value = 1;
            }
            else if (countyList.getPopulation( FIPSCode )<countyList.getPopulation( anAlert.getFIPSCode() ))
            {
                value = -1;
            }
        }

    }

    return value;
}

/**
 * Method that returns a string based on the alert information stored in the alert
 * It also uses information in the CountyList Class if available
 * @return String: A String message displaying the Alerts data in readable format
 */
@Override
public String toString()
{

    String str="";

    // get the warning type
    str += getWarningString() + " for ";

    // get the county name or FIPS code
    if (countyList!=null)
    {
        str += countyList.getName( FIPSCode ) + ",  " + countyList.getState( FIPSCode);
    }
    else
    {
        str += "county " + FIPSCode;
    }

    str +="\n";

    // get the start date
    str += startDate.toString() + " " + startTime.toString() + " - ";

    // get the end date
    str += endDate.toString() + " " + endTime.toString() + "\n";

    str += "Population Impact: ";

    // get the population if countyList exists
    if (countyList!=null)
    {
        str += String.format( "%,d" , countyList.getPopulation( FIPSCode ));
```

```
        }
        else
        {
            str += "???,???";
        }

        return str;
    }
}
```

```
/*********************************************************************************************
 *  This class stores an array of Alerts with county FIPS code, start time, end time and warning code
 *
 *  The method allows the the alerts to be sorted based on priority and populations
 *  The method allows the display of all the alerts by saving them as a String
 *
 *  CST 283 Programming Assignment 5
 *  @author Michael Clinesmith
 *********************************************************************************************/
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;

import java.io.File;
import java.io.IOException;
import java.util.NoSuchElementException;
import java.util.Optional;
import java.util.Scanner;
import java.util.StringTokenizer;

public class AlertList   √
{
    private static final int ARRAY_LIMIT = 1000;
    private Alerts[] alertsArray;
    private int numOfElements;
    private static CountyList countyList = null; // allows access to countyList for all elements of array

    /**
     * No-argument constructor
     */
    public AlertList()
    {
        alertsArray = new Alerts[ARRAY_LIMIT];    √
        numOfElements = 0;
    }

    /**
     * Constructor that creates Alerts from data located in the file filename
     *
     * THE USER MAY CHOOSE TO END THE PROGRAM IN THIS METHOD IF THERE ARE PROBLEMS LOADING THE DATA
     *
     * @param filename String: The file that holds Alerts data
     */
    public AlertList(String filename)
    {
        alertsArray = new Alerts[ARRAY_LIMIT];

        File alertData;
        String message;
        Scanner inputFile;
        StringTokenizer alerttokens;
        int i=0;

        String inputLine;


        String fips, startDateTime, endDateTime, code;                    // Work variables

        // Build list of county objects
        alertData = new File( filename );
```

```
if (!alertData.exists())  // file not found
{
    message = "The file " + filename + " containing alert data was not found.\n" +
        "Do you wish to end the program?";

    quitOption( message );  // give user option to end program
    numOfElements = 0;
}
else
{

    try
    {
        inputFile = new Scanner( alertData );

        // Read input file while more data exist
        // Read one line at a time (assuming each line contains one username)
        i = 0;          // used to work through array elements
        while (inputFile.hasNext() && i<ARRAY_LIMIT)
        {

            inputLine = inputFile.nextLine();
            alerttokens = new StringTokenizer( inputLine, "," );

            try
            {
                fips = alerttokens.nextToken();
                startDateTime = alerttokens.nextToken();
                endDateTime = alerttokens.nextToken();
                code = alerttokens.nextToken();

                alertsArray[i] = new Alerts( fips, startDateTime, endDateTime, code );
                i++;
            } catch (NoSuchElementException ex)
            {
                message = "There was an error processing an alert record in " + filename + ".\n" +
                        "Do you wish to end the program?  If not, the record will be skipped.";

                quitOption( message );
            }

        }
        numOfElements = i;    // Capture number of elements
        inputFile.close();
        message = "The alert data from the file " + filename +
                "\nis now uploaded into memory.";

        Alert alert = new Alert( Alert.AlertType.INFORMATION );
        alert.setTitle( "DATA LOADED" );
        alert.setContentText( message );
        alert.showAndWait();

        if (i==ARRAY_LIMIT)
        {
            message = "The maximum number of alerts is stored into memory. " +
                    "\nSome records may not have been uploaded." +
                    "\nDo you wish to end the program?";
            quitOption( message );  // give user option to end the program
        }
```

```
        }
        catch(IOException e)  // if error loading data, give error message and end program
        {
            message = "There was an error processing the file " + filename + ".\n" +
                    "No alerts were loaded." +
                    "\nDo you wish to end the program?";

            quitOption( message );      // give user option to end the program
        }
    }
    return;
}

/**
 * Method that allows an Alerts to be added to the alertsArray list
 * @param newAlert Alerts: Alerts object to be added to the alertsArray list
 * @return boolean: true if the object was added, false if not
 */
public boolean addAlert(Alerts newAlert)
{
    boolean isAdded = false;
    if (numOfElements<ARRAY_LIMIT)
    {
        isAdded = true;
        alertsArray[numOfElements]= new Alerts(newAlert);
        numOfElements++;
    }
    return isAdded;
}


/**
 * Method that returns all the alerts that have been stored into memory to a String
 * @return String: A readable String containing the contents of all the stored warnings in memory
 */
public static String getPossibleAlertsString()
{
    return Alerts.possibleAlertsString();
}

/**
 * Method that returns the countyList reference stored in this object
 * NOTE - THIS METHOD ONLY RETURNS THE MEMORY REFERENCE, IT DOES NOT MAKE A DEEP COPY OF THE OBJECT
 * @return
 */
public static CountyList getCountyList()
{
    return countyList;
}

/**
 * Method saves the address of the CountyList inside the Class and also sets it for the Alerts class
 * This method only saves the memory address, it DOES NOT make a deep copy of the list
 * @param list CountyList: the list of County data to allow access from Alerts Class
 */
public static void setCountyList(CountyList list)
{
    countyList = list;
    Alerts.setCountyList( list );
}
```

```
/**
 * Method that uses a bubble sort to sort the contents in the Alert list
 * The Alerts are sorted primarily by Alert Type, secondarily by population
 */
public void sortAlerts()
{
    boolean isDone = false;     // used to end unnecessary loops - if no changes made during loop, exit
    Alerts temp;

    // bubble sort
    for (int i=numOfElements-2; i>=0; i--) // loop sorts one less element each time
    {
        isDone = true;
        for (int j=0; j<=i; j++)
        {
            // if element at j has lower priority than element at j+1 - swap elements
            if (alertsArray[j].compareTo( alertsArray[j+1] )<0)
            {
                isDone = false;
                temp = alertsArray[j];
                alertsArray[j] = alertsArray[j+1];
                alertsArray[j+1] = temp;
            }
        }
    }

    return;
}

/**
 * Method that puts the all the alerts together as a String and displays them
 * @return  String: A concatenation of all the alert strings stored in the AlertList object
 */
public String displayAllAlerts()
{
    String str = "No alerts on record.";

    if (numOfElements>0)
    {
        str = "";
        for (int i=0; i<numOfElements; i++)
        {
            str += alertsArray[i].toString() + "\n\n";
        }
    }

    return str;
}

/**
 * Method that asks if the user wants to quit because of an error in processing
 * @param message String: The question to ask the user
 *
 * THIS METHOD WILL END THE PROGRAM IF THE USER SELECTS OK
 */
private void quitOption(String message)
{
    Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
    alert.setTitle( "Quit?" );
```

```
        alert.setContentText( message );
        Optional<ButtonType> result = alert.showAndWait();
        if (result.get() == ButtonType.OK)
        {
            System.exit( 0 );
        }
        return;
    }



}
```

```
/*********************************************************************************************
 *  This class contains the main driver and interface for viewing warning alerts data
 *
 *  The application initially displays the different definitions of alerts
 *
 *  The user has the option of loading from the default alerts.txt file and displaying those alerts
 *  or choosing a specific file to display those alerts
 *  or displaying the alert definitions
 *
 *  The alerts are sorted by level of security warning, then warnings, watches, and advisories
 *  In those levels, the alerts are sorted by population
 *
 *  CST 283 Programming Assignment 5
 *  - File heavily modified from CharacterListInterface.java from CST 283 PA 3
 *  @author Michael Clinesmith
 *********************************************************************************************/

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.FileChooser;
import javafx.stage.Stage;
import javafx.scene.layout.BorderPane;

import java.io.File;
import java.util.Optional;

public class WarningInterface extends Application
{
    // main node
    private BorderPane mainLayout;

    // holds data
    private CountyList countyList;
    private String currentFileName="";
    private AlertList alertsList;

    // data files
    private String COUNTY_FIPS_FILE = "fipsCounty.txt";
    private String COUNTY_POP_FILE = "popCounty.txt";
    private String WARNING_SPEC_FILE = "warningList.txt";
    private String ALERT_DEF_FILE = "alerts.txt";

    // titlebox objects
    private BorderPane TitleBar;
    private Image Airyimage;
    private ImageView AiryimageView;
    private Button quitButton;
    private Label appLabel;
    private HBox quitButtonHBox;

    // Load/Save objects
```

Nice job with the interface.

```
private VBox loadFileButtonVBox;
private Button loadFileButton, loadAlertsTxtButton, warningsButton;
private File selectedFile;

// Data Display objects
private TextArea characterDataDisplay;
private HBox centerHBox;

/**
 * Starting method of application - calls launch
 * @param args String[]: Not used
 */
public static void main( String[] args )
{
    // Launch the application.
    launch( args );
}

/**
 * Method that calls the initializeScene method and creates the scene
 * @param primaryStage Stage object used to create the stage
 */
@Override
public void start( Stage primaryStage )
{
    initializeScene(primaryStage);

    // Set up overall scene
    Scene scene = new Scene( mainLayout, 1100, 900 );
    scene.getStylesheets().add( "Warning.css" );
    primaryStage.setScene( scene );
    primaryStage.setTitle( "Security and Weather Alert Program" );
    primaryStage.show();
}

/**
 * Method that calls other methods to create the interface, then combines the parts in the
 *      mainLayout object
 * @param primaryStage Stage object used to create the stage
 */
public void initializeScene(Stage primaryStage)
{

    alertsList = new AlertList( ALERT_DEF_FILE );
    countyList = new CountyList( COUNTY_FIPS_FILE, COUNTY_POP_FILE );
    alertsList.setCountyList( countyList );

    initializeTitleBar();
    initializeLoadButton(primaryStage);
    initializeDataTextArea();

    mainLayout = new BorderPane();
    mainLayout.setTop( TitleBar );
    mainLayout.setLeft(loadFileButtonVBox);
    mainLayout.setCenter( centerHBox );
}

/**
 * Method that creates the title bar that contains an image, title, and quit button
 */
```

```
public void initializeTitleBar()
{
    Airyimage=new Image("file:AiryJavaJXDrawing.png");
    AiryimageView= new ImageView( Airyimage );

    quitButton = new Button( "QUIT" );
    quitButton.setOnAction( new AppButtonHandler() );
    quitButton.setAlignment( Pos.BASELINE_RIGHT );
    quitButton.setPadding( new Insets( 20 ) );

    quitButtonHBox = new HBox(quitButton);
    quitButtonHBox.setPadding(new Insets( 20 ));

    appLabel = new Label("Security and Weather Alert System");
    appLabel.setAlignment( Pos.CENTER );
    appLabel.setStyle( "-fx-font-size: 28; -fx-text-fill:  orange" );

    TitleBar = new BorderPane();
    TitleBar.setLeft(AiryimageView);
    TitleBar.setRight(quitButtonHBox);
    TitleBar.setCenter(appLabel);
}

/**
 * Method that creates the Load buttons and warning button
 * The actions of the buttons are implemented in this method
 *
 * @param primaryStage Stage: The Stage used to allow access to the file chooser
 */
public void initializeLoadButton(Stage primaryStage)
{
    FileChooser fileChooser = new FileChooser();

    loadFileButton = new Button("Load alerts data file");
    // code to load file
    loadFileButton.setOnAction(e -> {
        selectedFile = fileChooser.showOpenDialog(primaryStage);
        if (selectedFile!=null)
        {
            String message = "Do you want to load the file " + selectedFile.getName() + "?\n" +
                    "This will remove the alerts shown in the application?";
            Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
            alert.setTitle( "Load File?" );
            alert.setContentText( message );
            Optional<ButtonType> result = alert.showAndWait();

            if (result.get() == ButtonType.OK)
            {
                alertsList = new AlertList( selectedFile.getName() );
                alertsList.sortAlerts();
                currentFileName = selectedFile.getName();
                characterDataDisplay.setText( alertsList.displayAllAlerts() );                }

        }
    });
    loadFileButton.setPadding( new Insets( 20 ) );

    loadAlertsTxtButton = new Button("Load alerts.txt");
    loadAlertsTxtButton.setPadding( new Insets( 20 ) );
```

```
    // code to load alerts.txt
    loadAlertsTxtButton.setOnAction(e -> {

        String message = "Do you want to load the file " + ALERT_DEF_FILE + "?\n" +
            "This will remove the alerts shown in the application.";

        Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
        alert.setTitle( "Load File?" );
        alert.setContentText( message );
        Optional<ButtonType> result = alert.showAndWait();

        if (result.get() == ButtonType.OK)
        {
            alertsList = new AlertList( ALERT_DEF_FILE );
            alertsList.sortAlerts();
            characterDataDisplay.setText( alertsList.displayAllAlerts() );
            currentFileName = ALERT_DEF_FILE;
        }

    });

    characterDataDisplay = new TextArea( alertsList.getPossibleAlertsString()); // initially set to display warnings

    warningsButton = new Button("Display warnings");
    warningsButton.setPadding( new Insets( 20 ) );

    // code to display warnings
    warningsButton.setOnAction(e -> {

        String message = "Do you want to show warning definitions?" +
                "This will remove the alerts shown in the application.";

        Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
        alert.setTitle( "View Warning Definitions" );
        alert.setContentText( message );
        Optional<ButtonType> result = alert.showAndWait();

        if (result.get() == ButtonType.OK)
        {
            characterDataDisplay.setText( alertsList.getPossibleAlertsString() );
        }

    });

    loadFileButtonVBox = new VBox(20, loadAlertsTxtButton, loadFileButton, warningsButton );
    loadFileButtonVBox.setPadding( new Insets( 20 ) );
}

/**
 * Method that creates the text area data display
 */
public void initializeDataTextArea()
{
    characterDataDisplay = new TextArea( alertsList.getPossibleAlertsString());
    characterDataDisplay.setPrefColumnCount( 100 );
    centerHBox = new HBox( characterDataDisplay );
    centerHBox.setPadding( new Insets( 20 ) );
}

/**
```

```
     * Class that handles ActionEvents for quit button
     */
    class AppButtonHandler implements EventHandler<ActionEvent>
    {
        /**
         * Method that handles ActionEvents for the quit button
         * @param event ActionEvent: Event caused by clicking the quit button
         */
        @Override
        public void handle( ActionEvent event )
        {
            String message;

            if (event.getSource() == quitButton)   // user chooses to quit
            {
                message = "Do you want to quit the application?";

                Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
                alert.setTitle( "Quit?" );
                alert.setContentText( message );
                Optional<ButtonType> result = alert.showAndWait();
                if (result.get() == ButtonType.OK)
                {
                    System.exit( 0 );
                }
            }
        }
    }
}
```

Security and Weather Alert Program

QUIT

Security and Weather Alert System

Load alerts.txt

Load alerts data file

Display warnings

```
Significant risk of terrorist attacks for District of Columbia,  DC
July 1, 2016 12:00:00 AM - July 11, 2016 11:59:00 PM
Population Impact: 646,449

Low risk of terrorist attacks for Salt Lake County,  UT
December 24, 2016 12:00:00 AM - December 31, 2016 11:59:00 PM
Population Impact: 1,079,721

Winter Storm  Warning for Midland County,  MI
February 12, 2016 1:00:00 PM - February 13, 2016 12:00:00 PM
Population Impact: 83,919

Blizzard Warning for Petroleum County,  MT
February 15, 2016 7:00:00 AM - February 17, 2016 4:00:00 AM
Population Impact: 506

Hurricane Watch for Sarasota County,  FL
September 10, 2016 12:00:00 AM - September 12, 2016 12:00:00 AM
Population Impact: 390,429

Dense Fog Advisory for Androscoggin County,  ME
March 17, 2016 3:00:00 AM - March 18, 2016 10:00:00 PM
Population Impact: 107,604

Excessive Heat Advisory for Cherokee County,  TX
August 3, 2016 12:00:00 PM - August 5, 2016 6:00:00 PM
Population Impact: 50,878
```