30/30 points for Program 1
(Testing key included as last page)

```
/*********************************************************************************************
 *  This class handles the processing of a Location object
 *
 *  It was written to allow extra processing on the location data, but currently only has a string data object
 *
 *  CST 283 Programming Assignment 1
 *  @author Michael Clinesmith
 *********************************************************************************************/
public class Locations
{

    String location;

    /**
     * No-argument constructor
     */
    public Locations()
    {
        location = "";
    }

    /**
     * Constructor that takes a String object
     * @param loc String: The String containing an object's general location
     */
    public Locations(String loc)
    {
        location = loc;
    }

    /**
     * Accessor method that returns the String containing an object's general location
     * @return String: The general location of a quake record
     */
    public String getLocation()
    {
        return location;
    }

    /**
     * Mutator method that sets a location
     * @param location String: A String containing a general location of a record
     */
    public void setLocation( String location )
    {
        this.location = location;
    }

    /**
     * Method that returns the data stored in the Location object
     * @return String: The String that states the general location
     */
    @Override
    public String toString()
    {
        return location;
    }

    /**
     * Method that checks if this Location object contains the same data as another
```

Wow. Epic solution. Awesome job with the class and work breakdown. Program testing was spot-on. Error checking excellent. Really liked the flexibility built in with the command line interface. Very cool.

Be sure to archive this one. Some of these classes might well be reusable.

And, of course all javadoc- ready class files …

```
     * @param locations Location: A Location object to compare
     * @return boolean: true if the objects have the same description, false otherwise
     */
    public boolean isEqual( Locations locations)
    {
        boolean equal = false;
        if( location.equals( locations.getLocation()))
        {
            equal = true;
        }
        return equal;
    }
}
```

```
/*******************************************************************************************
 *  This class handles the processing of a quake records
 *
 *  It contains an array to manage all the data records uploaded from a file
 *  It has method to allow searching for data records by region, date, and magnitude
 *  The methods may display to the console, or to a file based on the user's request
 *
 *  CST 283 Programming Assignment 1
 *  @author Michael Clinesmith
 *******************************************************************************************/
import javax.swing.JOptionPane;
import java.util.NoSuchElementException;
import java.util.Scanner;
import java.io.*;
import java.util.StringTokenizer;

public class QuakeData
{
    private final int MAX_RECORDS = 130000;
    private QuakeRecord[] records = new QuakeRecord[MAX_RECORDS];
    private int numOfRecords, numOfRecordsMissed;

    /**
     * No-argument constructor
     */
    public QuakeData()
    {
        numOfRecords = 0;
        numOfRecordsMissed = 0;
    }

    /**
     * Constructor with filename
     * The constructor processes the data in the file name given, and if valid, creates an array of records
     * containing the quake data
     * @param fileName String: the name of the file to open containing quake data records
     */
    public QuakeData(String fileName)
    {
        String message;
        File quakeData;                     // file that holds the quake data
        Scanner inputFile;                  // used to get data from file
        String inputLine;                   // String used to get a line of file input

        numOfRecords = 0;
        numOfRecordsMissed = 0;

        try                                         // catch if problems loading data
        {
            quakeData = new File(fileName);

            if(!quakeData.exists())                 // end program if file data does not exist
            {
                message = "The file " + fileName + " does not exist for processing data.\n" +
                        "The program will now end.";

                JOptionPane.showMessageDialog(null, message);
                System.exit(0);
            }
```

```
        inputFile = new Scanner( quakeData );

        // Read input file while more data exist
        // Read one line at a time (assuming each line contains one quake record)
        while (inputFile.hasNext() && numOfRecords<MAX_RECORDS)
        {
            try                                 // catch if problem with an individual data record
            {
                inputLine = inputFile.nextLine();
                records[numOfRecords] = new QuakeRecord( inputLine );    // record processed in QuakeRecord class
                                                                         // will throw IOException if problem with data

                numOfRecords++;
            }
            catch (IOException e)
            {
                message = "There was an error processing a record in the file " + fileName + ".\n" +
                        "The record will be skipped.";

                JOptionPane.showMessageDialog( null, message, "Corrupted Data", JOptionPane.ERROR_MESSAGE );
                numOfRecordsMissed++;
            }

        }

        // if extra records not saved because array storage is full, count records so it can be noted
        if (inputFile.hasNext())
        {
            message = "Quake data array is full.\n" +
                    "The remaining records will be skipped.";

            JOptionPane.showMessageDialog( null, message, "Storage Full", JOptionPane.ERROR_MESSAGE );
        }
        while (inputFile.hasNext())
        {
            inputLine = inputFile.nextLine();
            numOfRecordsMissed++;                           // record not saved since no room
        }

        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" +
                "The program will now end.";

        JOptionPane.showMessageDialog(null, message);
        System.exit(0);
    }
}

/**
 * Accessor method for getting the number of records in the QuakeRecord array
 * @return int: The number of records in the array
 */
public int getNumOfRecords()
{
    return numOfRecords;
}
```

```
/**
 * Accessor method for getting the number of records that had problems and were not added to the QuakeRecord array
 * @return int: The number of records not added to the array
 */
public int getNumOfRecordsMissed()
{
    return numOfRecordsMissed;
}

/**
 * Accessor method of getting the max size of the QuakeRecord array
 * @return int: The maximum number of records that can be held in the array
 */
public int getMAX_RECORDS()
{
    return MAX_RECORDS;
}

/**
 * Accessor method for getting an array record
 * @param i int: The index number for the record in the QuakeRecord array
 * @return QuakeRecord: The deep copy of the record stored in the QuakeRecord array
 */
public QuakeRecord getRecord(int i)
{
    QuakeRecord rec;

    if ( i>=0 && i<numOfRecords)
    {
        rec = new QuakeRecord( records[i] );
    }
    else
    {
        rec = new QuakeRecord();
        rec.setLocation( "Invalid Record" );
    }
    return rec;
}

/**
 * Method to add a record to the QuakeRecords array
 * @param record QuakeRecord: Quake data stored in a QuakeRecord object
 * @return boolean: true if the record was saved in the array, false if not
 */
public boolean addRecord(QuakeRecord record)
{
    boolean recordAdded=true;

    if (numOfRecords==MAX_RECORDS)          // if array full, cannot add record
    {
        numOfRecordsMissed++;
        recordAdded = false;
    }
    else
    {
        records[numOfRecords] = new QuakeRecord( record );          // makes a deep copy of the input record
        numOfRecords++;
    }

    return recordAdded;
```

```
}

/**
 * Method to remove a record from the QuakeRecords array
 * @param record QuakeRecord: The QuakeRecord object to remove from the QuakeRecords array
 * @return boolean: true if the record was removed from the array, false if not
 */
public boolean removeRecord(QuakeRecord record)
{
    boolean recordRemoved = false;

    for(int i=0; i<numOfRecords && !recordRemoved; i++)
    {
        if (record.isEqual( records[i] ))
        {
            records[i] = records[numOfRecords-1];       // move the last record in the array to replace this record
            numOfRecords--;
            recordRemoved = true;
        }
    }

    return recordRemoved;
}

/**
 * Method to search the quake records by latitude and longitude and display them to the console
 * @param minLat double: the minimum latitude in the area to retreive records
 * @param maxLat double: the maximum latitude in the area to retreive records
 * @param minLon double: the minimum longitude in the area to retreive records
 * @param maxLon double: the maximum longitude in the area to retreive records
 * @return boolean: true if the search was done and displayed, false if there were problems with the parameters and
 *                  the records were not displayed
 */
public boolean regionSearch(double minLat, double maxLat, double minLon, double maxLon)
{
    boolean searchDone = true;
    int count = 0;
    String message = "";            // used for printing messages to console
    double recLat, recLong;         // temp variables for checking latitude and longitude in records
    // check if parameters are valid
    if( minLat >  maxLat || minLon > maxLon  || minLat < -90 || maxLat > 90 || minLon < -180 || maxLon > 180)
    {
        searchDone = false;
        message = "Latitude or longitude values not valid." +
                "\nSearch not processed.";
        System.out.println( message );
    }
    else
    {
        for(int i=0; i<numOfRecords; i++)
        {
            // if record in range, display it

            recLat = records[i].getLatitude();
            recLong = records[i].getLongitude();
            if (minLat <= recLat && recLat <= maxLat && minLon <= recLong && recLong <= maxLon)
            {
                System.out.println( records[i].toModString() );
                count++;
            }
```

```
        }
        System.out.println( count + " records found." );
    }

    return searchDone;
}

/**
 * Method to search the quake records by latitude and longitude and store them in the given filename
 * @param minLat double: the minimum latitude in the area to retreive records
 * @param maxLat double: the maximum latitude in the area to retreive records
 * @param minLon double: the minimum longitude in the area to retreive records
 * @param maxLon double: the maximum longitude in the area to retreive records
 * @param filename String: the name of the file to store the data
 * @return boolean: true if the search was done and saved, false if there were problems with the parameters or
 *                  the records were not saved
 */
public boolean regionSearch(double minLat, double maxLat, double minLon, double maxLon, String filename)
{
    File quakeOutputFile;
    PrintWriter outputFile;
    String message;
    String input;                   // used to get input from the user
    double recLat, recLong;         // temp variables for checking latitude and longitude in records
    boolean createFile = true;
    int count = 0;
    Scanner keyboard = new Scanner( System.in );

    quakeOutputFile = new File(filename);

    if(quakeOutputFile.exists())
    {
        message = "File " + filename + " already exists.\n" +
                "Do you wish to overwrite this file? Y/N";

        System.out.println( message );

        input = keyboard.nextLine();
        if (input==null || input.length()<1 || (input.charAt( 0 ) != 'Y' && input.charAt( 0 ) != 'y'))
        {
            createFile = false;
            message = "File not created.";
            System.out.println( message );
        }
    }
    //  check if parameter values for latitude and longitude are in range and valid
    if( minLat >  maxLat || minLon > maxLon  || minLat < -90 || maxLat > 90 || minLon < -180 || maxLon > 180)
    {
        createFile = false;
        message = "Latitude or longitude values not valid." +
                  "\nFile not created.";
        System.out.println( message );
    }

    if (createFile)
    {
        try                     // catch possible exception
        {
            outputFile = new PrintWriter( quakeOutputFile );
```

```
            for(int i=0; i<numOfRecords; i++)
            {
                // if record in range, display it
                recLat = records[i].getLatitude();
                recLong = records[i].getLongitude();
                if (minLat <= recLat && recLat <= maxLat && minLon <= recLong && recLong <= maxLon)
                {
                    outputFile.println( records[i].toModString() );
                    count++;
                }
            }
            outputFile.println( count + " records found." );
            outputFile.close();

            message = "Records saved in file " + filename + ".\n";
            System.out.println( message );

        }
        catch(FileNotFoundException e)
        {
            message = "File " + filename + " could not be opened.\n" +
                    "File not created.";
            System.out.println( message );
        }
    }

    return createFile;
}

/**
 * Method to retrieve and display quake records from the minDate to the maxDate and display them to the console
 * @param minDate Dates: The earliest date, saved in a Dates object
 * @param maxDate Dates: The latest date, saved in a Dates object
 * @return boolean: true if the search was done and displayed, false if there were problems with the parameters and
 *                  the records were not displayed
 */
public boolean dateSearch(Dates minDate, Dates maxDate)
{
    boolean searchDone = true;
    int count = 0;
    String message="";

    // check if date conditions valid and minDate not after maxDate
    if( !minDate.isValid() || !maxDate.isValid() || minDate.compareTo( maxDate)>0 )
    {
        searchDone = false;
        message = "Input dates not valid." +
                "\nSearch not processed.";
        System.out.println( message );
    }
    else
    {
        System.out.println( minDate.toModString() + " " + maxDate.toModString() );

        for(int i=0; i<numOfRecords; i++)
        {
            // if record date is not before minDate or after maxDate
            if (minDate.compareTo(records[i].getDate())<=0 && maxDate.compareTo( records[i].getDate())>=0)
            {
                System.out.println( records[i].toModString() );
```

```
                count++;
            }
        }
        System.out.println( count + " records found." );
    }

    return searchDone;
}

/**
 * Method to retrieve and display quake records from the minDate to the maxDate and to save to the given filename
 * @param minDate Dates: The earliest date, saved in a Dates object
 * @param maxDate Dates: The latest date, saved in a Dates objecct
 * @param filename String: The filename in which to save the record
 * @return boolean: true if the search was done and saved, false if there were problems with the parameters or
 *                  the records were not saved
 */
public boolean dateSearch(Dates minDate, Dates maxDate, String filename)
{
    File quakeOutputFile;
    PrintWriter outputFile;
    String message;
    String input;           // used to get input from the user
    boolean createFile = true;
    int count = 0;
    Scanner keyboard = new Scanner( System.in );

    quakeOutputFile = new File(filename);

    if(quakeOutputFile.exists())
    {
        message = "File " + filename + " already exists.\n" +
                "Do you wish to overwrite this file? Y/N";

        System.out.println( message );

        input = keyboard.nextLine();
        if (input==null || input.length()<1 || (input.charAt( 0 ) != 'Y' && input.charAt( 0 ) != 'y'))
        {
            createFile = false;
            message = "File not created.";
            System.out.println( message );
        }
    }

    // check if date conditions valid and minDate not after maxDate
    if( !minDate.isValid() || !maxDate.isValid() || minDate.compareTo( maxDate)>0 )
    {
        createFile = false;
        message = "Input dates not valid." +
                "\nSearch not processed.";
        System.out.println( message );
    }

    if (createFile)
    {
        try                     // catch possible exception
        {
            outputFile = new PrintWriter( quakeOutputFile );
```

```
            for (int i = 0; i < numOfRecords; i++)
            {
                // if record date is not before minDate or after maxDate
                if (minDate.compareTo( records[i].getDate() ) <= 0 && maxDate.compareTo( records[i].getDate() ) >= 0)
                {
                    outputFile.println( records[i].toModString() );
                    count++;
                }
            }
            outputFile.println( count + " records found." );
            outputFile.close();

            message = "Records saved in file " + filename + ".\n";
            System.out.println( message );

        }
        catch (FileNotFoundException e)
        {
            message = "File " + filename + " could not be opened.\n" +
                    "File not created.";
            System.out.println( message );
        }

    }

    return createFile;
}

/**
 * Method to search and retrieve records of quakes with magnitudes at least minMag and display to the console
 * @param minMag double: The minimum magnitude for quakes to retrieve data
 * @return boolean: true if the search was done and displayed, false if there were problems with the parameters and
 *                  the records were not displayed
 */
public boolean magnitudeSearch(double minMag)
{
    boolean searchDone = true;
    int count = 0;
    String message="";

    // check if minimum magnitude in appropriate range
    if( minMag<4.0 || minMag>10.0 )
    {
        searchDone = false;
        message = "Magnitude must be between 4.0 and 10.0" +
                  "\nSearch not processed.";
        System.out.println( message );
    }
    else
    {
        for(int i=0; i<numOfRecords; i++)
        {
            // if magnitude is at least minumum specified
            if (records[i].getRichter()>=minMag)
            {
                System.out.println( records[i].toModString() );
                count++;
            }
        }
        System.out.println( count + " records found." );
```

```
    }

    return searchDone;
}

/**
 * Method to search and retrieve records of quakes with magnitudes at least minMag and save to filename
 * @param minMag double: The minimum magnitude for quakes to retrieve data
 * @param filename String: The name of the file to use to save the quake records
 * @return boolean: true if the search was done and displayed, false if there were problems with the parameters and
 *                  the records were not displayed
 */
public boolean magnitudeSearch(double minMag, String filename)
{
    File quakeOutputFile;
    PrintWriter outputFile;
    String message;
    String input;           // used to get input from the user
    boolean createFile = true;
    int count = 0;
    Scanner keyboard = new Scanner( System.in );

    quakeOutputFile = new File(filename);

    if(quakeOutputFile.exists())
    {
        message = "File " + filename + " already exists.\n" +
                "Do you wish to overwrite this file? Y/N";

        System.out.println( message );

        input = keyboard.nextLine();
        if (input==null || input.length()<1 || (input.charAt( 0 ) != 'Y' && input.charAt( 0 ) != 'y'))
        {
            createFile = false;
            message = "File not created.";
            System.out.println( message );
        }
    }

    // check if minimum magnitude in appropriate range
    if( minMag<4.0 || minMag>10.0 )
    {
        createFile = false;
        message = "Magnitude must be between 4.0 and 10.0" +
                "\nSearch not processed.";
        System.out.println( message );
    }

    if (createFile)
    {
        try                     // catch possible exception
        {
            outputFile = new PrintWriter( quakeOutputFile );

            for (int i = 0; i < numOfRecords; i++)
            {
                // if magnitude is at least minumum specified
                if (records[i].getRichter()>=minMag)
                {
```

```
                    outputFile.println( records[i].toModString() );
                    count++;
                }
            }
            outputFile.println( count + " records found." );
            outputFile.close();

            message = "Records saved in file " + filename + ".\n";
            System.out.println( message );

        }
        catch (FileNotFoundException e)
        {
            message = "File " + filename + " could not be opened.\n" +
                    "File not created.";
            System.out.println( message );
        }

    }

    return createFile;
}

/**
 * Method to return information about the QuakeData object saved as a String
 * It does not include the data saved in the records as part of the string
 * @return String: A String listing the number of records, the number of records missed, the maximum array size,
 *                  and the first record of the QuakeRecord array
 */
@Override
public String toString()
{
    String str="";

    str += "Number of records: " + numOfRecords;
    str += "\nNumber of records missed: " + numOfRecordsMissed;
    str += "\nMaximum number records that can be loaded: " + MAX_RECORDS;

    if( numOfRecords>0)
    {
        str += "\nFirst record: " + getRecord( 0 ).toString();
    }

    return str;
}

/**
 * Method to return information about the QuakeData object saved as a String
 * It does not include the data saved in the records as part of the string
 * @return String: A String listing the number of records, the number of records missed, the maximum array size,
 *                  and the first record of the QuakeRecord array
 */
public String toModString()
{
    String str="";

    str += "Number of records: " + numOfRecords;
    str += "\nNumber of records missed: " + numOfRecordsMissed;
    str += "\nMaximum number records that can be loaded: " + MAX_RECORDS;
```

```
        if( numOfRecords>0)
        {
            str += "\nFirst record: " + getRecord( 0 ).toModString();
        }

        return str;
    }

}
```

```java
/*********************************************************************************************
 *  This class contains the main driver of the program, providing a basic interface for the user
 *  to request earthquake data.
 *
 *  The program loads data from the filename stored in QUAKE_FILENAME, then allows the user to
 *  request data in three different formats.
 *
 *  The user has the option to display the data to the console, or choose a filename to save the data to.
 *
 *  It does basic error checking on the data, requiring the data to be in range and maxes to be greater
 *  than mins.
 *
 *  The program will loop until the user indicates that no more requests are desired.
 *
 *  CST 283 Programming Assignment 1
 *  @author Michael Clinesmith
 *********************************************************************************************/

import javax.swing.JOptionPane;
import java.util.NoSuchElementException;
import java.util.Scanner;
import java.io.*;
import java.util.StringTokenizer;

public class QuakeMain
{
    private final static String QUAKE_FILENAME = "quakes.txt";
    private final static String QUAKE_REGULAR_OUTPUT_FILENAME = "quakesFormatted.txt";
    private final static String QUAKE_MOD_OUTPUT_FILENAME = "quakesModFormatted.txt";
    private static QuakeData quakeData;

    /**
     * Method is the main method of the program
     * It calls methods to create the quake data objects, display info messages, and run the main program
     * @param args String[]: Not used
     */
    public static void main( String args[] )
    {

        quakeData = importQuakeData( QUAKE_FILENAME );

        displayIntroMessage();
        mainProgram();
        displayEndingMessage();

    }

    /**
     * Method that directs the program to upload quake records to the computer memory
     * @param filename String: File name that contains the quake data
     * @return QuakeData: A QuakeData object that contains the quake data that was saved in filename
     */
    public static QuakeData importQuakeData( String filename )
    {
        String messageStr = "";
        QuakeData data = new QuakeData( filename );

        messageStr = filename + " opened and uploaded.\n" +
                data.getNumOfRecords() + " records uploaded.\n" +
                data.getNumOfRecordsMissed() + " records not uploaded."
```

```
    ;

    JOptionPane.showMessageDialog( null, messageStr );      // messages uses that data uploaded
    return data;
}

/**
 * Method that prints an introductory message
 */
public static void displayIntroMessage()
{
    String message =    "Welcome to the quake data processing program!\n" +
                        "Programmed by Michael Clinesmith\n\n" +
                        "Enter requests in the following forms:\n" +
                        "R, minLat, maxLat, minLon, maxLon\n" +
                        "D, minDate, maxDate\n" +
                        "M, minMag\n" +
                        "Enter HELP for more information on the commands.";

    System.out.println( message );

}

/**
 * Method that prints a help message explaining the commands allowed in the program
 */
public static void displayHelpMessage()
{
    String message =    "Console commands:\n" +
                        "R, minLat, maxLat, minLon, maxLon\n" +
                        "List all quakes in the dataset by region with north/south boundary between\n" +
                        "minLat and maxLat, and east/west boundary between minLon and maxLon\n" +
                        "Latitudes are between -90.0 amd 90.0 degrees and Longitudes between -180.0 and 180.0 degrees.\n\n" +
                        "D, minDate, maxDate\n" +
                        "List all quakes in the dataset by date with the calendar date of the quake from\n" +
                        "minDate to maxDate.  Dates are to be in the format of YYYY-MM-DD.\n\n" +
                        "M, minMag\n" +
                        "List all quakes in the dataset with magnitude minMag or greater\n" +
                        "Magnitudes need to be between 4.0 and 10.0.\n\n" +
                        "HELP\n" +
                        "Displays this help message";

    System.out.println( message );
}

/**
 * Method that prints an ending message to the computer console
 */
public static void displayEndingMessage()
{
    String message =    "Thank you for using the quake data processing program!";

    System.out.println( message );

}

/**
 * Method that handles the looping of the information requests of the program
 */
public static void mainProgram()
```

```
{
    String message = "Enter a quake data request or type HELP for more information:";
    boolean again = true;

    while (again)   // handles one complete user request each iteration
    {
        System.out.println( message );
        again = handleUserRequest();
    }
}

/**
 * Method that handles one information request from the user
 * It prompts for the user request then sends it to the appropriate method for processing
 * When completed, it prompts the user if another request is desired
 *
 * @return boolean: true if the user can make another info request, false if the user chooses to quit
 */
public static boolean handleUserRequest()
{
    boolean anotherRequest = true;
    String message, input;
    char requestType = ' ';
    Scanner keyboard = new Scanner(System.in);
    input = keyboard.nextLine();
    input = input.toUpperCase();                // converts string to upper case to ease processing

    if (input.length()>0)
    {
        requestType = input.charAt( 0 );
    }

    // handle request
    switch (requestType)
    {
        case 'R':
            processRRequest(input);
            break;
        case 'D':
            processDRequest(input);
            break;
        case 'M':
            processMRequest(input);
            break;
        case 'H':
            displayHelpMessage();
            break;
        default:
            System.out.println( "Request not understood or in the proper format.\nType HELP to view the proper format\n" );
    }

    message = "Do you want to process another request?";
    System.out.println( message );

    input = keyboard.nextLine();
    if ( input.length()>0 && (input.charAt( 0 ) == 'N' || input.charAt( 0 ) == 'n'))
    {
        anotherRequest = false;

    }
```

```
    return anotherRequest;
}

/**
 * Method that handles the processing quake data by region
 * It prompts the user for a filename if the user wishes to save to a file then calls a method to process the request
 * @param str String: Input string received from the user for the region quake data search
 */
public static void processRRequest(String str)
{
    double minLat=0.0, maxLat=0.0, minLon=0.0, maxLon=0.0;
    String message = "";
    boolean validRequest = true;
    String fileName = "";                    // filename to use to store data if requested

    StringTokenizer request = new StringTokenizer( str , ",");

    try                          // used to catch processing errors
    {

        request.nextToken();    // skips the first token - The R
        minLat = Double.parseDouble(  request.nextToken());
        maxLat = Double.parseDouble(  request.nextToken());
        minLon = Double.parseDouble(  request.nextToken());
        maxLon = Double.parseDouble(  request.nextToken());

    }
    catch (NumberFormatException | NoSuchElementException e)
    {
        message =   "The attempt to search for records was not in the proper format, the correct format is:\n" +
                    "R, minLat, maxLat, minLon, maxLon\n";
        System.out.println( message );
        validRequest = false;
    }

    if (validRequest)
    {
        fileName = filenameRequest();                    // check if user wishes to save to a file

        if (fileName.isEmpty())                         // user did not enter filename - display to console
        {
            quakeData.regionSearch( minLat, maxLat, minLon, maxLon );
        }
        else
        {
            quakeData.regionSearch( minLat, maxLat, minLon, maxLon, fileName );
        }
    }

}

/**
 * Method that handles the processing quake data by date
 * It prompts the user for a filename if the user wishes to save to a file then calls a method to process the request
 * @param str String: Input string received from the user for the date quake data search
 */
public static void processDRequest(String str)
{
    Dates minDate = new Dates();
    Dates maxDate = new Dates();
```

```
    String message = "";
    boolean validRequest = true;
    String fileName = "";                    // filename to use to store data if requested

    StringTokenizer request = new StringTokenizer( str , ",");

    try                          // used to catch processing errors
    {
        // extracts the two dates in string format and sends them to the Dates constructor to create
        // Dates objects
        request.nextToken();    // skips the first token - The D
        minDate = new Dates(request.nextToken());
        maxDate = new Dates(request.nextToken());
    }
    catch (NumberFormatException | NoSuchElementException e)
    {
        message =   "The attempt to search for records was not in the proper format, the correct format is:\n" +
                    "D, minDate, maxDate\n";
        System.out.println( message );
        validRequest = false;
    }

    if (validRequest)
    {
        fileName = filenameRequest();                  // check if user wishes to save to a file

        if (fileName.isEmpty())                        // user did not enter filename - display to console
        {
            quakeData.dateSearch( minDate, maxDate );
        }
        else
        {
            quakeData.dateSearch( minDate, maxDate, fileName );
        }
    }

}

/**
 * Method that handles the processing quake data by magnitude
 * It prompts the user for a filename if the user wishes to save to a file then calls a method to process the request
 * @param str String: Input string received from the user for the magnitude quake data search
 */
public static void processMRequest(String str)
{
    double magnitude = 0.0;
    String message = "";
    boolean validRequest = true;
    String fileName = "";                    // filename to use to store data if requested

    try                          // used to catch processing errors
    {
        magnitude = Double.parseDouble( str.substring( 2 ) );       // gets the double after the "M,"
    }
    catch (NumberFormatException | NoSuchElementException e)
    {
        message =   "The attempt to search for records was not in the proper format, the correct format is:\n" +
                "M, minMag\n";
        System.out.println( message );
        validRequest = false;
```

```
        }

        if (validRequest)
        {
            fileName = filenameRequest();                  // check if user wishes to save to a file

            if (fileName.isEmpty())                        // user did not enter filename - display to console
            {
                quakeData.magnitudeSearch( magnitude );
            }
            else
            {
                quakeData.magnitudeSearch( magnitude, fileName );
            }
        }

    }

    /**
     * Method that asks the user about saving to a quake data to a file and gets the filename to do so
     * @return String: A filename to save the quake data to.  If the user wishes to display data to the console,
     *                 the return String will be empty
     */
    public static String filenameRequest()
    {
        String message, input;
        Scanner keyboard = new Scanner( System.in );

        message =   "Do you wish to save this data to a file?  If yes, enter the filename and press enter.\n" +
                    "Otherwise just press enter and the data will be displayed to the console.";
        System.out.println( message );
        input = keyboard.nextLine();

        return input;
    }

}
```

```java
/*******************************************************************************************
 *  This class handles the processing of a quake Record
 *
 *  It stores the date, time, latitude, longitude, richter and location is data types and numerous
 *  operations of the data records
 *
 *  CST 283 Programming Assignment 1
 *  @author Michael Clinesmith
 *******************************************************************************************/
import java.io.IOException;
import java.util.NoSuchElementException;
import java.util.StringTokenizer;

public class QuakeRecord
{
    private Dates date;
    private Times time;
    private double latitude, longitude, richter;
    private Locations location;

    /**
     * No-argument constructor creates a default location (at the north pole)
     */
    public QuakeRecord()
    {
        date = new Dates();
        time = new Times();
        latitude = 90.0;
        longitude = 0.0;
        richter = 0.0;
        location = new Locations();
    }

    /**
     * Constructor that takes a formatted string and uses it to create a QuakeRecord object
     * The string must be of the form:
     * yyyy-mm-ddThh:mm:ss.sssZ|(decimal from -90 to 90)|(decimal from -180 to 180)|(decimal from 4.0 to 10.0)|(string)
     *
     * if the string is not in the proper form, the constructor will throw an IOexception
     * @param stringRecord String: a String in the proper form
     */
    public QuakeRecord(String stringRecord) throws IOException
    {
        // work variables to parse data
        String dateAndTime, strLat, strLong, strRic;
        StringTokenizer stringData;

        try
        {
            stringData = new StringTokenizer( stringRecord, "|" );

            // Read all data from string
            dateAndTime = stringData.nextToken();
            strLat = stringData.nextToken();
            strLong = stringData.nextToken();
            strRic = stringData.nextToken();
            location = new Locations(stringData.nextToken());

            //process raw data as needed
            date = new Dates(dateAndTime.substring( 0, 10 ));
```

```
            time = new Times(dateAndTime.substring( 11, 23 ));
            latitude = Double.parseDouble( strLat );
            longitude = Double.parseDouble( strLong );
            richter = Double.parseDouble( strRic );
        }
        catch (NumberFormatException | NoSuchElementException e)
        {
            // give record default data then throw Exception
            date = new Dates();
            time = new Times();
            latitude = 90.0;
            longitude = 0.0;
            richter = 0.0;
            location = new Locations();

            throw new IOException( "Bad Data" );
        }

    }

    /**
     * Constructor that takes a QuakeRecord object and makes a deep copy of it
     * @param record
     */
    public QuakeRecord( QuakeRecord record)
    {
        date = new Dates(record.getDate());
        time = new Times(record.getTime());
        latitude = record.getLatitude();
        longitude = record.getLongitude();
        richter = record.getRichter();
        location = new Locations( record.getLocation() );
    }

    /**
     * Accessor method to get the Date object with the date
     * @return Dates: A new Dates object that contains the date of the quake
     */
    public Dates getDate()
    {
        return new Dates(date);
    }

    /**
     * Accessor method to get the Time object with the time
     * @return Times: A new Times object the contains the time of the quake
     */
    public Times getTime()
    {
        return new Times(time);
    }

    /**
     * Accessor method to get the latitude of the quake
     * @return double: The latitude of the quake
     */
    public double getLatitude()
    {
        return latitude;
    }
```

```
/**
 * Accessor method to get the longitude of the quake
 * @return double: The longitude of the quake
 */
public double getLongitude()
{
    return longitude;
}

/**
 * Accessor method to get the magnitude of the quake
 * @return double: The magnitude of the quake
 */
public double getRichter()
{
    return richter;
}

/**
 * Accessor method to get the string data about the location of the quake
 * @return String: The location of the quake
 */
public String getLocation()
{
    return location.getLocation();
}

/**
 * Accessor method to get the year of the quake
 * @return int: The year of the quake
 */
public int getYear()
{
    return date.getYear();
}

/**
 * Accessor method to get the month of the quake
 * @return int: The month of the quake
 */
public int getMonth()
{
    return date.getMonth();
}

/**
 * Accessor method to get the day of the quake
 * @return int: The day of the quake
 */
public int getDay()
{
    return date.getDay();
}

/**
 * Accessor method to get the hour of the quake
 * @return int: The hour of the quake
 */
public int getHour()
```

```
{
    return time.getHour();
}

/**
 * Accessor method to get the minute of the quake
 * @return int: The minute of the quake
 */
public int getMinute()
{
    return time.getMinute();
}

/**
 * Accessor method to get the second (including decimal) of the quake
 * @return double: The second (including decimal) of the quake
 */
public double getSecond()
{
    return time.getSecond();
}

/**
 * Mutator method to set the date of the quake
 * @param date Dates: the date of the quake
 */
public void setDate( Dates date )
{
    this.date = new Dates(date);
}

/**
 * Mutator method to set the time of the quake
 * @param time Times: the time of the quake
 */
public void setTime( Times time )
{
    this.time = new Times(time);
}

/**
 * Mutator method to set the latitude of the quake
 * @param latitude double: the latitude of the quake
 */
public void setLatitude( double latitude )
{
    this.latitude = latitude;
}

/**
 * Mutator method to set the longitude of the quake
 * @param longitude double: the longitude of the quake
 */
public void setLongitude( double longitude )
{
    this.longitude = longitude;
}

/**
 * Mutator method to set the magnitude of the quake
```

```
 * @param richter double: The magnitude of the quake
 */
public void setRichter( double richter )
{
    this.richter = richter;
}

/**
 * Mutator method to set a String description of the location of the quake
 * @param loc String: A string representing the location of the quake
 */
public void setLocation( String loc )
{
    location.setLocation(loc);
}

/**
 * Mutator method to set the year of the quake
 * @param year int: The year of the quake
 */
public void setYear( int year)
{
    date.setYear( year );
}

/**
 * Mutator method to set the month of the quake
 * @param month int: the month of the quake
 */

public void setMonth(int month)
{
    date.setMonth( month );
}

/**
 * Mutator method to set the day of the quake
 * @param day int: the day of the quake
 */
public void setDay(int day)
{
    date.setDay( day );
}

/**
 * Mutator method to set the hour of the quake
 * @param hour int: the hour of the quake
 */
public void setHour(int hour)
{
    time.setHour( hour );
}

/**
 * Mutator method to set the minute of the quake
 * @param minute int: the minute of the quake
 */
public void setMinute(int minute)
{
    time.setMinute( minute );
```

```
}

/**
 * Mutator method to set the second of the quake
 * @param second double: the second of the quake (can contain a decimal part)
 */
public void setSecond(double second)
{
    time.setSecond( second );
}

/**
 * Method to check if the data stored in a QuakeRecord is valid
 * The method does not check the data stored in Location for validity
 *
 * @return boolean: true if the data is valid, false otherwise
 */
public boolean isValid()
{
    boolean valid = true;

    if (!date.isValid() || !time.isValid())             // checks if date or time is invalid
    {
        valid = false;
    }
    else if( latitude>90 || latitude<-90 )              // checks in latitude is in range
    {
        valid = false;
    }
    else if( longitude>180 || longitude<-180)           // checks in longitude is in range
    {
        valid = false;
    }
    else if (richter>10.0 || richter<4.0)               // checks if richter value is in range
    {
        valid = false;
    }

    return valid;
}

/**
 * Method to check if the record passed as a parameter contains the same data as this record
 * @param record QuakeRecord: A record saved in a QuakeRecord object
 * @return boolean: True if the records contain the same data, false if not
 */
public boolean isEqual( QuakeRecord record)
{
    boolean equal=false;
    //  check if equivalent parts of the records are equal
    if (date.isEqual( record.getDate() ) && time.isEqual( record.getTime() ) && latitude == record.getLatitude()
            && longitude == record.getLongitude() && richter == record.getRichter() &&
            location.getLocation().equals( record.getLocation() ))
    {
        equal = true;
    }

    return equal;
}
```

```
    /**
     * Method to convert the data in the QuakeRecord object to a String
     * @return String: A string representing the QuakeRecord object information
     */
    @Override
    public String toString()
    {
        String str;

        str = "Date: " + date.toString() +
                "\tTime: " + time.toString() +
                "\tLatitude: " + latitude +
                "\tLongitude: " + longitude +
                "\tMagnitude: " + richter +
                "\tLocation: " + location.toString();
        return str;
    }

    /**
     * Method to convert the data in the QuakeRecord object to a String in the way the instructor prefers
     * @return String: A string representing the QuakeRecord object information
     */
    public String toModString()
    {
        String str;

        str = date.toModString() + " " +
                time.toModString() + ", " +
                latLongString() + ", \tMag: " +
                String.format( "%.1f" , richter) +
                "\t" + location.toString();

        return str;
    }

    /**
     * Method to create a displayable string representing the latitude and longitude in the QuakeRecord object
     * @return
     */
    public String latLongString()
    {
        String str;

        str = String.format( "(%.2f,%.2f)", latitude, longitude );
        return str;
    }
}
```

```
/*******************************************************************************************
 *  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 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
 *
 *  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
     */
```

```
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-1) % 12 + 1;
    }
    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;
    }

}
```

```
TESTING
- Program executed efficiently
- User interface intuitive and friendly
- Modular strategy utilized for code design
- Code well structured, documented, and easy to read & follow
- Minimal repetitive code repetition
ERROR CHECKING
- Checked for a non- 'R', 'D', 'M'
- Tested:
   M,3.0
   R,30,20,-120,-130
   R,100,110,200,190
   D,2016-02-31,2015-04-31
TEST CASES
--------------------------------------------------------------------------------
M,7.9
23 JAN 18 0932Z, (56.00,-149.17), Mag: 7.9, 280km SE of Kodiak; Alaska
:
14 quakes
:
27 FEB 10 0634Z, (-36.12,-72.90), Mag: 8.8, offshore Bio-Bio; Chile
--------------------------------------------------------------------------------
R,30,38,-140,-120
04 JAN 18 1040Z, (37.86,-122.26), Mag: 4.4, 2km SE of Berkeley; CA
:
23 quakes
:
07 JAN 10 1810Z, (37.48,-121.80), Mag: 4.1, San Francisco Bay area; California
--------------------------------------------------------------------------------
D,2015-05-02,2015-05-03
03 MAY 15 2350Z, (-5.63,151.71), Mag: 4.5, 153km SSW of Kokopo; Papua New Guinea
:
90 quakes
:
02 MAY 15 0011Z, (27.38,85.18), Mag: 4.0, 15km ESE of Hitura; Nepal
--------------------------------------------------------------------------------
```