```
1,Airothian Shedinn,Paladin,2,None,0,751,Golden dragonborn,Noble,Lawful Good,Michael,18,11,14,10,10,16,10,19,19
2,Thia,Wizard,2,None,0,693,High elf,Acolyte,Chaotic Good,Liberty,10,15,14,16,12,8,13,12,12
3,Andraste Nailo,Druidic,2,None,0,751,Wood elf,Hermit,Chaotic Good,Bethany,12,16,15,13,17,13,15,16,19
4,Dorum Konselgen,Fighter,3,Eldrich Knight,3,1000,Orc,Bounty Hunter,Chaotic Good,Daniel,9,18,14,15,10,6,10,19,39
5,Mara,Rogue,2,None,0,764,Drow,Criminal,True Neutral,Liberty,7,20,14,12,16,14,13,16,17
6,Airothian Defender,Fighter,2,None,0,751,Golden dragonborn,Soldier,Lawful Good,Jonathan,17,14,15,11,13,11,11,16,20
7,Ravelle,Wild,2,None,0,751,Wood elf,None,Lawful Good,Autumn,11,14,15,12,16,15,13,12,14
8,Mei,Fighter,3,None,0,1116,Human,Folk hero,Lawful good,Suraya,14,16,15,11,13,9,13,14,32
9,Joe Average,None,0,None,0,0,Human,Peasant,True neutral,Michael,10,10,10,10,10,10,10,10,3
10,Jane Average,None,0,None,0,0,Human,Peasant,True neutral,Michael,9,11,9,10,10,11,10,10,2
11,Junky Person,Junk,0,None,0,0,Junk,Junk,True neutral,Michael,10,10,10,10,10,10,10,10,3
12,Someone,Fighter,1,Thinker,1,1,Human,The Void,Unknown,Michael,16,14,12,8,6,4,8,12,11
```

30/30 points for Program 3

Checked:
- Basic list item class
- Array of objects
- List algorithms
- GUI to manage algorithms
- File output

Pretty awesome solution. Well above and beyond the call of duty.

GUI very cool. Edit features nicely designed and flowed smoothly. Good use of CSS.

Ha, it did take me a bit of time to test the “Add” feature. Still, pretty cool.

```
/***************************************************************************************************
 *  This class handles the processing of a character Record
 *
 *  It stores the 20 individual items for each record.  The fields are stored as strings, but
 *  certain fields are flagged to be sorted as integers
 *
 *  CST 283 Programming Assignment 3
 *  @author Michael Clinesmith
 **************************************************************************************************/
import javafx.scene.control.Alert;

import java.io.IOException;
import java.util.NoSuchElementException;
import java.util.StringTokenizer;

public class CharacterRecord    √
{
    static final private int NUM_OF_FIELDS = 20;

    static final private String[] FIELD_LABELS = {"Key", "Character Name", "Class", "Level", "2nd Class",
            "2nd Level", "Experience Points", "Race", "Background", "Alignment",
            "Player Name", "Strength", "Dexterity", "Constitution", "Intelligence",
            "Wisdom", "Charisma", "Perception", "Armor Class", "Max Hit Points"};
    static final private String[] FIELD_SHORT = {"Key", "Name", "Class", "Level", "Clas2",
            "Lev2", "Exp", "Race", "Back", "Align",
            "Owner", "Str", "Dex", "Con", "Int",
            "Wis", "Cha", "Per", "AC", "MaxHP"};
    static final private Boolean[] FIELD_INT = {true, false, false, true, false,
            true, true, false, false, false,
            false, true, true, true, true,
            true, true, true, true, true};

    static private int nextKey=1; // used to generate new keys for new records
    static private int[] fieldLongest= new int[NUM_OF_FIELDS];  // keeps track of longest record in each field for formatting
    static private boolean fieldLongestInitialized=false; // used to initialize field lengths when first created
    private String[] record= new String[NUM_OF_FIELDS];

    /**
     * No-argument constructor creates an empty record with key value of nextKey, which will be unique when no
     * created records have a larger value for a key
     */
    public CharacterRecord()
    {
        record[0]=Integer.toString( nextKey );  // gives the record a new key
        nextKey++;
        for(int i=1; i<NUM_OF_FIELDS; i++)
        {
            if(FIELD_INT[i])    // empty record has different values based on an integer or String field
            {
                record[i]="0";
            }
            else
            {
                record[i]="None";
            }
        }
        if(!fieldLongestInitialized)
        {
            initializeFieldLongest();
        }
    }

    /**
     * Constructor that takes a formatted string where fields are separated by commas and uses it to create
     *      a CharacterRecord object
     * The String should have NUM_OF_FIELDS=20 fields
     * The key value will be checked and nextKey will be set to a larger value than the key value
     *
     * if the string is short of data, the constructor will throw an IOexception
     * @param stringRecord String: a String in the proper form
     */
    public CharacterRecord(String stringRecord) throws IOException
    {
        StringTokenizer stringData;
        int i=0, keyValue;

        if(!fieldLongestInitialized)
        {
            initializeFieldLongest();
        }

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

            for(i=0; i<NUM_OF_FIELDS; i++)
            {
                record[i]=stringData.nextToken();
                if(record[i].length()>fieldLongest[i])
                {
                    fieldLongest[i]=record[i].length();
                }
            }
        }
        catch (NoSuchElementException e)    // if string does not have enough data
        {
            if (i==0) // blank record - throw exceptions and do not create a record
            {
                String message = "There was a blank record in the file.\n" +
                        "This record will be ignored";

                Alert alert = new Alert( Alert.AlertType.ERROR );
                alert.setTitle( "Blank Record" );
                alert.setContentText( message );
                alert.showAndWait();

                throw new IOException("Blank Record");
            }

            for(; i<NUM_OF_FIELDS; i++)     // start at the current value of i
            {
                if(FIELD_INT[i])    // empty record has different values based on an integer or String field
                {
                    record[i]="0";
                }
                else
                {
                    record[i]="None";
                }
            }
            String message = "There was an error processing a record in the file.\n" +
                    "The record is incomplete.";

            Alert alert = new Alert( Alert.AlertType.ERROR );
            alert.setTitle( "Incomplete Data" );
            alert.setContentText( message );
            alert.showAndWait();
        }

        // check key and set next key to be bigger
        try
        {
            keyValue = Integer.parseInt( record[0] );
            if (keyValue>=nextKey)
            {
                nextKey = keyValue + 1;
            }
        }
        catch (NumberFormatException e)
        {
            // do nothing, just catch exception to continue
        }
    }

    /**
     * Constructor record that makes a deep copy of a CharacterRecord object
     * @param charRecord CharacterRecord: The object to make a copy of
     */
    public CharacterRecord( CharacterRecord charRecord)
    {
        for (int i=0; i<getNumOfFields(); i++)
        {
            record[i]=charRecord.getFieldValue(i);
        }
        updateFieldLongest( charRecord );
    }

    /**
     * Constructor record that creates a CharacterRecord with the given key
     * with defaults=true, the other fields are set to "0" or "None"
     * with defaults=false, the other fields are set to ""
     *
     * @param key String: Value used to set the CharacterRecord key
     * @return CharacterRecord: A CharacterRecord with a key and otherwise default or blank fields
     */
    public CharacterRecord( String key, boolean defaults)
    {
        record[0]=key;
        if (defaults)
        {
            for(int i=1; i<NUM_OF_FIELDS; i++)
            {
                if(FIELD_INT[i])    // empty record has different values based on an integer or String field
                {
                    record[i]="0";
                }
                else
                {
                    record[i]="None";
                }
            }
        }
        else
        {
            for(int i=1; i<NUM_OF_FIELDS; i++)
            {

                    record[i]="";
            }
        }
    }

    /**
     * Accessor method to get the number of fields in a record
     * @return int: The number of fields in a record (20)
     */
    public static int getNumOfFields()
    {
        return NUM_OF_FIELDS;
    }

    /**
     * Atatic ccessor method to get the field label for a position
     * @param index int: The position in the FIELD_LABELS array
     * @return String: The label for the index position
     */
    public static String getFieldLabel(int index)
    {
        String label="Out of Bounds";
        if(index>=0 && index<NUM_OF_FIELDS)
        {
            label = FIELD_LABELS[index];
        }
        return label;
    }

    /**
     * Static accessor method to get the shortened field label for a position
     * @param index int: The position in the FIELD_LABELS array
     * @return String: The shorter label for the index position
     */
    public static String getFieldShort(int index)
    {
        String label="Out of Bounds";
        if(index>=0 && index<NUM_OF_FIELDS)
        {
            label = FIELD_SHORT[index];
        }
        return label;
    }

    /**
     * Static accessor method to get the next available valid key
     * The key should be bigger than that in all the records currently created
     *
     * @return String: The next available key
     */
    public static String getNextKey()
    {
        return Integer.toString( nextKey );
    }

    /**
     * Accessor method to get the value of a position in the record
     * @param index int: Index position in the record to get the value of
     * @return String: The value at that position
     */
    public String getFieldValue(int index)
    {
        String label="Out of Bounds";
        if(index>=0 && index<NUM_OF_FIELDS)
        {
            label = record[index];
        }
        return label;
    }

    public static boolean isFieldTypeInt(int index)
    {
        boolean isInt=false;
        if(index>=0 && index<=getNumOfFields())
        {
            isInt = FIELD_INT[index];
        }
        return isInt;
    }

    /**
     * Accessor method to get the key of a record - in position 0
     * @return String: The key of a record
     */
    public String getKey()
    {
        return record[0];
    }

    /**
     * Method that will search to find if the given label is one of the fields in a record and returns that index
     * @param label String: A label to search for in the field label constant arrays
     * @return int: The index of that label if found, otherwise -1
     */
    public static int getLabelIndex(String label)
    {
        int index=-1;
        boolean found = false;

        for(int i=0; i<NUM_OF_FIELDS && !found; i++)
        {
            // check if label in label lists
            if(label.equals( FIELD_LABELS[i] ) || label.equals( FIELD_SHORT[i] ))
            {
                index=i;
                found=true;
            }
        }
        return index;
    }

    /**
     * Mutator method to set the value of a position in the record
     * @param index int: The index position in the record to set
     * @param str String: The value to set at that position
     * @return boolean: Returns true if a value was set, false otherwise
     */
    public boolean setFieldValue(int index, String str)
    {
        boolean setValue=false;

        if(index>=0 && index<NUM_OF_FIELDS)
        {
            record[index]= str;
            setValue = true;
        }
        return setValue;
    }

    /**
     * Mutator method to set the value of a position in the record
     * @param fieldLabel String: A String representing the field label of the position to set
     * @param str String: The value to set at that position
     * @return boolean: Returns true if a value was set, false otherwise
     */
    public boolean setFieldValue(String fieldLabel, String str)
    {
        boolean setValue=false;
        int index=getLabelIndex( fieldLabel );

        if(index>=0 && index<NUM_OF_FIELDS)
        {
            record[index]= str;
            setValue = true;
        }
        return setValue;
    }

    /**
     * method to adjust the longest field lengths when a record is encountered.
     * @param rec The record check to update longest field lengths
     */
    public static void updateFieldLongest(CharacterRecord rec)
    {
        String str="";                      // used to get field values to check for length
        for (int i=0; i<NUM_OF_FIELDS; i++)
        {
            str = rec.getFieldValue( i );
            if(str.length()>fieldLongest[i])
            {
                fieldLongest[i]=str.length();
            }
        }
    }

    /**
     * Method that initializes the fieldLongest array - which keeps track of the longest
     */
    public static void initializeFieldLongest()
    {
        for (int i=0; i<NUM_OF_FIELDS; i++)
        {
            fieldLongest[i]=FIELD_SHORT[i].length();
        }
        fieldLongestInitialized=true;
    }

    /**
     * Method that resets the length of longest field records and nextKey field
     * when dealing with new CharacterData
     */
    public static void resetRecordFields()
    {
        initializeFieldLongest();
        nextKey=1;
    }

    /**
     * Method to convert the data in the CharacterRecord object to a String
     * @return String: A string representing the CharacterRecord object information
     */
    @Override
    public String toString()
    {
        String str=record[0];

        for (int i=0; i<NUM_OF_FIELDS-1; i++)
        {
            str += "," + record[i + 1];
        }

        return str;
    }

    /**
     * Method to convert the data in the CharacterRecord object to a String, formatted so that each field is
     * the same length as the longest item in the field for easier reading
     * @return String: A formatted, easier to read string representing the CharacterRecord object information
     */
    public String toStringFormatted()
    {
        String str= String.format( "%"+fieldLongest[0]+"s", record[0] );
        for (int i=0; i<NUM_OF_FIELDS-1; i++)
        {
            str += "," + String.format( "%"+fieldLongest[i+1]+"s",  record[i + 1]);
        }
        return str;
    }

    /**
     * Method to convert the data in the CharacterRecord object to a String, formatted so that each field is
     * the same length as the longest item in the field for easier reading
     * @return String: A formatted, easier to read string representing the CharacterRecord object information
     */
    public static String toStringLabelsFormatted()
    {
        if(!fieldLongestInitialized)
        {
            initializeFieldLongest();
        }

        String str= String.format( "%"+fieldLongest[0]+"s", FIELD_SHORT[0] );
        for (int i=0; i<NUM_OF_FIELDS-1; i++)
```

```
        {
            str += "," + String.format( "%"+fieldLongest[i+1]+"s",  FIELD_SHORT[i + 1]);
        }
        return str;
    }

}
```

```
/*****************************************************************************************
 *  This class contains the main driver and interface for viewing character data
 *
 *  The user can load and save the data into files
 *  The user can add, modify and delete records
 *  The user can sort any of the record fields ascending or descending and randomize them
 *
 *  CST 283 Programming Assignment 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.GridPane;
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.io.IOException;
import java.io.PrintWriter;
import java.util.Optional;

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

    // holds data
    private CharacterData characterData;
    private String currentFileName="";

    // titlebox objects
    private BorderPane TitleBar;
    private Image DnDimage;
    private ImageView DnDimageView;
    private Button quitButton;
    private Label appLabel;
    private HBox quitButtonHBox;

    // Load/Save objects
    private VBox loadFileButtonVBox;
    private Button loadFileButton, saveFileButton;
    private File selectedFile;

    // Data sorting objects
    private Button addElementButton, sortButton, updateElementButton;

    private HBox sortHBox;

    // Data retreive record object
    private ComboBox<String> retreiveKeyListComboBox;
    private Label retreiveKeyLabel;
    private VBox retreiveKeyVBox;
    private Button retreiveBackButton;

    // Sorting objects
    private ComboBox<String> sortingFieldList;
    private RadioButton sortingAscending, sortingDescending, sortingRandom;
    private ToggleGroup sortingRadioToggle;
    private Label sortingLabel;
    private Button sortingButton;
    private VBox sortingContainer;

    // Data single record editing objects
    private GridPane oneRecordGridPane;
    private Label[] recordLabels=new Label[CharacterRecord.getNumOfFields()];
    private TextField[] recordValues=new TextField[CharacterRecord.getNumOfFields()];
    private Button saveRecordButton, exitRecordButton, deleteRecordButton;

    // 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( "DaD.css" );
        primaryStage.setScene( scene );
        primaryStage.setTitle( "Dungeons & Dragons Character Data Viewer" );
        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)
    {
        characterData = new CharacterData(  );

        initializeTitleBar();
        initializeLoadButton(primaryStage);
        initializeDataButtons();
        initializeDataTextArea();
        initializeSingleRecordArea();
        initializeRetrieveRecordObjects();
        initializeSortingObjects();

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

    /**
     * Method that creates the title bar that contains an image, title, and quit button
     */
    public void initializeTitleBar()
    {
        DnDimage=new Image("file:dndlogo.jpg");
        DnDimageView= new ImageView( DnDimage );

        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("Dungeons and Dragons Character Data Sorter");
        appLabel.setAlignment( Pos.CENTER );
        appLabel.setStyle( "-fx-font-size: 28; -fx-text-fill:  orange" );

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

    /**
     * Method that creates the Load and Save buttons
     * The actions of the load and save buttons are implemented in this method
     *
     * @param primaryStage
     */
    public void initializeLoadButton(Stage primaryStage)
    {
        FileChooser fileChooser = new FileChooser();

        loadFileButton = new Button("Load File");
        // code to load file
        loadFileButton.setOnAction(e -> {
            selectedFile = fileChooser.showOpenDialog(primaryStage);
            if (selectedFile!=null)
            {
                String message = "Do you want to load this file?\n" +
                        "This will erase any data already 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)
                {
                    characterData = new CharacterData( selectedFile.getName() );
                    currentFileName = selectedFile.getName();
                    characterDataDisplay.setText( characterData.toStringFormatted() );
                }
            }
        });
        loadFileButton.setPadding( new Insets( 20 ) );

        saveFileButton = new Button("Save File");
        // code to save file
        saveFileButton.setOnAction(e -> {
            selectedFile = fileChooser.showSaveDialog(primaryStage);
            if(selectedFile!=null)
            {
                String message = "Do you want to save this file?\n" +
                        "This will overwrite the data in the file.";
                Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
                alert.setTitle( "Save File?" );
                alert.setContentText( message );
                Optional<ButtonType> result = alert.showAndWait();
                if (result.get() == ButtonType.OK)
                {
                    try
                    {
                        PrintWriter outputFile = new PrintWriter(selectedFile.getName());
                        System.out.println( characterData.toString() );
                        System.out.println( characterData.toStringFormatted() );
                        // data gotten from saveDataString method and saves the filename in CharacterData
                        outputFile.println( characterData.saveDataString( selectedFile.getName() ) );
                        outputFile.close();
                        message = "File " + selectedFile.getName() +" saved.";
                        alert = new Alert( Alert.AlertType.INFORMATION );
                        alert.setTitle( "File Saved" );
                        alert.setContentText( message );
                        alert.showAndWait();

                    }
                    catch (IOException ex)
                    {
                        message = "There were problems with saving the data.\n" +
                                "The data was not saved.";
                        alert = new Alert( Alert.AlertType.ERROR );
                        alert.setTitle( "IOException Error" );
                        alert.setContentText( message );
                        alert.showAndWait();

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

        loadFileButtonVBox = new VBox(20, loadFileButton, saveFileButton);
        loadFileButtonVBox.setPadding( new Insets( 20 ) );
    }
```

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

    /**
     * Method that creates the buttons at the bottom of the interface to regulate data operations
     */
    public void initializeDataButtons()
    {
        //private RadioButton ascending, descending;
        //private ToggleGroup radioToggle;

        addElementButton = new Button( "Add Record" );
        addElementButton.setOnAction( new AppButtonHandler() );
        addElementButton.setAlignment( Pos.CENTER );
        addElementButton.setPadding( new Insets( 20 ) );

        sortButton = new Button( "Sort Records" );
        sortButton.setOnAction( new AppButtonHandler() );
        sortButton.setAlignment( Pos.CENTER );
        sortButton.setPadding( new Insets( 20 ) );

        updateElementButton = new Button( "Update/Delete Record" );
        updateElementButton.setOnAction( new AppButtonHandler() );
        updateElementButton.setAlignment( Pos.CENTER );
        updateElementButton.setPadding( new Insets( 20 ) );

        sortHBox = new HBox( 20, addElementButton, updateElementButton, sortButton);
        sortHBox.setPadding( new Insets( 20 ) );
        sortHBox.setAlignment( Pos.CENTER );
    }

    /**
     * Method that creates an interface for interacting with a single data object
     */
    public void initializeSingleRecordArea()
    {
        int halfRecords= CharacterRecord.getNumOfFields()/2;

        oneRecordGridPane = new GridPane();

        // create two columns of fields for records
        for (int i=0; i<halfRecords; i++)
        {
            recordLabels[i] = new Label(CharacterRecord.getFieldLabel( i ));
            recordValues[i] = new TextField();
            recordValues[i].setPadding( new Insets( 10 ) );
            oneRecordGridPane.add(recordLabels[i], 0, i+1);
            oneRecordGridPane.add(recordValues[i], 1, i+1);

        }
        for (int i=halfRecords; i<CharacterRecord.getNumOfFields(); i++)
        {
            recordLabels[i] = new Label(CharacterRecord.getFieldLabel( i ));
            recordValues[i] = new TextField();
            recordValues[i].setPadding( new Insets( 10 ) );
            oneRecordGridPane.add(recordLabels[i], 3, i-halfRecords+1);
            oneRecordGridPane.add(recordValues[i], 4, i-halfRecords+1);

        }

        recordValues[0].setDisable( true ); //disable key value from being directly edited

        //create single record buttons
        saveRecordButton = new Button( "Save Record" );
        saveRecordButton.setOnAction( new RecordButtonHandler() );
        saveRecordButton.setAlignment( Pos.CENTER );
        saveRecordButton.setPadding( new Insets( 10 ) );

        deleteRecordButton = new Button( "Delete Record" );
        deleteRecordButton.setOnAction( new RecordButtonHandler() );
        deleteRecordButton.setAlignment( Pos.CENTER );
        deleteRecordButton.setPadding( new Insets( 10 ) );

        exitRecordButton = new Button( "Exit Record" );
        exitRecordButton.setOnAction( new RecordButtonHandler() );
        exitRecordButton.setAlignment( Pos.CENTER );
        exitRecordButton.setPadding( new Insets( 10 ) );

        // add buttons to GridPane
        oneRecordGridPane.add(saveRecordButton, 1, halfRecords+2  );
        oneRecordGridPane.add (deleteRecordButton, 2, halfRecords+2);
        oneRecordGridPane.add(exitRecordButton, 4, halfRecords+2);

    }

    /**
     * Method that creates the interface to selecting a key for a record to view
     */
    public void initializeRetrieveRecordObjects()
    {
        retreiveKeyLabel = new Label("Select record key");
        retreiveKeyListComboBox = new ComboBox<>();
        retreiveKeyListComboBox.setOnAction( new RecordButtonHandler() );

        retreiveBackButton = new Button("Back");
        retreiveBackButton.setOnAction( new RecordButtonHandler() );
        retreiveBackButton.setAlignment( Pos.CENTER );
        retreiveBackButton.setPadding( new Insets( 10 ) );

        retreiveKeyVBox = new VBox(20, retreiveKeyLabel, retreiveKeyListComboBox, retreiveBackButton);
    }

    /**
     * Method that creates the interface to selecting a type of sort for the records
     */
    public void initializeSortingObjects()
    {
        sortingLabel = new Label("Select the type and field to sort");

        sortingFieldList = new ComboBox<String>( );
        sortingFieldList.setPadding( new Insets( 20 ) );
        for (int i=0; i<CharacterRecord.getNumOfFields(); i++)
        {
            sortingFieldList.getItems().add( CharacterRecord.getFieldLabel( i ) );
        }

        sortingRadioToggle = new ToggleGroup();

        sortingAscending = new RadioButton( "Ascending" );
        sortingAscending.setToggleGroup( sortingRadioToggle );
        sortingDescending = new RadioButton( "Descending" );
        sortingDescending.setToggleGroup( sortingRadioToggle );
        sortingRandom = new RadioButton( "Random" );
        sortingRandom.setToggleGroup( sortingRadioToggle );
        sortingRandom.setSelected( true );

        sortingButton = new Button( "Sort!" );
        sortingButton.setOnAction( new RecordButtonHandler() );
        sortingButton.setAlignment( Pos.CENTER );
        sortingButton.setPadding( new Insets( 20 ) );

        sortingContainer = new VBox( 20 , sortingLabel, sortingFieldList, sortingAscending, sortingDescending,
                sortingRandom, sortingButton);

    }

    /**
     * Method that resets the interface so it shows the CharacterData in the center display
     */
    public void restoreCharacterDisplay()
    {
        centerHBox.getChildren().clear();
        centerHBox.getChildren().add( characterDataDisplay );
        characterDataDisplay.setText( characterData.toStringFormatted() );

        // enable buttons when leaving one record menu
        sortHBox.setDisable( false );
        loadFileButtonVBox.setDisable( false );

    }

    /**
     * Method that locks the action buttons besides those in the center display
     */
    public void lockButtons()
    {
        sortHBox.setDisable( true );
        loadFileButtonVBox.setDisable( true );
    }

    /**
     * Class that handles ActionEvents for setting a new game or quitting
     */
    class AppButtonHandler implements EventHandler<ActionEvent>
    {
        /**
         * Method that handles ActionEvents for the new game and quit buttons
         * @param event ActionEvent: Event caused by clicking the new game and quit buttons
         */
        @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 );
            }
        }
        if (event.getSource()==addElementButton)
        {
            CharacterRecord newRecord = new CharacterRecord( );

            // confirm new key is not in characterData, otherwise try again
            while(characterData.isUsedKey(newRecord.getKey()))
            {
                newRecord = new CharacterRecord(  );
            }

            // set up interface to show one record panel
            centerHBox.getChildren().clear();
            centerHBox.getChildren().add( oneRecordGridPane );
            // disable buttons when in other menu
            lockButtons();

            recordValues[0].setText( newRecord.getFieldValue( 0 ) ); // place key value in field
        }
        if (event.getSource()==updateElementButton)
        {
            centerHBox.getChildren().clear();
            centerHBox.getChildren().add( retreiveKeyVBox );

            // disable buttons when in other menu
            lockButtons();

            retreiveKeyListComboBox.getItems().clear();
            characterData.updateComboBox( retreiveKeyListComboBox);
        }
        if (event.getSource()==sortButton)
        {
            centerHBox.getChildren().clear();
            centerHBox.getChildren().add( sortingContainer );

            // disable buttons when in other menu
            lockButtons();
        }
    }
}

/**
 * Class that handles ActionEvents for setting a new game or quitting
 */
class RecordButtonHandler implements EventHandler<ActionEvent>
{
```

```
/**
 * Method that handles ActionEvents for the new game and quit buttons
 *
 * @param event ActionEvent: Event caused by clicking the new game and quit buttons
 */
@Override
public void handle( ActionEvent event )
{
    String message;

    if (event.getSource() == exitRecordButton)   // user chooses to exit the record screen
    {
        message = "Do you want to go back to the main screen without editing the record?";

        Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
        alert.setTitle( "Exit Record" );
        alert.setContentText( message );
        Optional<ButtonType> result = alert.showAndWait();
        if (result.get() == ButtonType.OK)
        {
            restoreCharacterDisplay();
        }
    }
    else if (event.getSource() == retreiveBackButton)   // user chooses to exit the record screen
    {
            restoreCharacterDisplay();
    }
    else if(event.getSource()==saveRecordButton)
    {
        message = "Do you want to save this record?";

        Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
        alert.setTitle( "Save Record" );
        alert.setContentText( message );
        Optional<ButtonType> result = alert.showAndWait();
        if (result.get() == ButtonType.OK)
        {
            CharacterRecord rec = new CharacterRecord( recordValues[0].getText(), true );

            // put textfield data in record then clear them
            for(int i=1; i<CharacterRecord.getNumOfFields(); i++)
            {
                rec.setFieldValue( i,recordValues[i].getText());
                recordValues[i].setText("");
            }
            if (characterData.isUsedKey( rec.getKey() ))    // key already in characterData
            {
                characterData.updateRecord( rec );
            }
            else                                            // key not in characterData
            {
                characterData.addRecord( rec );
            }

            restoreCharacterDisplay();
        }
    }
    else if(event.getSource()==deleteRecordButton)
    {
        message = "Are you sure you want to delete the record with key " + recordValues[0].getText() +"?";
```

```
                Alert alert = new Alert( Alert.AlertType.CONFIRMATION );
                alert.setTitle( "Delete Record" );
                alert.setContentText( message );
                Optional<ButtonType> result = alert.showAndWait();
                if (result.get() == ButtonType.OK)
                {
                    characterData.deleteRecord( recordValues[0].getText() );

                    restoreCharacterDisplay();
                }

            }
            else if(event.getSource()==retreiveKeyListComboBox)
            {
                String key = retreiveKeyListComboBox.getValue();
                CharacterRecord rec = characterData.getRecordFromKey( key );

                // set up interface to show one record panel
                centerHBox.getChildren().clear();
                centerHBox.getChildren().add( oneRecordGridPane );
                // disable buttons when in other menu
                sortHBox.setDisable( true );
                loadFileButtonVBox.setDisable( true );

                // put record values into textfields
                for(int i=0; i<CharacterRecord.getNumOfFields(); i++)
                {
                    recordValues[i].setText( rec.getFieldValue( i ) );
                }
            }
            if (event.getSource()==sortingButton)
            {
                String key=sortingFieldList.getValue();

                if(sortingAscending.isSelected())
                {
                    characterData.sortRecords( key, true );
                }
                else if(sortingDescending.isSelected())
                {
                    characterData.sortRecords( key, false );
                }
                else if(sortingRandom.isSelected())
                {
                    characterData.sortRandom();
                }

                restoreCharacterDisplay();

            }
        }
    }
}
```

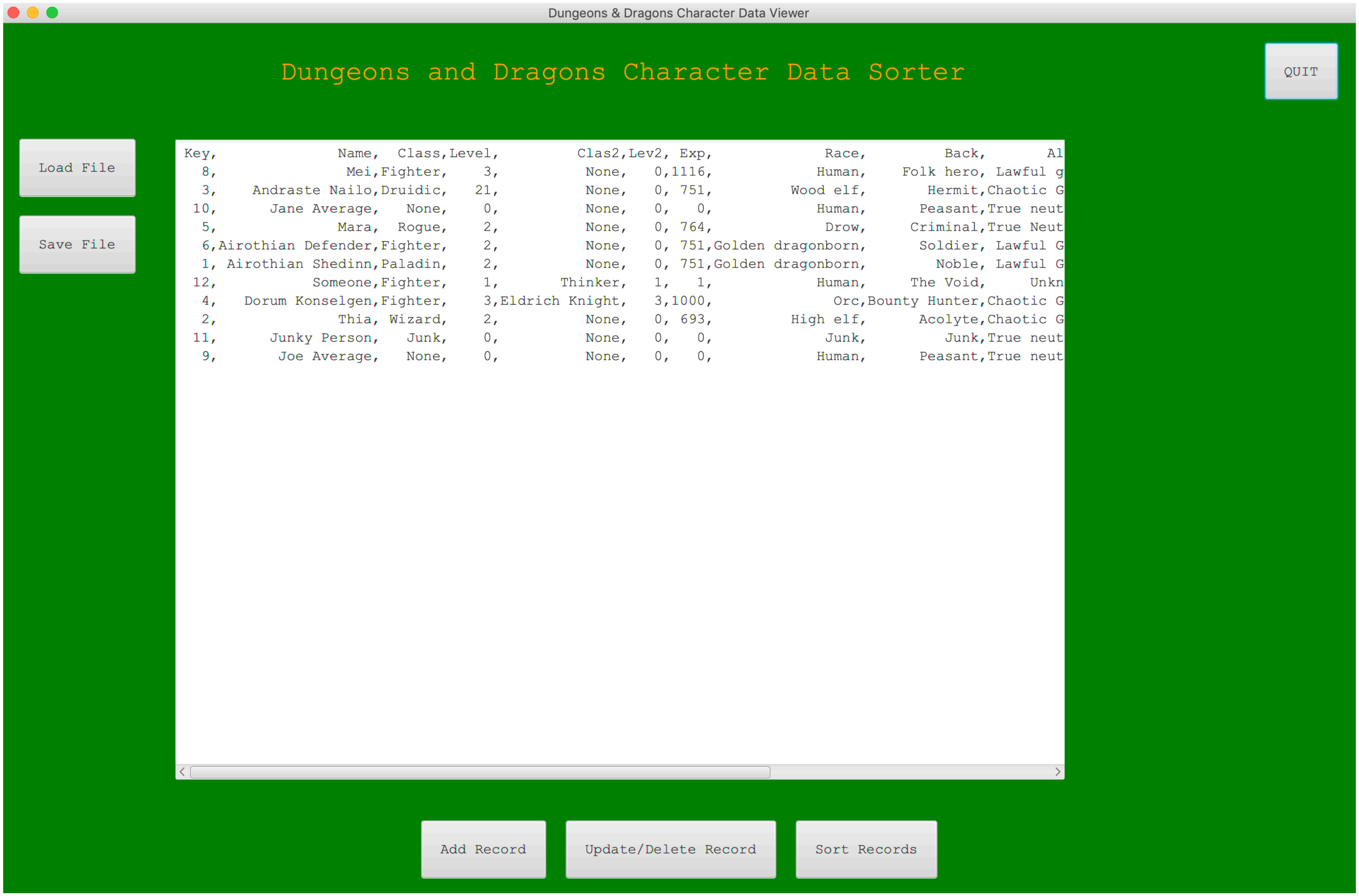
```
/*************************************************************************************************
 *  This class contains the character data
 *
 *  Note: Key values should be unique and the program will work to keep them that way, but there is
 *  nothing stopping someone creating a data file where the keys are not unique
 *
 *  The class implements sorting and randomizing the order of records
 *  The records can be viewed through the toString methods
 *
 *
 *
 *  CST 283 Programming Assignment 3
 *  @author Michael Clinesmith
 ************************************************************************************************/

import javafx.scene.control.Alert;
import javafx.scene.control.ComboBox;

import java.io.File;
import java.io.IOException;
import java.util.Random;
import java.util.Scanner;

public class CharacterData     √
{
    private final int MAX_RECORDS = 10000;
    private CharacterRecord[] records = new CharacterRecord[MAX_RECORDS];   √
    private int numOfRecords, numOfRecordsMissed;
    private String currentFile="";

    /**
     * No-argument constructor
     */
    public CharacterData()
    {
        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 character data
     * @param fileName String: the name of the file to open containing character data records
     */
    public CharacterData(String fileName)
    {
        String message;
        File characterData;                     // file that holds the character 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
        {
            characterData = new File(fileName);

            if(!characterData.exists())                 // end program if file data does not exist
            {

                message = "The file " + fileName + " does not exist for processing data.\n" +
                        "No data was uploaded.";
                Alert alert = new Alert( Alert.AlertType.ERROR );
                alert.setTitle( "File not found" );
                alert.setContentText( message );
                alert.showAndWait();

            }
            else
            {
                inputFile = new Scanner( characterData );

                // Read input file while more data exist
                // Read one line at a time (assuming each line contains one character record)
                while (inputFile.hasNext() && numOfRecords < MAX_RECORDS)
                {

                    inputLine = inputFile.nextLine();
                    try{
                        records[numOfRecords] = new CharacterRecord( inputLine );    // record processed in CharacterRecord class
                        numOfRecords++;
                    }
                    catch (IOException ex)
                    {
                        // blank line sent to create a record, so ignore
                    }

                }

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

                    Alert alert = new Alert( Alert.AlertType.ERROR );
                    alert.setTitle( "Too much data - Array Overflow" );
                    alert.setContentText( message );
                    alert.showAndWait();

                }
                while (inputFile.hasNext())
                {
                    inputLine = inputFile.nextLine();
                    numOfRecordsMissed++;                           // record not saved since no room
                }

                inputFile.close();
                currentFile=fileName;
            }
        }
        catch (IOException e)  // if error loading data, give error message
        {
            message = "There was an error encountered.  No data was loaded.";
            Alert alert = new Alert( Alert.AlertType.ERROR );
            alert.setTitle( "IOException Error" );
            alert.setContentText( message );
            alert.showAndWait();
        }
    }

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

    /**
     * Accessor method of getting the most recent filename used to load or save the file
     * @return String: The name of the file used to load or save the file
     */
    public String getCurrentFile()
    {
        return currentFile;
    }

    /**
     * Method that gets the record given a particular key
     * @param key String: The key to identify the record to be retrieved
     * @return CharacterRecord: A deep copy of the record searched for.  If not found, null is returned.
     */
    public CharacterRecord getRecordFromKey(String key)
    {
        CharacterRecord rec = null; // if no record found, return null value
        int index = getIndexFromKey( key );
        if(index>=0 && index<numOfRecords)
        {
            rec = new CharacterRecord( records[index]); // creates a deep copy of record
        }

        return rec;
    }

    /**
     * Method that gets the index of the record given a particular key
     * @param key String: The key to identify the record
     * @return int: the index of the record
```

```java
     */
    public int getIndexFromKey(String key)
    {
        boolean found=false;
        int index=-1;
        for(int i=0; i<numOfRecords & !found; i++)
        {
            if( key.equals( records[i].getKey()) )
            {
                found=true;
                index=i;
            }
        }
        return index;
    }

    /**
     * Method that gets the record given a particular index
     * @param index int: The index of the record in the data array
     * @return CharacterRecord: A deep copy of the record located at the index position in the data
     */
    public CharacterRecord getRecordFromIndex(int index)
    {
        CharacterRecord rec = null; // if no record found, return null value
        if(index>=0 && index<numOfRecords)
        {
            rec = new CharacterRecord( records[index]); // creates a deep copy of record
        }
        return rec;
    }

    /**
     * Method that adds a record to CharacterData
     * @param rec CharacterRecord: record to be added to CharacterData
     * @return boolean: true if the record was added, false if not (because of too many records)
     */
    public boolean addRecord( CharacterRecord rec)
    {
        boolean recordAdded=true;
        if(numOfRecords>=MAX_RECORDS)
        {
            recordAdded=false;
        }
        else
        {
            records[numOfRecords] = new CharacterRecord( rec ); // makes a deep copy
            numOfRecords++;
        }

        return recordAdded;
    }

    /**
     * Method that deletes a record from CharacterData
     * @param rec CharacterRecord: record to be deleted from CharacterData (only key is accessed)
     * @return boolean: true if the record was deleted, false if not
     */
    public boolean deleteRecord( CharacterRecord rec)
    {
        boolean recordDeleted=false;
        int index=getIndexFromKey( rec.getKey() );  // finds the index of the record from the rec's key
        if(index>=0 && index<numOfRecords)
        {
            //delete the record but keep other records in the same order
            for(int i=index; i<numOfRecords; i++)
            {
                records[i]=records[i+1]; // shallow copy moving of records - no risk of changing records from outside here
            }
            numOfRecords--;
            recordDeleted=true;
        }
        return recordDeleted;
    }

    /**
     * Method that deletes a record from CharacterData
     * @param key String: Key of the record to be deleted from CharacterData
     * @return boolean: true if the record was deleted, false if not
     */
    public boolean deleteRecord( String key)
    {
        boolean recordDeleted=false;
        int index=getIndexFromKey( key );  // finds the index of the record from the key
        if(index>=0 && index<numOfRecords)
        {
            //delete the record but keep other records in the same order
            for(int i=index; i<numOfRecords; i++)
            {
                records[i]=records[i+1]; // shallow copy moving of records - no risk of changing records from outside here
            }
            numOfRecords--;
            recordDeleted=true;
        }
        return recordDeleted;
    }

    /**
     * Method that updates a record from CharacterData
     * Note the method does not add the record if the key is not already in CharacterData
     * @param rec CharacterRecord: record to be updated in CharacterData
     * @return boolean: true if the record was updated, false if not
     */
    public boolean updateRecord( CharacterRecord rec)
    {
        boolean recordUpdated = false;
        int index=getIndexFromKey( rec.getKey() );  // finds the index of the record from the rec's key
        if(index>=0 && index<numOfRecords)
        {
            records[index]=new CharacterRecord( rec ); // makes a deep copy
            recordUpdated=true;
        }

        return recordUpdated;
    }

    /**
     * Method that resets the CharacterData object
     */
    public void resetCharacterData()
    {
        numOfRecords = 0;
        numOfRecordsMissed = 0;
        currentFile="";
        CharacterRecord.resetRecordFields();
    }

    /**
     * Method to check if a key has been used (Helps with giving unique keys)
     * @param key String: The key to check if it has been used already
     * @return boolean: True if it has been used, false if not
     */
    public boolean isUsedKey(String key)
    {
        boolean used=false;
        {
            for(int i=0; i<numOfRecords && !used; i++)
            {

                if(key.equals( records[i].getKey() ))
                {
                    used=true;
                }
            }
        }
        return used;
    }

    /**
     * Method to update a comboBox with all the keys in CharacterData
     * @param comboBox ComboBox: A ComboBox that needs the keys entered into it so they can be selected
     */
    public void updateComboBox( ComboBox<String> comboBox )
    {
        for (int i = 0; i<numOfRecords; i++)
        {
            comboBox.getItems().add(records[i].getKey());
        }
    }

    /**
     * Method that gets the data stored in the CharacterRecords and returns it as a String
     * it also updates the filename value
     * @param filename String: The name of the file to be used to store the CharacterRecords data
     * @return String: A string of CharacterRecords
     */
    public String saveDataString(String filename)
    {
        currentFile=filename;
        return toString();
    }

    /**
     * Method that sorts the records according to the field sent in fieldLabel
     * @param fieldLabel String: the field to sort
     * @param ascOrDes boolean: indicates if the sort is to be ascending (true) or descending(false)
     * @return boolean: true if the records were sorted, false if not
     */
    public boolean sortRecords(String fieldLabel, boolean ascOrDes)
    {
        int fieldIndex=CharacterRecord.getLabelIndex( fieldLabel );
        boolean sortedRecords=false;

        if (fieldIndex>=0 && fieldIndex<=CharacterRecord.getNumOfFields())  // index must be valid
        {
            sortedRecords=true;
            if (ascOrDes)                   // ascending sort
            {
                if (CharacterRecord.isFieldTypeInt( fieldIndex ))           // sort with ints
                {
                    sortRecordsAscInt(fieldIndex);
                }
                else                                                        // sort with strings
                {
                    sortRecordsAscString(fieldIndex);
                }
            }
            else                            // descending sort
            {
                if (CharacterRecord.isFieldTypeInt( fieldIndex ))           // sort with ints
                {
                    sortRecordsDesInt(fieldIndex);
                }
                else                                                        // sort with strings
                {
                    sortRecordsDesString(fieldIndex);

                }
            }
        }
        return sortedRecords;
    }

    /**
     * Method that sorts the records ascending in String order based in the field represented by fieldIndex
     * @param fieldIndex int: the field to sort
     */
    public void sortRecordsAscString(int fieldIndex)
    {
        // bubblesort modified from program shown in class
        int lastPos;
        int index;
        CharacterRecord temp;

        for (lastPos = numOfRecords - 1; lastPos >= 0; lastPos--)
        {
            for (index = 0; index <= lastPos - 1; index++)
            {
                if (records[index].getFieldValue( fieldIndex ).compareTo( records[index+1].getFieldValue( fieldIndex ) ) > 0)
                {
                    temp = records[index];
                    records[index] = records[index + 1];
                    records[index + 1] = temp;
                }
            }
        }
    }
    /**
     * Method that sorts the records descending in String order based in the field represented by fieldIndex
     * @param fieldIndex int: the field to sort
     */
```

```
public void sortRecordsDesString(int fieldIndex)
{
    // bubblesort modified from program shown in class
    int lastPos;
    int index;
    CharacterRecord temp;

    for (lastPos = numOfRecords - 1; lastPos >= 0; lastPos--)
    {
        for (index = 0; index <= lastPos - 1; index++)
        {
            if (records[index].getFieldValue( fieldIndex ).compareTo( records[index+1].getFieldValue( fieldIndex ) ) < 0)
            {
                temp = records[index];
                records[index] = records[index + 1];
                records[index + 1] = temp;
            }
        }
    }
}

/**
 * Method that sorts the records ascending in integer order based in the field represented by fieldIndex
 * @param fieldIndex int: the field to sort
 */
public void sortRecordsAscInt(int fieldIndex)
{
    // bubblesort modified from program shown in class
    int lastPos;
    int index;
    CharacterRecord temp;
    int thisValue, nextValue;

    try                 // watch for parsing errors
    {
        for (lastPos = numOfRecords - 1; lastPos >= 0; lastPos--)
        {
            for (index = 0; index <= lastPos - 1; index++)
            {
                thisValue = Integer.parseInt( records[index].getFieldValue( fieldIndex ) );
                nextValue = Integer.parseInt( records[index+1].getFieldValue( fieldIndex ) );

                if (thisValue>nextValue)
                {
                    temp = records[index];
                    records[index] = records[index + 1];
                    records[index + 1] = temp;
                }
            }
        }
    }
    catch(NumberFormatException e)  // not all value are integers, so format for strings
    {
        sortRecordsAscString( fieldIndex );
    }
}

/**
 * Method that sorts the records descending in integer order based in the field represented by fieldIndex
 * @param fieldIndex int: the field to sort
```

```
     */
    public void sortRecordsDesInt(int fieldIndex)
    {
        // bubblesort modified from program shown in class
        int lastPos;
        int index;
        CharacterRecord temp;
        int thisValue, nextValue;

        try                 // watch for parsing errors
        {
            for (lastPos = numOfRecords - 1; lastPos >= 0; lastPos--)
            {
                for (index = 0; index <= lastPos - 1; index++)
                {
                    thisValue = Integer.parseInt( records[index].getFieldValue( fieldIndex ) );
                    nextValue = Integer.parseInt( records[index+1].getFieldValue( fieldIndex ) );

                    if (thisValue<nextValue)
                    {
                        temp = records[index];
                        records[index] = records[index + 1];
                        records[index + 1] = temp;
                    }
                }
            }
        }
        catch(NumberFormatException e)  // not all value are integers, so format for strings
        {
            sortRecordsAscString( fieldIndex );
        }
    }

    public void sortRandom()
    {
        if (numOfRecords>0)
        {
            // create an array of random numbers then sort them along with records
            int[] randomValue= new int[numOfRecords];
            Random randomNumbers = new Random( );

            for(int i=0; i<numOfRecords; i++)
            {
                randomValue[i]=randomNumbers.nextInt();
            }

            // bubblesort modified from program shown in class
            int lastPos;
            int index;
            int temp;
            CharacterRecord tempRecord;

            for (lastPos = numOfRecords - 1; lastPos >= 0; lastPos--)
            {
                for (index = 0; index <= lastPos - 1; index++)
                {
                    if ( randomValue[index]<randomValue[index+1])
                    {
                        temp = randomValue[index];
                        randomValue[index] = randomValue[index+1];
                        randomValue[index+1]=temp;
                        tempRecord = records[index];
                        records[index] = records[index + 1];
                        records[index + 1] = tempRecord;
                    }
                }
            }

        }
    }

    /**
     * Method that gets the data stored in the CharacterRecords and returns it as a String
     * @return String: A string of CharacterRecords
     */
    @Override
    public String toString()
    {
        String str="";

        for(int i=0; i<numOfRecords-1; i++)
        {
            str += records[i].toString() + "\n";
        }
        if(numOfRecords>0)                      // if at least one record
        {
            str += records[numOfRecords-1];     // add last record without newline
        }

        return str;
    }

    /**
     * Method that gets the data stored in the CharacterRecords and saves it as a String
     * It formats the string so that there are labels and the fields line up evenly
     * @return String: A formatted string of CharacterRecords
     */
    public String toStringFormatted()
    {
        String str=CharacterRecord.toStringLabelsFormatted();

        for(int i=0; i<numOfRecords; i++)
        {
            str += "\n" + records[i].toStringFormatted();
        }

        return str;
    }


}
```

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

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

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



Dungeons and Dragons Character Data Sorter

```
Key,              Name,  Class,Level,         Clas2,Lev2, Exp,             Race,         Back,       Al
  8,               Mei,Fighter,    3,          None,   0,1116,            Human,    Folk hero, Lawful g
  3,    Andraste Nailo,Druidic,   21,          None,   0, 751,         Wood elf,       Hermit,Chaotic G
 10,      Jane Average,   None,    0,          None,   0,   0,            Human,      Peasant,True neut
  5,              Mara,  Rogue,    2,          None,   0, 764,             Drow,     Criminal,True Neut
  6,Airothian Defender,Fighter,    2,          None,   0, 751,Golden dragonborn,      Soldier, Lawful G
  1, Airothian Shedinn,Paladin,    2,          None,   0, 751,Golden dragonborn,        Noble, Lawful G
 12,           Someone,Fighter,    1,       Thinker,   1,   1,            Human,     The Void,     Unkn
  4,   Dorum Konselgen,Fighter,    3,Eldrich Knight,   3,1000,              Orc,Bounty Hunter,Chaotic G
  2,              Thia, Wizard,    2,          None,   0, 693,         High elf,      Acolyte,Chaotic G
 11,      Junky Person,   Junk,    0,          None,   0,   0,             Junk,         Junk,True neut
  9,       Joe Average,   None,    0,          None,   0,   0,            Human,      Peasant,True neut
```