Criterion C: Development

Technique:Validation (conditional statements/expressions)

Class: Session

Success Criteria
● The client can input fret numbers for individual strings and get a chord name ● The client can input a chord name and receive at least one set of fret numbers to play (one chord pattern) ● There is error management in the event of invalid input

Justification: Conditional statements such as if and else are helpful for validating input and displaying error messages in the event of invalid input.

Explanation: I used conditional statements to perform a presence check before following through with the executable function(“Conditionals”). I also used it to perform a range check by making sure that if a value was less that two digits, a zero would precede it in order for the concatenation further along to keep in alignment with a concatenated user input . This validation was essential for the accuracy of the searching functions (Drien).

Code sample:

```
while (chkinp)
{
System.out.println("G: ");
 s3 = readers.nextLine().trim();

    if (s3.equals("x"))
    {
     s3 = "xx";
     chkinp = false;
     break;
    }
    if(!s3.equals(""))
    {
        checkFret(s3);
    }
    else
    {
        System.out.println("Error! Please enter a fret number!");
    }
}
```

Technique: Java built-in classes
Class: Session and SessionController

Success Criteria
• The program automatically stores the date, time, and session number

Justification: Built-in classes were used to extract the current date and time.
Explanation: For the user to track their progress across a number of days, there was a need for the current date and time to be stored and displayed when accessed. In order to format the system time, the java.text package was used. This contains the necessary classes for converting dates to string format and representing times according the "conventions of the locale." ("The Java.text Package")
Code sample:

```
import java.util.*;
import java.io.*;
import java.text.*;
```

```
public static void getDate()
{
    Date now = new Date(); // get current date & time
    System.out.println( "\nCurrent date: " + now );

}
public static String dateToString(Date date)
{
    SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
    String output = dateFormat.format(date);
    return output;
}
```

Technique:Validation (while loop)
Class: Session

Success Criteria
• There is error management in the event of invalid input • Client can then easily return to current session and continue learning (refer to 1 and 2)

- The client should see a menu upon starting the program with appropriate options (“Start New Session”, “Session log”, “Recommendations”, “Help”, “Simple Search” etc.)

Justification: Using a while loop allows repeatability of certain commands until specific conditions are met. This helps improve validation processes since until a valid input is entered, for example, the error message will continue to display.
Explanation: The most prominent use was when user input was entered in the switch case and the display of the menu was based on breaking nested while loops to return to the required menus (Singh, Chaitanya).
Code sample:

```
boolean xdecisionloop = true;
    while (xdecisionloop)
    {
    System.out.println("What would you like help with? ");
    System.out.println("[ W ] Why should you use this program?");
    System.out.println("[ I ]  Guitar notation as used in this program");
    System.out.println("[ B ] Back to main menu ");
    char decision3 = IBIO.inputChar("\tEnter one letter for your option: ");
    decision3 = Character.toLowerCase(decision3);
    System.out.println();
    boolean simpldecisionloop = true;
        while (simpldecisionloop)
        {
            switch (decision3)
            {
                case 'w':
                displayWhy();
                simpldecisionloop = false;
                break;

                case 'i':
                displayNotation();
                simpldecisionloop = false;
                break;

                case 'b':
                System.out.println("Back to the main menu...");
                xdecisionloop = false;
                simpldecisionloop=false;
                break;

                default:
                System.out.println("Error! Please pick a valid option.");
                simpldecisionloop=false;
                break;

            }
        }

    }
```

Technique: Data Structures
Class: SessionController

Success Criteria
• The program automatically stores the date, time, and session number • All the input and relevant output are saved by the program in a log or history of sessions • The client is able to easily access this log and view previous sessions • The client can start a “New Session” for every new practice session • Client can then easily return to current session and continue learning (refer to 1 and 2)

Justification: Used to store data for the session log and data from new sessions.
Explanation: I chose arraylists as my primary data structure to hold and save data from the client’s sessions as well as the chords from my chords.txt file. Each element of the sessions arraylist contains date, session number, and session details (the chord that the user searched for)
Arraylists were chosen because of their size flexibility. Since my client wants a continually building log of data from sessions, using arraylists allows the addition of new elements without needing to define the size from the beginning. My implementation of this technique was through class study (Drien).
Code sample:

```
public static ArrayList<String> savedata = new ArrayList<String>();
public static ArrayList<String> sessionlog = new ArrayList<String>();
public static ArrayList<Session> sessiondet = new ArrayList<Session>();
```

Technique: Menus (switch-case)
Class: SessionController

Success Criteria
• There is error management in the event of invalid input • The client can start a “New Session” for every new practice session • Client can then easily return to current session and continue learning (refer to 1 and 2) • The client should see a menu upon starting the program with appropriate options (“Start New Session”, “Session log”, “Recommendations”, “Help”, “Simple Search” etc.)

Justification: Switch case asks the client to pick an option that -- if valid -- allows the execution of a certain function.

Explanation: I used nested switch-cases to create multiple sub-menus under the main menu switch case. This was done to allow multiple decisions to be made as the client utilised the program. For example, if the client pressed ‘s’ for simple search, the core functions of my program (the search by chord name and search by fret numbers functions) would be displayed as options for the client. This made navigation of program options easier for the client. This also allowed error management through default cases wherein an invalid option input by the user would result in an error message by default (Arora, 293).

Code sample:

```
boolean mainmenuloop = true;
    while (mainmenuloop)
    {   //main menu

            boolean decisionloop = true;
            System.out.println("\n\t== MENU ==");
            System.out.println("[ N ] New session");
            System.out.println("[ S ] Simple Search");
            System.out.println("[ L ] Log");
            System.out.println("[ H ] Help/How to use this program");
            System.out.println("[ Q ] Quit");
            char decision = IBIO.inputChar("\tEnter one letter for your option: ");
            decision = Character.toLowerCase(decision);
            System.out.println();
```

```
boolean simpdeciloop = true;
    while (simpdeciloop)
    {
        switch (decision1)
        {
            case 'm':
            findChord(chord,true);
            simpdeciloop = false;
            break;

            case 'c':
            //FIND CHORD THROUGH CHORD NAME
            Scanner reader = new Scanner(System.in);
            System.out.println("Enter a chord name: ");
            String s = reader.nextLine().trim();
            //System.out.println("You entered: "  + s);
            expandChord(chord, s, true);
            simpdeciloop =false;
            break;

            case 'e':
            System.out.println("Your session has ended. Check log for a history of your searches and sessions.");
            simpdeciloop = false;
            ndecisionloop = false;
            break;

            default:
            System.out.println("Error! Please pick a valid option.");
            simpdeciloop=false;
            break;

        }
    }
}
break;
```

Technique: Abstraction

Class: Session

Success Criteria
● The program automatically stores the date, time, and session number ● All the input and relevant output are saved by the program in a log or history of sessions ● The client is able to easily access this log and view previous sessions

Justification: The Session class was created in order to hold details about every session the client had on the learning tool.

Explanation: The Session class contains information used per element in arraylists that create new sessions for the user. Each element has a session number, date (and time), and session details. This data is stored in a text file and displayed in a Log. This provides a simple presentation of necessary details for the client without requiring the client to fully understand the technical workings of the objects. It provides necessary functionality with ease of user interaction (Arora, 2).

Code sample:

```
public class Session
{
    int sessionNumber;
    Date sessionDate;
    String sessionDetails;
```

Technique: Exception Handling (try-catch)

Class: SessionController

Success Criteria
● There is error management in the event of invalid input

Justification: Try/catch was used to handle invalid user inputs for the Manual Identification function

Explanation: Since the valid user inputs are not limited to integers (“x” is allowed), exception handling was required to prevent Java’s OutOfBounds error. Using try/catch as a custom type check improved the efficiency of handling invalid user input (“Java Exception Handling Examples”).

Code sample:

```
try

{
    int chk_i = Integer.parseInt(s6chk.trim());
    //System.out.println("INTEGER " + chk_i);
    return true;
}
catch (NumberFormatException e)
{
    //System.out.println("PANIC!");
    return false;
}
```

Technique: Encapsulation.

Class: Session

Success Criteria
• The client can start a “New Session” for every new practice session

Justification: Encapsulation protects base class attributes from external modification. It promotes interaction between methods and classes and allow changes to be made to the code independently.

Explanation: Encapsulation is used with accessors and mutators for the various class variables. It provides security and flexibility to the code and makes it easier to maintain data within objects. This technique was learnt in class (Drien).

Code sample:

```
public class Session
{
    int sessionNumber;
    Date sessionDate;
    String sessionDetails;

    public Session(int sessionNumber, Date sessionDate, String sessionDetails)
    {
        this.sessionNumber = sessionNumber;//counter;
        this.sessionDate = sessionDate;
        this.sessionDetails = sessionDetails;
    }

    public int getSessionNumber()
    {
        return sessionNumber;
    }
    public void setSessionNumber(int sessionNumber)
    {
        this.sessionNumber = sessionNumber;
    }
    public Date getSessionDate()
    {
        return sessionDate;
    }

    public void setSessionDate(Date sessionDate)
    {
        this.sessionDate = sessionDate;
    }
```

```
public String getSessionDetails()
{
    return sessionDetails;
}

public void setSessionDetails(String sessionDetails)
{
    this.sessionDetails = sessionDetails;
}
```

Technique: Files and permanent storage

Class: SessionController

Success Criteria
• The client can input fret numbers for individual strings and get a chord name • The client can input a chord name and receive at least one set of fret numbers to play (one chord pattern) • The program automatically stores the date, time, and session number • All the input and relevant output are saved by the program in a log or history of sessions • The client is able to easily access this log and view previous sessions

Justification: Files were used to hold chord details (chords.txt) and session log information, both of which are essential for searching for chords and keeping a record.
Explanation: I created a text file with individual chord names and fret numbers per string for the chords. These details are delimited by commas. The sessionlog.txt file is created when the client starts a new session for the first time. After this, the file updates data with searches made in a session and stores it permanently. This allows incremental session numbers even after the client has quit the program (Hill, 6.5).
Code sample:

```
public static void save(String concat) throws IOException
{
    File logFile= new File("sessionlog.txt");
    BufferedWriter bw = new BufferedWriter(new FileWriter( logFile, true));
    PrintWriter save = new PrintWriter(bw);
    save.println(concat);
    sessionlog.add(concat);
    save.close();
    //System.out.println("Data file saved.");
}
```

Technique: Data structure traversal
Class: Session

Success Criteria
• The client can input fret numbers for individual strings and get a chord name • The client can input a chord name and receive at least one set of fret numbers to play (one chord pattern)

Justification: For loops make traversing arrays and arraylists easier and thus help with searching for specific data within the data structures.
Explanation: I used multiple for loops throughout the program to traverse the **chords** file and compare it with user input to find chord details. I also used it in my isFound function to check whether certain chords existed in the **chords** file. I also used for loops when printing the session log onto the screen. The technique itself was learnt through lectures in school (Drien).
Code sample:

```
for (int i=0; i<chord.size();i++)
  {
      String z = chord.get(i);
      //System.out.println("Z: " + z);
      String delims = "[,]";
      String [] chordDetails = z.split(delims);
      //System.out.println(chordDetails[0]);
      if (name.equalsIgnoreCase(chordDetails[0]))
          {
              return true;
          }

  }
  System.out.println("Chord not found");
  return false;
```

Technique: Sorting
Class: SessionController

Success Criteria
• The client is able to easily access this log and view previous sessions • The log can be sorted according to session number for easier viewing

Justification: Sorting was used to make the log display more interactive and easy for the user to visit, especially after a large number of sessions have been saved.

Explanation: The sorting was done by session number per saved element of the arraylist of string session-log elements. For loops were used to sort both ascending and descending versions of the log (Drien).

Code sample:

```
System.out.println("Log sorted from newest session to oldest by session number and time.");
for(int rr=sessionlog.size()-1; rr>=0;rr--)
    {
        System.out.println(sessionlog.get(rr));
    }

System.out.println("Log sorted from oldest session to latest by session number and time.");
for (int y = 0; y<sessionlog.size(); y++)
    {
        System.out.println(sessionlog.get(y));
    }
```

Technique: Searching

Class: SessionController

Success Criteria
• The client can input fret numbers for individual strings and get a chord name • The client can input a chord name and receive at least one set of fret numbers to play (one chord pattern) • There is error management in the event of invalid input

Justification: Searching functions were chosen to output required values for the user such as fret numbers based on chord name and vice versa.

Explanation: Searches were performed on arraylists of strings compared with strings in text files. For example, when searching for a chord name to display the fret numbers, the first part of the first element (as separated by delimiters) was compared to the same in the text file. Traversals were used to conduct searches (Arora, 298).

Code sample:

```
for (int x=1; x<chordDetails.length; x++)
{
    chkChord = chkChord + chordDetails[x];
}

if (chkChord.equalsIgnoreCase(combined))
{
    System.out.println("You seem to be playing: " + chordDetails[0]);
    if (save ==true)
    {
    UpdateSession(chordDetails[0]);
    }
    return "Chord found!";
}
```

WORKS CITED:

Arora, Sumita. “Chapter 1: Concept of Objects.” *Computer Applications*, Sixth ed., Dhanpat Rai & Co., 2014, pp. 2–5.

“Conditionals.” *Learn Java - Interactive Java Tutorial*, www.learnjavaonline.org/en/Conditionals.

Dimitriou, Kostas, and Markos Hatzitaskos. "4.3 Introduction to Programming." *Core Computer Science: For the IB Diploma Program (International Baccalaureate)*. Newbury (Berkshire): Express, 2017. N. pag. Print.

Hill, Edward. "6.5.File Input." *Learning to Program Java*. Lincoln, NE: IUniverse, 2005. N. pag. Print.

Horstmann, Cay S. *Big Java: International Student Edition*. Chichester: John Wiley & Sons, 2010. Print.

Singh, Chaitanya, et al. “Java Exception Handling Examples.”*Beginnersbook.com*, 6 Sept. 2017, beginnersbook.com/2013/04/exception-handling-examples/.

Singh, Chaitanya. “While Loop in Java with Examples.” *Beginnersbook.com*, 10 Sept. 2017, beginnersbook.com/2015/03/while-loop-in-java-with-examples/.

"The Java.text Package." *Java: Fundamental Classes Reference*. N.p., n.d. Web.

“The Switch Statement.” *The Switch Statement (The Java™ Tutorials > Learning the Java Language > Language Basics)*, docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html.