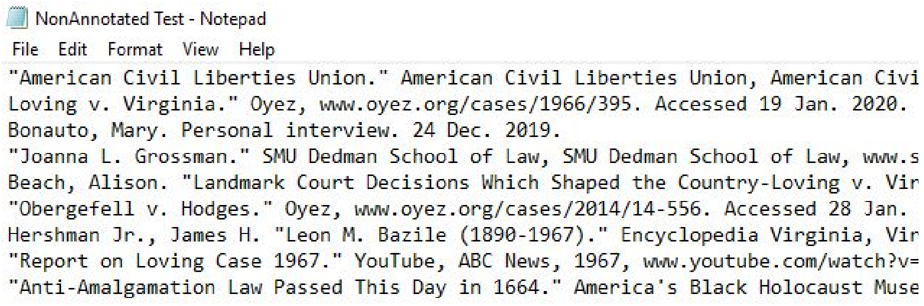
Criterion C: Product Development

Techniques Employed

1) Importing a .txt file into Java code and organizing it so it can later be formatted (with methods in the code)
2) Formatting sources of annotated bibliography from .txt file while not interfering with annotations (except for indenting them) (Algorithmic Complexity)
3) Alphabetizing each source from the .txt file that is imported into Java code (Algorithmic Complexity)
4) Add a hanging indent to each source from the .txt file that is imported into Java code (Algorithmic Complexity)
5) Graphical User Interface (GUI)

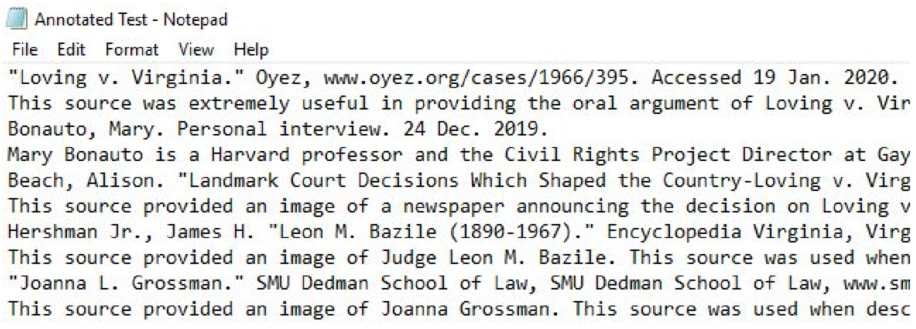
Important Note: This solution will format bibliographies in the standard MLA 8th edition format.

<u>1. Importing a .txt file into Java code and organizing it so it can be formatted (with methods in the code)</u>



```
"American Civil Liberties Union." American Civil Liberties Union, American Civi
Loving v. Virginia." Oyez, www.oyez.org/cases/1966/395. Accessed 19 Jan. 2020.
Bonauto, Mary. Personal interview. 24 Dec. 2019.
"Joanna L. Grossman." SMU Dedman School of Law, SMU Dedman School of Law, www.s
Beach, Alison. "Landmark Court Decisions Which Shaped the Country-Loving v. Vir
"Obergefell v. Hodges." Oyez, www.oyez.org/cases/2014/14-556. Accessed 28 Jan.
Hershman Jr., James H. "Leon M. Bazile (1890-1967)." Encyclopedia Virginia, Vir
"Report on Loving Case 1967." YouTube, ABC News, 1967, www.youtube.com/watch?v=
"Anti-Amalgamation Law Passed This Day in 1664." America's Black Holocaust Muse
```

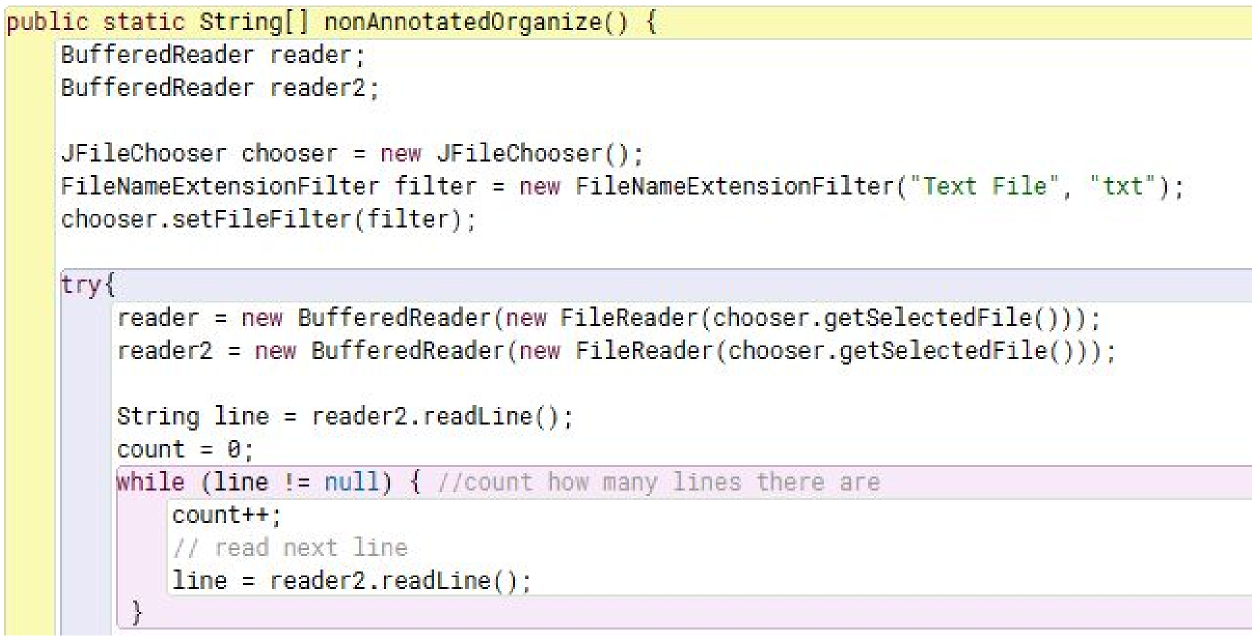
Figure 1: Example of a non-annotated .txt file that the user would prepare and later import into Java code.



```
"Loving v. Virginia." Oyez, www.oyez.org/cases/1966/395. Accessed 19 Jan. 2020.
This source was extremely useful in providing the oral argument of Loving v. Vir
Bonauto, Mary. Personal interview. 24 Dec. 2019.
Mary Bonauto is a Harvard professor and the Civil Rights Project Director at Gay
Beach, Alison. "Landmark Court Decisions Which Shaped the Country-Loving v. Virg
This source provided an image of a newspaper announcing the decision on Loving v
Hershman Jr., James H. "Leon M. Bazile (1890-1967)." Encyclopedia Virginia, Virg
This source provided an image of Judge Leon M. Bazile. This source was used when
"Joanna L. Grossman." SMU Dedman School of Law, SMU Dedman School of Law, www.sm
This source provided an image of Joanna Grossman. This source was used when desc
```

Figure 2: Example of an annotated .txt file that the user would prepare and later import into Java code.

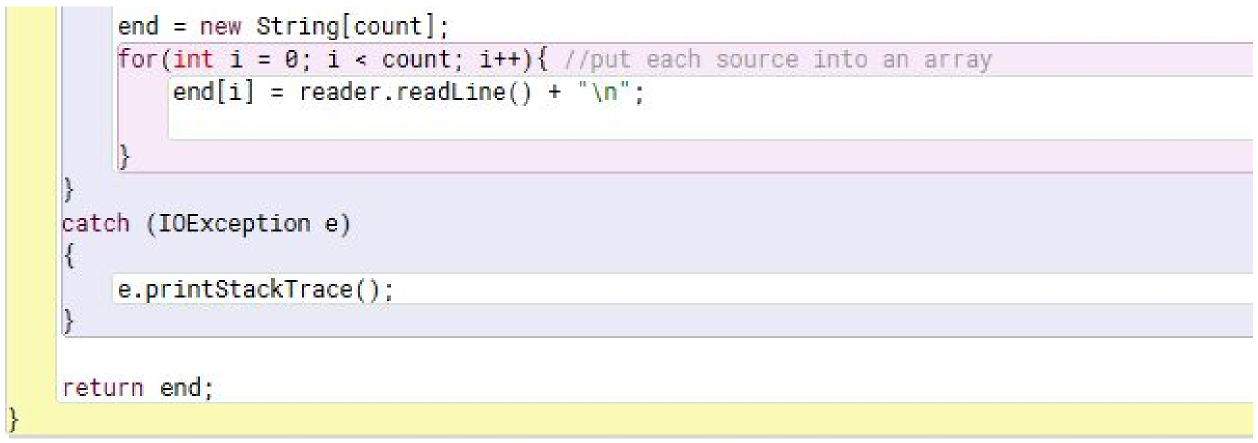
Before using the code, the user must prepare a .txt file with their unformatted bibliography. They can either do this by saving a Word Document as a .txt file or make one in Notepad. For non-annotated bibliographies, each source must be on a different line with no spaces in-between each line (as seen in figure 1). For annotated bibliographies, the annotations for each source must take up the line directly under the source that it is associated with. There also is no space in-between any of the lines (as seen in figure 2).



```
public static String[] nonAnnotatedOrganize() {
    BufferedReader reader;
    BufferedReader reader2;

    JFileChooser chooser = new JFileChooser();
    FileNameExtensionFilter filter = new FileNameExtensionFilter("Text File", "txt");
    chooser.setFileFilter(filter);

    try{
        reader = new BufferedReader(new FileReader(chooser.getSelectedFile()));
        reader2 = new BufferedReader(new FileReader(chooser.getSelectedFile()));

        String line = reader2.readLine();
        count = 0;
        while (line != null) { //count how many lines there are
            count++;
            // read next line
            line = reader2.readLine();
        }
```

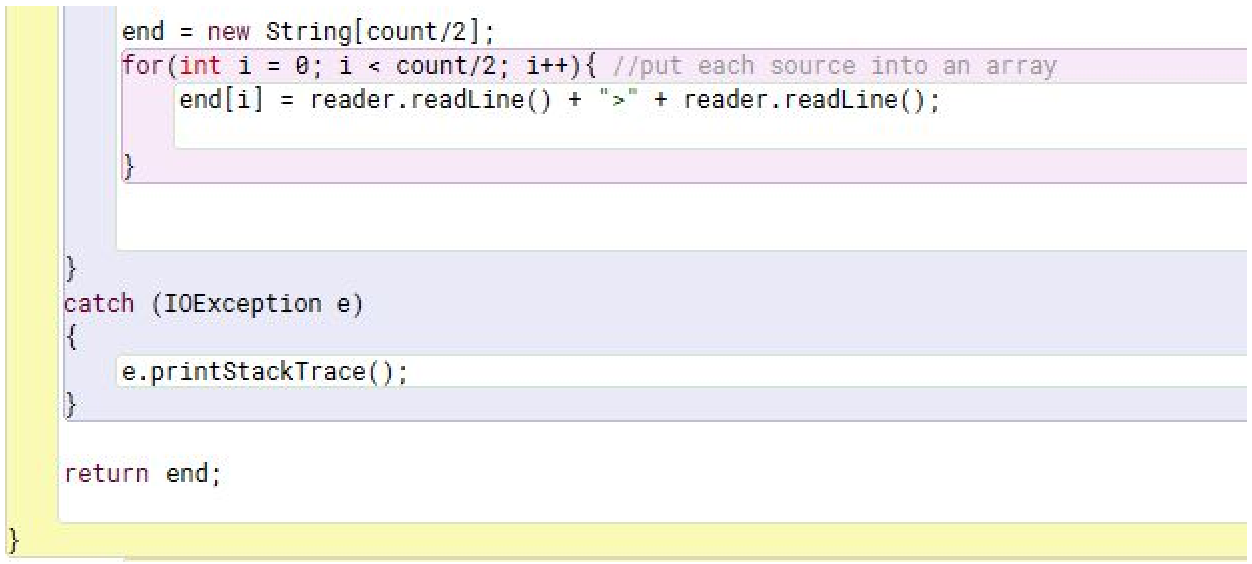
Figure 3: Segment of nonAnnotatedOrganize() method (same as start of annotatedOrganize()).

Once the user has prepared the bibliography that they want to format, one of the organize methods (**nonAnnotatedOrganize()** or **annotatedOrganize()**) will read each line and count how many lines there are. To do this, the **BufferedReader** class will be utilized. The **readLine()** method is used to count how many lines there are using a **while loop.** Once the user selects their .txt file (this will be explained more in-depth in the GUI section), two **BufferedReader** variables (**reader** and **reader2**) are declared as coming from the .txt file. After this, a **line** variable (String) is declared as the **reader2** variable with the **readLine()** method. This will then be put into a **while loop** which as long as a line exists, will add 1 to the **count** variable (int) thus counting how many lines there are in the .txt file.



```
        end = new String[count];
        for(int i = 0; i < count; i++){ //put each source into an array
            end[i] = reader.readLine() + "\n";

        }
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    return end;
}
```

Figure 4: Segment of nonAnnotatedOrganize() method.

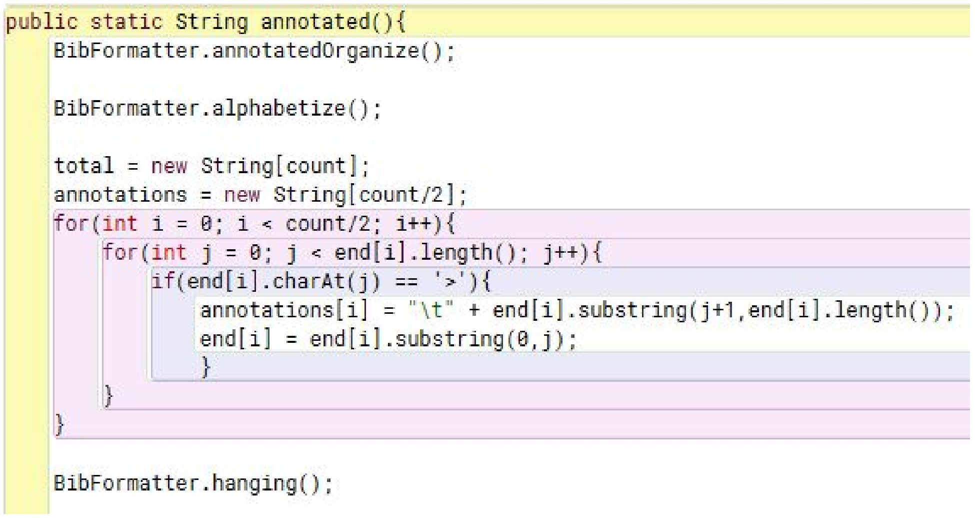


```
        end = new String[count/2];
        for(int i = 0; i < count/2; i++){ //put each source into an array
            end[i] = reader.readLine() + ">" + reader.readLine();

        }


    }
    catch (IOException e)
    {
        e.printStackTrace();
    }

    return end;

}
```

Figure 4: Segment of annotatedOrganize() method.

After each line is counted, the sources are put into an array titled **end**. If the bibliography is not annotated, each source will be put into an array (with the length of the number of sources there are (**count)**). If the bibliography is annotated, each source and their respective annotations will be put into an array (with the length of the number of sources there are (**count**/2)). The source and it's annotation will be separated with "\>" because they will need to later be separated and this will allow for a simple **for loop** to find the "\>" and identify where the source and annotation is (the source being everything left of "\>" and the annotation being everything right of "\>").

<u>Formatting sources of annotated bibliography from .txt file while not interfering with annotations (except for indenting them)</u>



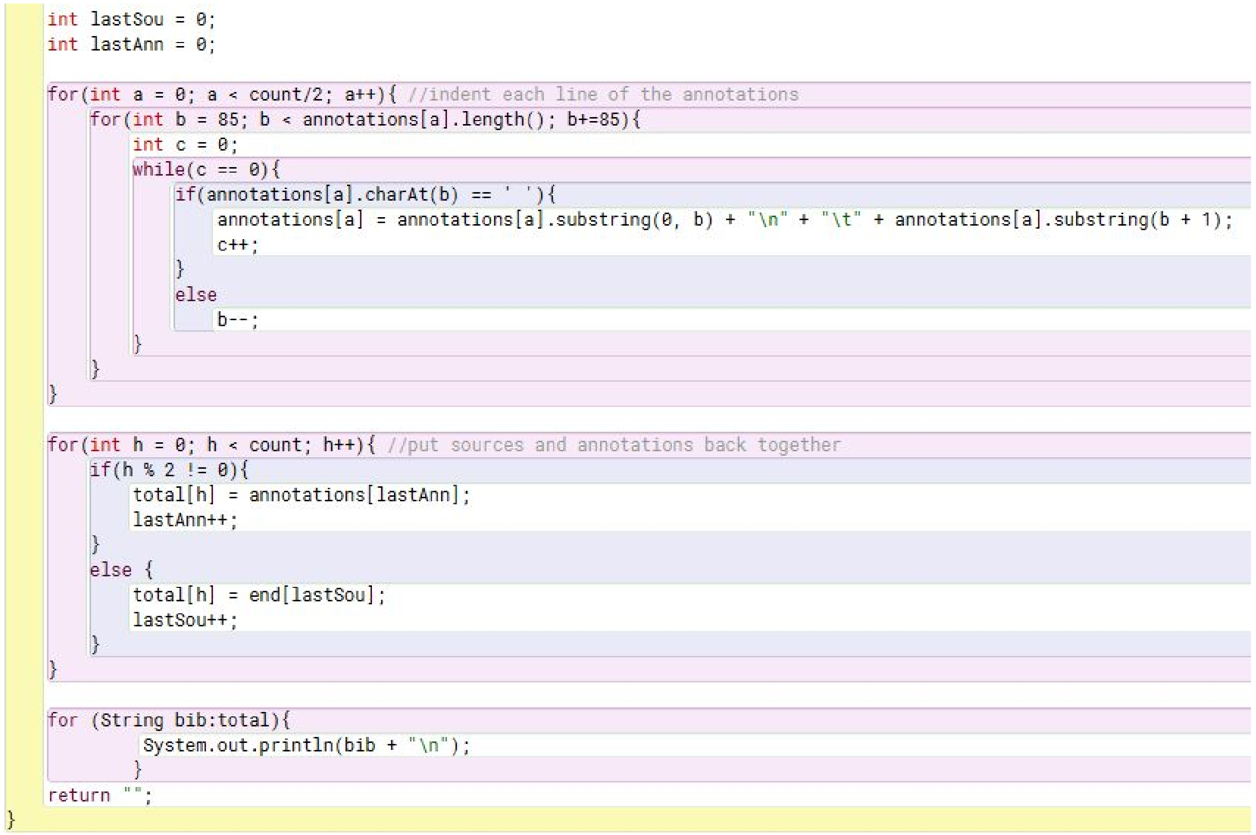
```
public static String annotated(){
    BibFormatter.annotatedOrganize();

    BibFormatter.alphabetize();

    total = new String[count];
    annotations = new String[count/2];
    for(int i = 0; i < count/2; i++){
        for(int j = 0; j < end[i].length(); j++){
            if(end[i].charAt(j) == '>'){
                annotations[i] = "\t" + end[i].substring(j+1,end[i].length());
                end[i] = end[i].substring(0,j);
                }
        }
    }

    BibFormatter.hanging();
```

Figure 5: Segment of annotated() method.

If a bibliography is annotated, it will go through the **alphabetize()** method with the sources and their annotations on one line separated with a ">" because the annotations must stay with their respective sources and the sources will be reordered after the **alphabetize()** method. Also, the annotations will not affect the alphabetization process because only the first few characters of each line (which belong to the sources) will be read.

After this, the sources and annotations will be separated because only the sources will have hanging indents applied to them. Before this, all of the sources has been in the **end array** with a length of **count**/2. Now a separate **array** with a length of **count**/2 will be declared and called **annotations.** A **for loop** will go through all of the spots in the **annotations array** and the **end array**. Inside that will be another **for loop** which will go through each **character** in a spot in the **end array** (it will go through each spot because of the first **for loop**). Once the position of the ">" added earlier is found, the annotation will be added to the **annotations array** at the spot determined by the first **for loop** and the source will be put into the **end array** at the spot determined by the first **for loop** (this will replace the source and annotation that was originally in that spot).

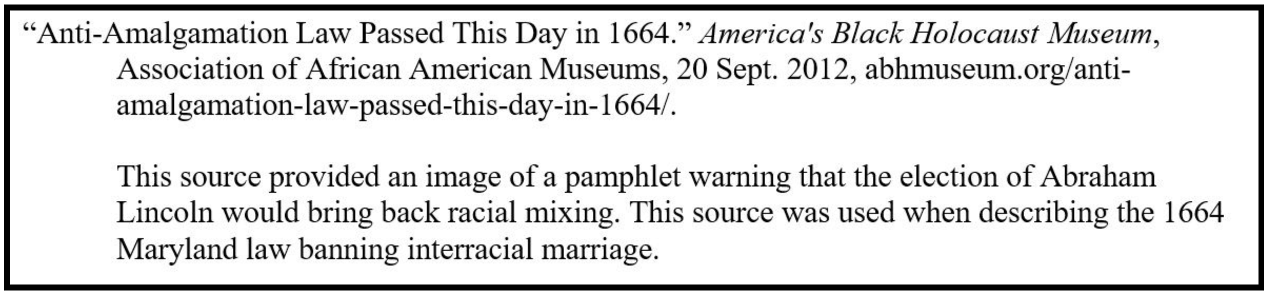
Now, the **hanging()** method will add hanging indents to each element in the **end array** (this is the sources only).



```
    int lastSou = 0;
    int lastAnn = 0;

    for(int a = 0; a < count/2; a++){ //indent each line of the annotations
        for(int b = 85; b < annotations[a].length(); b+=85){
            int c = 0;
            while(c == 0){
                if(annotations[a].charAt(b) == ' '){
                    annotations[a] = annotations[a].substring(0, b) + "\n" + "\t" + annotations[a].substring(b + 1);
                    c++;
                }
                else
                    b--;
            }
        }
    }

    for(int h = 0; h < count; h++){ //put sources and annotations back together
        if(h % 2 != 0){
            total[h] = annotations[lastAnn];
            lastAnn++;
        }
        else {
            total[h] = end[lastSou];
            lastSou++;
        }
    }

    for (String bib:total){
          System.out.println(bib + "\n");
        }
    return "";
}
```

Figure 6: Final segment of annotated() method.

The sources are formatted (alphabetized and have hanging indent), so now the annotations need to be formatted.



"Anti-Amalgamation Law Passed This Day in 1664." *America's Black Holocaust Museum*,
    Association of African American Museums, 20 Sept. 2012, abhmuseum.org/anti-
    amalgamation-law-passed-this-day-in-1664/.

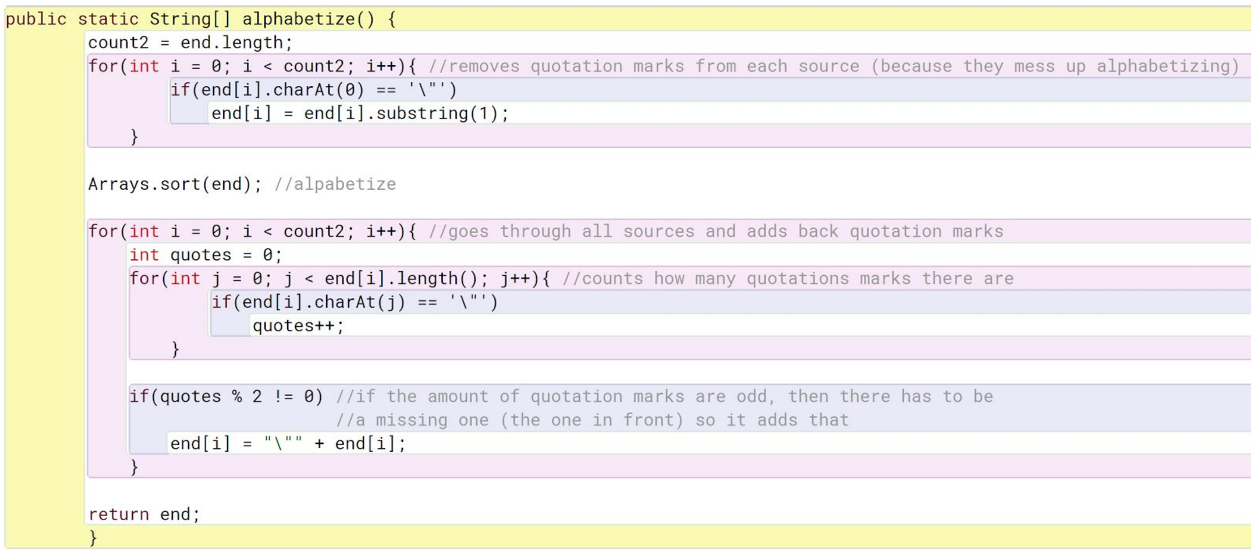
    This source provided an image of a pamphlet warning that the election of Abraham
    Lincoln would bring back racial mixing. This source was used when describing the 1664
    Maryland law banning interracial marriage.

Figure 7: Formatted annotated source.

As seen in Figure 7, each line of annotations is indented. The programming used to add these indents is the same as the one used in the **hanging()** method, so this be explained in the section explaining the **hanging()** method.

Finally, the sources and their annotations must be put back together. They are recombined into the **total array** which has a length of **count**. The annotations in **annotations** are in the same order as their respective sources in **end.** Because of this, the sources and annotations just need to alternate when being added to **total.** Sources and their annotations will be added back to separate spots in the array (with annotations being in the spot directly after their respective sources). There will be a **for loop** which will go through each spot in the **final array**. If that spot number is even, a source from **end** will be added to **total**. If the spot is odd, an annotation from **annotations** will be added to **total**. This is because the first spot in an **array** is identified as spot 0. The sources and annotations are already in the order that they are supposed to be added in, so once the first element of one of the **arrays** is added, there is a counter (**int lastSou** or **int lastAnn**) that will go up, so when the next element of that **array** will be added to **total**. For example, after the first source is added to **total**, **lastSou** will go up so the next source added to **total** is the second source.

<u>Alphabetizing each source from the .txt file that is imported into Java code</u>

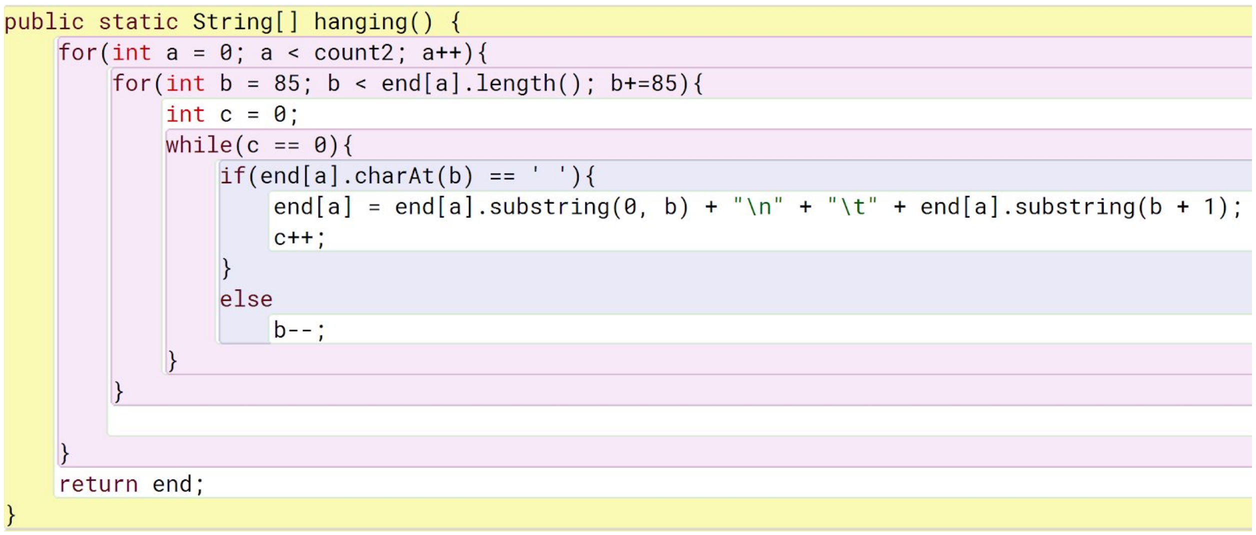


```
public static String[] alphabetize() {
        count2 = end.length;
        for(int i = 0; i < count2; i++){ //removes quotation marks from each source (because they mess up alphabetizing)
                if(end[i].charAt(0) == '\"')
                    end[i] = end[i].substring(1);
            }

        Arrays.sort(end); //alpabetize

        for(int i = 0; i < count2; i++){ //goes through all sources and adds back quotation marks
            int quotes = 0;
            for(int j = 0; j < end[i].length(); j++){ //counts how many quotations marks there are
                    if(end[i].charAt(j) == '\"')
                        quotes++;
                }

            if(quotes % 2 != 0) //if the amount of quotation marks are odd, then there has to be
                                //a missing one (the one in front) so it adds that
                end[i] = "\"" + end[i];
            }

        return end;
        }
```

Figure 8: alphabetize() method.

The sources (in the **end array**) will be alphabetized using the **Array.sort() method**. However, if a quotation mark is the first character in a source, the **Array.sort() method** will not properly alphabetize the sources. So, a **for loop** will remove the first quotation mark (if there is one), then the **Array.sort() method** will alphabetize the sources. After that, the first quotation mark needs to be added back to the sources that had them removed. To do this, there is a **for loop** that will go through each source and count how many quotation marks there are. If the number of quotations marks is odd, that means that the first one had to have been removed (because there should always be a start and end quotation mark (meaning the number of quotations marks should be even)) and it is added back to the start of the source.

<u>Add a hanging indent to each source from the .txt file that is imported into Java code</u>



```java
public static String[] hanging() {
    for(int a = 0; a < count2; a++){
        for(int b = 85; b < end[a].length(); b+=85){
            int c = 0;
            while(c == 0){
                if(end[a].charAt(b) == ' '){
                    end[a] = end[a].substring(0, b) + "\n" + "\t" + end[a].substring(b + 1);
                    c++;
                }
                else
                    b--;
            }
        }

    }
    return end;
}
```

Figure 9: hanging() method.

The **hanging method** adds a hanging indent to each source. As seen in Figure 7, a source is supposed to have an indent on every line other than the first one. The **hanging method** starts with a **for loop** that goes through each source. The most amount of characters that can be on a line with an indent (with Times New Roman 12 point font (the standard MLA format)) is 85. There is a second for loop that starts at 85 and counts the characters back from that until the first space in the first line (so the next line does not start in the middle of a word). Once the first space is found, a new line and tab is added. If there is another line that needs to have a hanging indent added, 85 is added to the second **for loop** and the same process will occur. This will repeat until all of the lines (other than the first one) has a hanging indent.

<u>Graphical User Interface (GUI)</u>

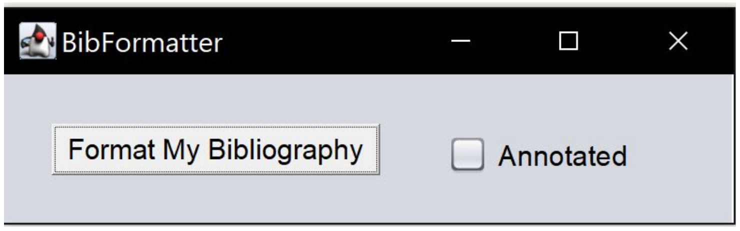


Figure 10: GUI.

The GUI allows users to easily select whether or not their bibliography is annotated then import their .txt file with their unformatted bibliography.

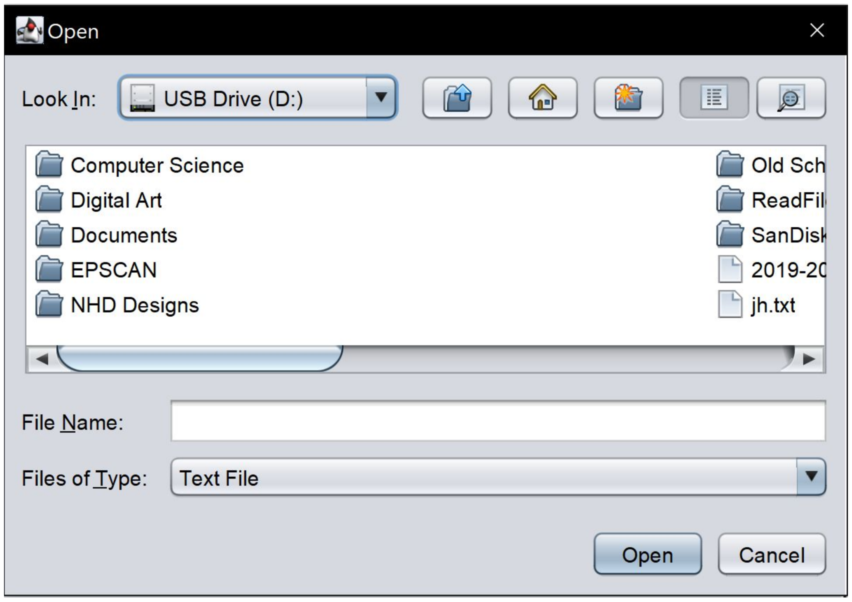


Figure 11: File selector (GUI).

When the user selects the ‘Format My Bibliography’ button, the file selector (as seen in Figure 11) opens. It only allows for .txt files to be selected and allows the user to more easily choose the unformatted bibliography file that they created.

WORD COUNT: 1257