Mark Kardash

Professor Carl Eyler

SDEV425 6381: Mitigating Software Vulnerabilities

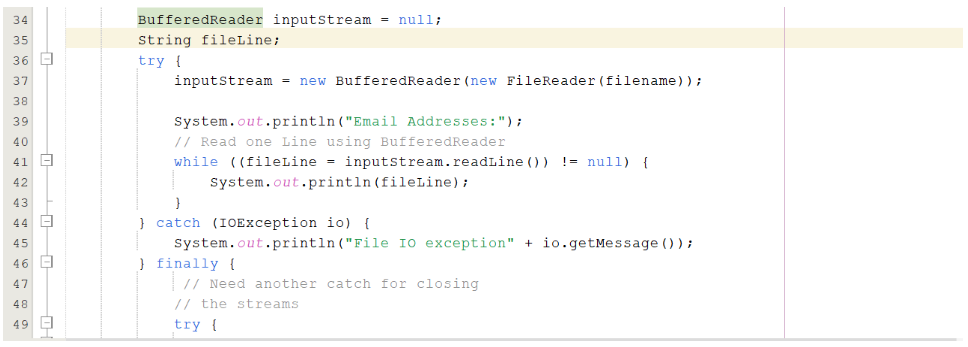
Homework 1

24 January 2024

University of Maryland Global Campus

The first thing I went about doing when correcting the code was to make file names and paths safer and more efficient to use. To find out HOW to do this, I used the guidelines and recommendations found on the Confluence Wiki found in the course materials (IDS50-J. Use conservative file naming conventions - SEI CERT Oracle Coding Standard for Java - Confluence (cmu.edu)). I carefully looked through all the names, and compared them to the rules. Thankfully, the class and file names appeared to comply with all regulations, passing this initial text.

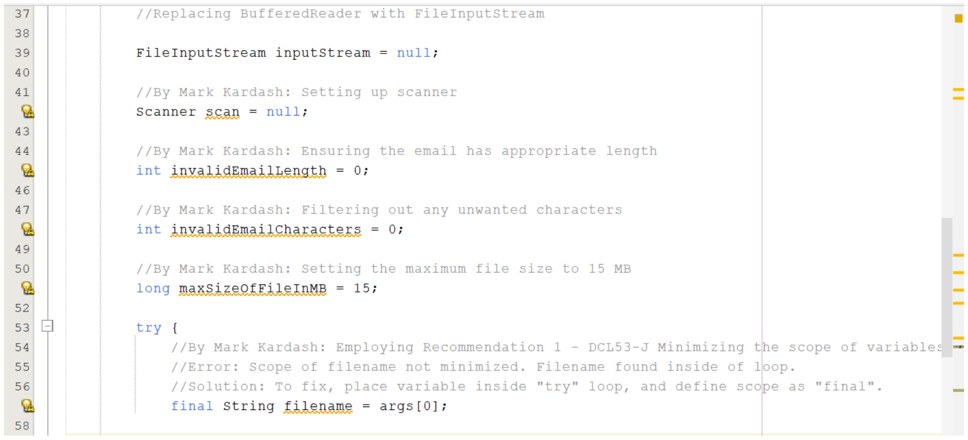
The first real error I found was a violation of Rule 49 MSC05-J, from the Miscellaneous section. The rule is called "Do not exhaust heap space." What it essentially means is that the program may run out of memory and have difficulty reading the entire file. To resolve this problem and comply to the rule, I decided to replace the "BufferedReader" function, which I know very little about, with the "scanner" function, which I am slightly more familiar with. This was done because scanner is not affected by file size as much as the previous function. However, just to be safe, I also specified the maximum file size allowed.



```java
        BufferedReader inputStream = null;
        String fileLine;
        try {
            inputStream = new BufferedReader(new FileReader(filename));

            System.out.println("Email Addresses:");
            // Read one Line using BufferedReader
            while ((fileLine = inputStream.readLine()) != null) {
                System.out.println(fileLine);
            }
        } catch (IOException io) {
            System.out.println("File IO exception" + io.getMessage());
        } finally {
             // Need another catch for closing
            // the streams
            try {
```

Figure 3: Original code with Buffered Reader Function

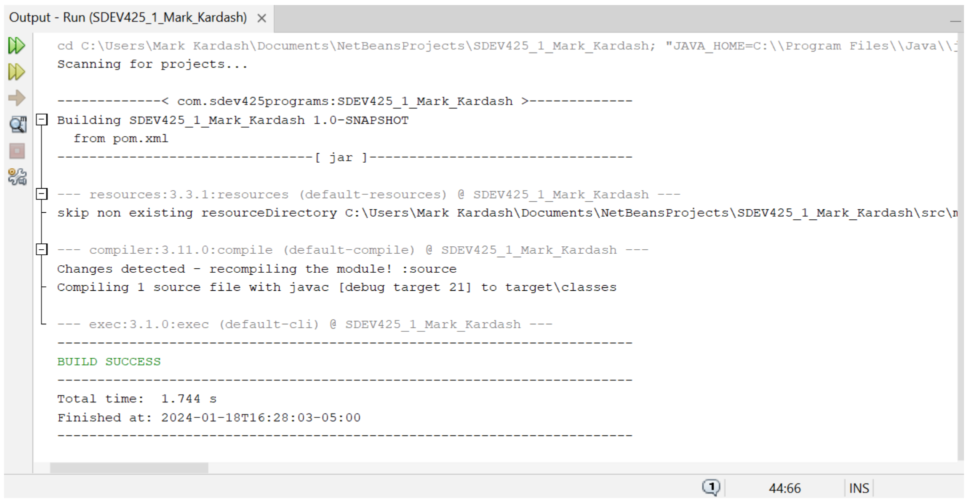
To effectively fix the issue, I also had to employ DCL53-J, from Recommendation 1, which involves minimizing the scope of variables. In simple words, it means defining exactly the nature of a variable, and its location within the code of a program. For easier code readability, it is recommended to place the "filename" variable inside the main loops. I placed mine into the "try" loop, and designated it as "final".



```
        //Replacing BufferedReader with FileInputStream

        FileInputStream inputStream = null;

        //By Mark Kardash: Setting up scanner
        Scanner scan = null;

        //By Mark Kardash: Ensuring the email has appropriate length
        int invalidEmailLength = 0;

        //By Mark Kardash: Filtering out any unwanted characters
        int invalidEmailCharacters = 0;

        //By Mark Kardash: Setting the maximum file size to 15 MB
        long maxSizeOfFileInMB = 15;

        try {
            //By Mark Kardash: Employing Recommendation 1 - DCL53-J Minimizing the scope of variables
            //Error: Scope of filename not minimized. Filename found inside of loop.
            //Solution: To fix, place variable inside "try" loop, and define scope as "final".
            final String filename = args[0];
```

Figure 4: The code after replacing "BufferedReader" with a scanner, and minimizing scope of variables.

As a test, I ran the code to see if it works so far. For some reason, the program misidentified the directory where I had placed the file with the email addresses. To fix this, I decided I needed to specify the directory within the code itself (This will be done a bit later).

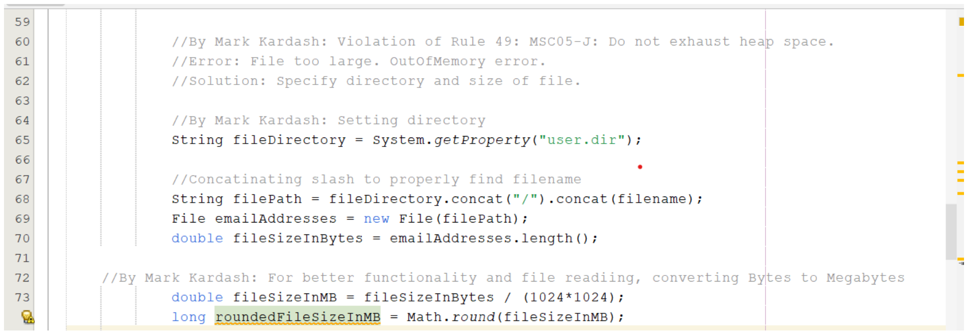


Output - Run (SDEV425_1_Mark_Kardash)

```
cd C:\Users\Mark Kardash\Documents\NetBeansProjects\SDEV425_1_Mark_Kardash; "JAVA_HOME=C:\\Program Files\\Java\\
Scanning for projects...

-------------< com.sdev425programs:SDEV425_1_Mark_Kardash >-------------
Building SDEV425_1_Mark_Kardash 1.0-SNAPSHOT
  from pom.xml
--------------------------------[ jar ]---------------------------------

--- resources:3.3.1:resources (default-resources) @ SDEV425_1_Mark_Kardash ---
skip non existing resourceDirectory C:\Users\Mark Kardash\Documents\NetBeansProjects\SDEV425_1_Mark_Kardash\src\m

--- compiler:3.11.0:compile (default-compile) @ SDEV425_1_Mark_Kardash ---
Changes detected - recompiling the module! :source
Compiling 1 source file with javac [debug target 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ SDEV425_1_Mark_Kardash ---
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  1.744 s
Finished at: 2024-01-18T16:28:03-05:00
------------------------------------------------------------------------
```

Figure 5: Result of first code test run. It lists the Email Addresses file as being inside the “src” (the source code itself), for some reason, when, in reality, the file is simply in the root directory.

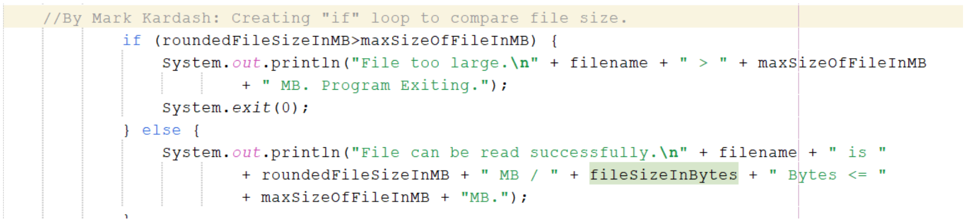
While attempting to fix the directory error, I once again encountered a violation of Rule 49 MSC05-J, “Do not exhaust heap space”. In this case, fixing it would involve specifying the directory and size of the file that the program is supposed to read. To simplify the reading process for the program, I also converted the size from bytes to Megabytes, and rounded them.



```
            //By Mark Kardash: Violation of Rule 49: MSC05-J: Do not exhaust heap space.
            //Error: File too large. OutOfMemory error.
            //Solution: Specify directory and size of file.

            //By Mark Kardash: Setting directory
            String fileDirectory = System.getProperty("user.dir");

            //Concatinating slash to properly find filename
            String filePath = fileDirectory.concat("/").concat(filename);
            File emailAddresses = new File(filePath);
            double fileSizeInBytes = emailAddresses.length();

        //By Mark Kardash: For better functionality and file readiing, converting Bytes to Megabytes
            double fileSizeInMB = fileSizeInBytes / (1024*1024);
            long roundedFileSizeInMB = Math.round(fileSizeInMB);
```

Figure 6: Setting up file directory and calculating file size.

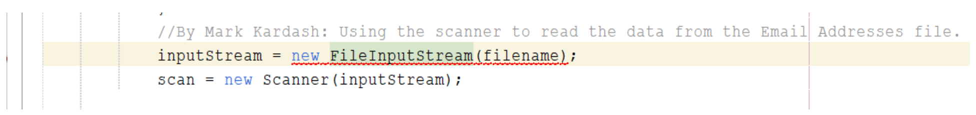
Next, to make sure the file is readable, I had to create an “if” loop, to compare the size of the file with the maximum size I set earlier. The program will read the file if the size is appropriate, or display an error message if it is too large.



```
  //By Mark Kardash: Creating "if" loop to compare file size.
          if (roundedFileSizeInMB>maxSizeOfFileInMB) {
              System.out.println("File too large.\n" + filename + " > " + maxSizeOfFileInMB
                      + " MB. Program Exiting.");
              System.exit(0);
          } else {
              System.out.println("File can be read successfully.\n" + filename + " is "
                      + roundedFileSizeInMB + " MB / " + fileSizeInBytes + " Bytes <= "
                      + maxSizeOfFileInMB + "MB.");
```

Figure 7: Creating an “if/else” loop to control file size and read file.

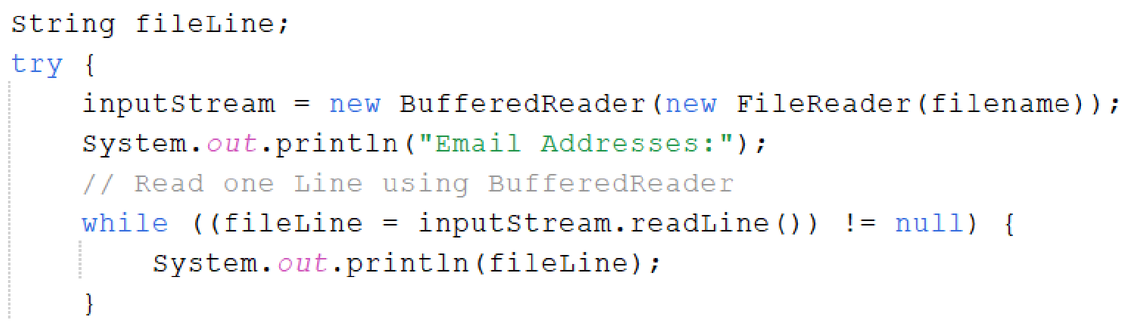
The next step was to actually give the program the ability to read the file, using a scanner and the "InputStream" function (Function marked as error due to program being incomplete. Error will be dealt with later).



```
//By Mark Kardash: Using the scanner to read the data from the Email Addresses file.
inputStream = new FileInputStream(filename);
scan = new Scanner(inputStream);
```

Figure 8: Functions to read the assigned file.

Next, I took advantage of recommendation Rec.00: IDS56-J: Prevent Arbitrary File Upload, to give the program a safer file reading option than it had. The "inputStream.readline()" function of the program may have allowed it to read files with all sorts of extensions, and not just ".txt", including malicious ones.

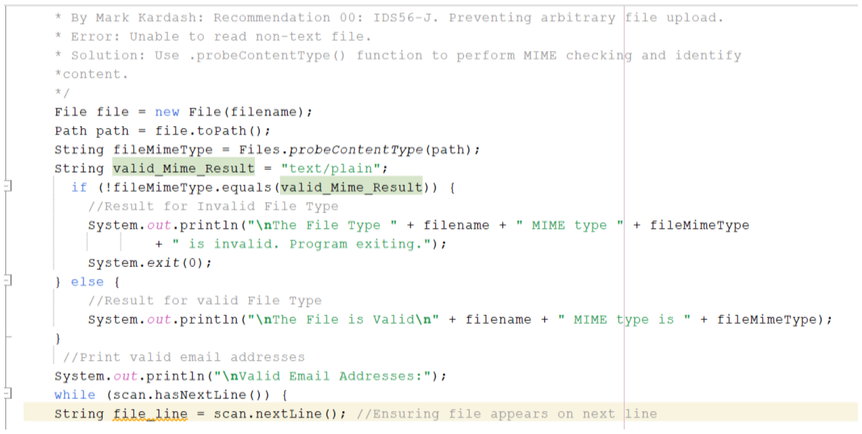


```
String fileLine;
try {
    inputStream = new BufferedReader(new FileReader(filename));
    System.out.println("Email Addresses:");
    // Read one Line using BufferedReader
    while ((fileLine = inputStream.readLine()) != null) {
        System.out.println(fileLine);
    }
```

Figure 9a: Original unsafe file reading system

The function in the above figure does not distinguish between file types, which could allow the upload of anything, including infected files with dangerous extensions. At the same time, the program will throw an error for files it cannot read.

To fix this problem, I performed MIME checking, which is the process of checking a file's contents, and recognizing the file type based on them. I did this via a function called ".probeContentType()". Should a file type be invalid, the program will be exited immediately.

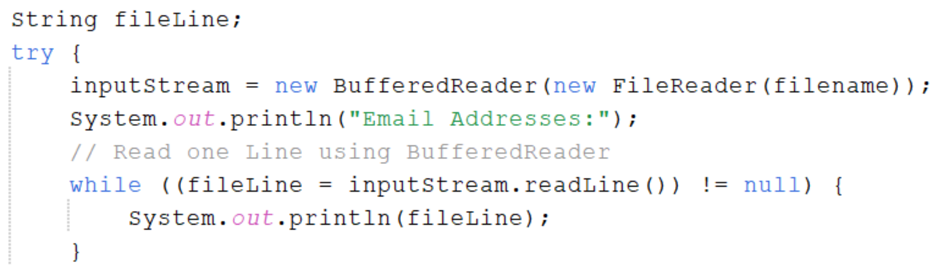


```java
 * By Mark Kardash: Recommendation 00: IDS56-J. Preventing arbitrary file upload.
 * Error: Unable to read non-text file.
 * Solution: Use .probeContentType() function to perform MIME checking and identify
 *content.
 */
File file = new File(filename);
Path path = file.toPath();
String fileMimeType = Files.probeContentType(path);
String valid_Mime_Result = "text/plain";
  if (!fileMimeType.equals(valid_Mime_Result)) {
    //Result for Invalid File Type
    System.out.println("\nThe File Type " + filename + " MIME type " + fileMimeType
            + " is invalid. Program exiting.");
    System.exit(0);
} else {
    //Result for valid File Type
    System.out.println("\nThe File is Valid\n" + filename + " MIME type is " + fileMimeType);
}
 //Print valid email addresses
System.out.println("\nValid Email Addresses:");
while (scan.hasNextLine()) {
String file_line = scan.nextLine(); //Ensuring file appears on next line
```

Figure 9b: Correct code with content-probing function

The figure above shows the new, updated code, which includes the .probeContentType() MIME function (The program is not yet ready for test runs, as it is incomplete due to the changes).

My next move to properly secure the application was to eliminate the possibility invalid inputs that could produce errors, or cause the app to crash. The best option in this scenario would be to follow Rule 00: IDS06-J: Exclude Unsanitized User Input from Format Strings.

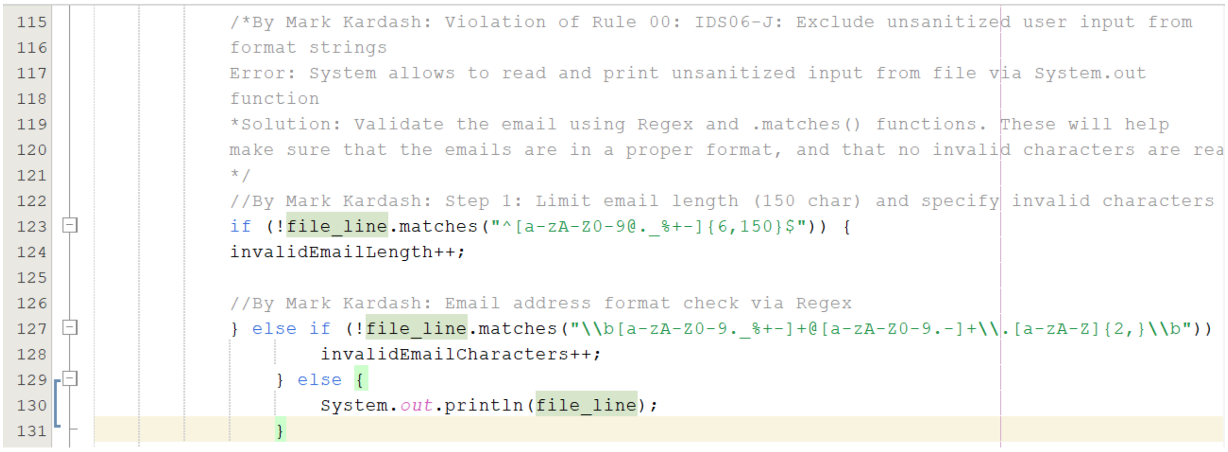


```java
String fileLine;
try {
    inputStream = new BufferedReader(new FileReader(filename));
    System.out.println("Email Addresses:");
    // Read one Line using BufferedReader
    while ((fileLine = inputStream.readLine()) != null) {
        System.out.println(fileLine);
    }
```

Figure 10a: Original unsafe code with System.out function (Screenshot taken from Homework 1 file).

As can be seen above, in practically the same piece of code as the arbitrary file reading error, the system allows to read unsanitized files (aka files containing elements and naming conventions that are not appropriate). They are then printed out using the System.out function.

This problem can be solved by using two functions, Regex and .matches(), to limit the length of an email, and specify the characters that should not be included in it.

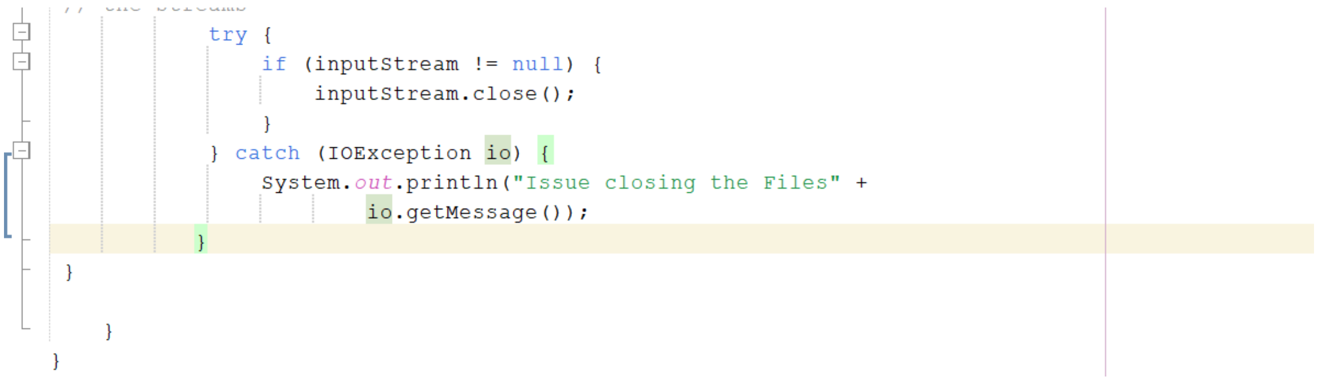


```
/*By Mark Kardash: Violation of Rule 00: IDS06-J: Exclude unsanitized user input from
format strings
Error: System allows to read and print unsanitized input from file via System.out
function
*Solution: Validate the email using Regex and .matches() functions. These will help
make sure that the emails are in a proper format, and that no invalid characters are rea
*/
//By Mark Kardash: Step 1: Limit email length (150 char) and specify invalid characters
if (!file_line.matches("^[a-zA-Z0-9@._%+-]{6,150}$")) {
invalidEmailLength++;

//By Mark Kardash: Email address format check via Regex
} else if (!file_line.matches("\\b[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}\\b"))
        invalidEmailCharacters++;
    } else {
        System.out.println(file_line);
    }
```

Figure 10b: Repaired code with Regex and .matches() functions

In the above image, the possibility of an invalid email being read or invalid characters being admitted is eliminated, as the code is sanitized for any errors or inconsistencies, and a proper email length is set.

The last significant error I found when checking the program was a violation of Rule 13: FI002-J: Detect and handle file-related errors. This is because the file with the Email Addresses was not originally in the arguments of the program, so the program could not properly detect it.

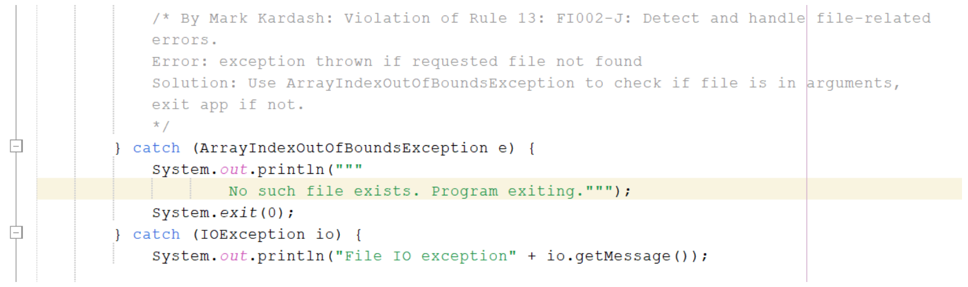


```
            try {
                if (inputStream != null) {
                    inputStream.close();
                }
            } catch (IOException io) {
                System.out.println("Issue closing the Files" +
                        io.getMessage());
            }
    }

        }
}
```

Figure 11a: Original Code (No way to verify file existence)

In the original code above, the program has a very poor catch statement, not having any option for how to behave should it fail to detect the file.

I fixed this issue by making the program catch an ArrayIndexOutOfBoundsException to see if the file exists, and, in the case it doesn't, close itself by exiting.

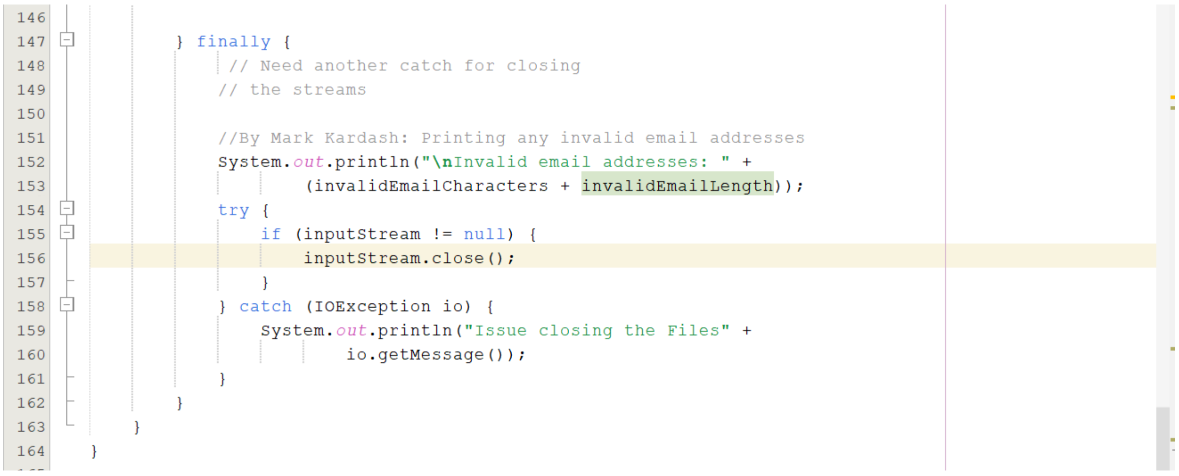


```
        /* By Mark Kardash: Violation of Rule 13: FIO02-J: Detect and handle file-related
        errors.
        Error: exception thrown if requested file not found
        Solution: Use ArrayIndexOutOfBoundsException to check if file is in arguments,
        exit app if not.
        */
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("""
                No such file exists. Program exiting.""");
        System.exit(0);
    } catch (IOException io) {
        System.out.println("File IO exception" + io.getMessage());
```

Figure 11b: Corrected code with catch exception

Now that the code has a "catch" exception, the program can verify that the requested file exists without running into errors, and exiting if it does not.

Before completing the modifications, we must also print any emails that are considered invalid.

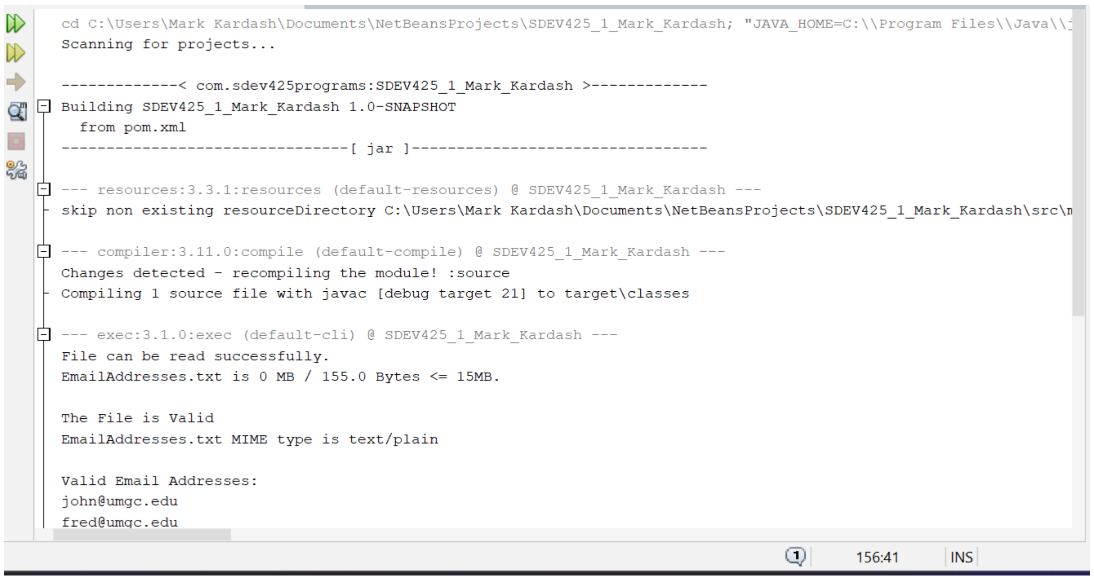


```
        } finally {
            // Need another catch for closing
            // the streams

            //By Mark Kardash: Printing any invalid email addresses
            System.out.println("\nInvalid email addresses: " +
                    (invalidEmailCharacters + invalidEmailLength));
            try {
                if (inputStream != null) {
                    inputStream.close();
                }
            } catch (IOException io) {
                System.out.println("Issue closing the Files" +
                        io.getMessage());
            }
        }
    }
}
```

Figure 12: Printing invalid email characters

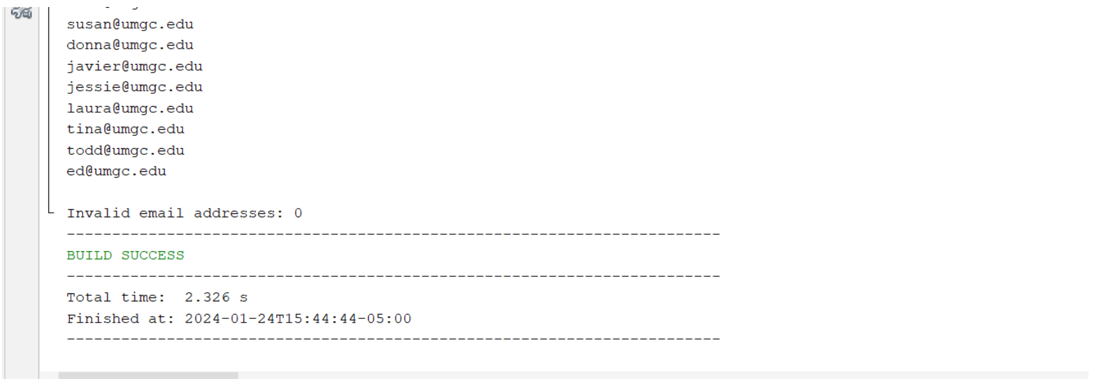
In the figure above, a print statement, including the functions for invalid email characters and invalid email length, was inserted into the "finally" loop of the app to print out any invalid emails.

<u>Test Run:</u>



```
cd C:\Users\Mark Kardash\Documents\NetBeansProjects\SDEV425_1_Mark_Kardash; "JAVA_HOME=C:\\Program Files\\Java\\
Scanning for projects...

-------------< com.sdev425programs:SDEV425_1_Mark_Kardash >-------------
Building SDEV425_1_Mark_Kardash 1.0-SNAPSHOT
  from pom.xml
--------------------------------[ jar ]---------------------------------

--- resources:3.3.1:resources (default-resources) @ SDEV425_1_Mark_Kardash ---
skip non existing resourceDirectory C:\Users\Mark Kardash\Documents\NetBeansProjects\SDEV425_1_Mark_Kardash\src\m

--- compiler:3.11.0:compile (default-compile) @ SDEV425_1_Mark_Kardash ---
Changes detected - recompiling the module! :source
Compiling 1 source file with javac [debug target 21] to target\classes

--- exec:3.1.0:exec (default-cli) @ SDEV425_1_Mark_Kardash ---
File can be read successfully.
EmailAddresses.txt is 0 MB / 155.0 Bytes <= 15MB.

The File is Valid
EmailAddresses.txt MIME type is text/plain

Valid Email Addresses:
john@umgc.edu
fred@umgc.edu
```

Figure 13a: Reading file, confirming file and displaying email addresses.



```
susan@umgc.edu
donna@umgc.edu
javier@umgc.edu
jessie@umgc.edu
laura@umgc.edu
tina@umgc.edu
todd@umgc.edu
ed@umgc.edu

Invalid email addresses: 0
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.326 s
Finished at: 2024-01-24T15:44:44-05:00
------------------------------------------------------------------------
```

Figure 13b: Confirming Build Success, Printing Invalid Emails, Displaying Timestamp

Figures 13a and 13b demonstrate the successful completion of the application's test run, with the file having been confirmed, the email addresses printed, and no invalid emails found.

Summary:

Although the assignment itself was easy to understand, it took me quite a long time to explain and demonstrate everything. I must say that following the Confluence rules and recommendations is indeed very important, for the convenience and safety of both application and customer. The changes I made to the program were indeed effective, and, as seen in the "Test Run" section, it passed the run with flying colors.

References

DCL53-J. Minimize the scope of variables - SEI CERT Oracle coding standard for Java - Confluence. (n.d.). https://wiki.sei.cmu.edu/confluence/display/java/DCL53-J.+Minimize+the+scope+of+variables

FIO02-J. Detect and handle file-related errors - SEI CERT Oracle coding standard for Java - Confluence. (n.d.). https://wiki.sei.cmu.edu/confluence/display/java/FIO02-J.+Detect+and+handle+file-related+errors

IDS06-J. Exclude unsanitized user input from format strings - SEI CERT Oracle coding standard for Java - Confluence. (n.d.). https://wiki.sei.cmu.edu/confluence/display/java/IDS06-J.+Exclude+unsanitized+user+input+from+format+strings

IDS56-J. Prevent arbitrary file upload - SEI CERT Oracle coding standard for Java - Confluence. (n.d.). https://wiki.sei.cmu.edu/confluence/display/java/IDS56-J.+Prevent+arbitrary+file+upload

MSC05-J. Do not exhaust heap space - SEI CERT Oracle coding standard for Java - Confluence. (n.d.). https://wiki.sei.cmu.edu/confluence/display/java/MSC05-J.+Do+not+exhaust+heap+space