Tutorial for file I/O: reading

In this tutorial, you'll practice opening and reading text from a file. You'll create a book reader, which reads a book file and displays it to the user's terminal.

Project Gutenberg is an online library of free eBooks, whose mission is to "to encourage the creation and distribution of eBooks." You can choose from a selection of over 60,000 books to download and read. Your book reader reads a book downloaded from Project Gutenberg, removes unwanted lines, and displays book content to the user.

Every downloaded book has a set of informational lines at the top of the file, describing terms of use, date added to the Gutenberg library, character format, and more. This information isn't useful to the reader, so your program skips these lines. At the end of this section, the file contains a line that looks like:

```
*** START OF THE PROJECT GUTENBERG EBOOK THE STRANGE CASE OF DR. JEKYLL AND MR.
HYDE  ***
```

The lines after that are book content. Likewise, at the end of book content, there is a line that looks like this:

```
*** END OF THE PROJECT GUTENBERG EBOOK THE STRANGE CASE OF DR. JEKYLL AND MR. HYDE
***
```

After that, Gutenberg adds hundreds of lines of text describing Project Gutenberg, which are also not interesting to the book reader. Your program doesn't display these lines to the user.

After your program displays the book, it shows a summary line with the total number of lines of book content displayed.

To get started, import this project into IntelliJ. You'll start your work in the file `BookReader.java`.

Design

Your program must do the following:

1. Prompt the user for a book filename.
2. Open the file for reading.
3. Read each line in the file until it finds a line that begins with the string "*** START OF."
4. Continue reading lines, print each one to the terminal, and count the number of lines.
5. Look for the marker "*** END OF" in a line to indicate the book content is complete.
6. Skip the rest of the file, as there's no need to read the final lines.

Step One: Prompt the user for a filename

In the tutorial starter code, open the file `BookReader.java`. Notice the code under the **Step 1:** comment asks the user for a filename and then waits for the user to type and press **Enter**.

Run this program. You see the prompt in the Run window. If you type something and press **Enter**, the program exits.

Step Two: Open the book file and handle errors

Now add code that attempts to open a file using the file path given by the user. First, create a Java `File` object. Under the **Step 2:** comment, add the following:

```
// Create a File object using the path
File bookFile = new File(filePath);
```

This doesn't create a new file on disk, or open the file on disk. This only creates a Java object to represent a file.

Now add the code to open the file. When you create a new `Scanner`, passing the `File` object you created to its constructor, Java tries to open the file. Since you don't have control over what the user typed, the filename may be incorrect. If that's the case, Java throws a `FileNotFoundException`. Your code must catch that.

Add this code:

```
// Open the file
try (Scanner fileInput = new Scanner(bookFile)) {
    // Loop until the end of file is reached

} catch (FileNotFoundException e) {
    // Could not find the file at the specified path.
    System.out.println("The file was not found: " + bookFile.getAbsolutePath());
}
```

Run this program. At the prompt, type a random word, like "cat." The program informs you that the file wasn't found. Notice that the program tells you the *full path* in which it looks for the file. The `File` method `getAbsolutePath()` gets this information. You can use this method to show the file path that didn't work when the program can't find a file.

> Note: When the program expects a file path, that path may be *absolute* or *relative*. An absolute path usually starts with a drive letter (`C:`) or a root indicator (`/`) depending on what operating system you're running. A relative path starts with a folder name or alias (`.` or `..`). Relative paths use your *current working directory* as the starting point, and then goes to a folder from that point. When you run a Java program from IntelliJ, your *current working directory* is your project folder (the `tutorial` folder in this case).

The `data` folder inside your project contains some book files. Run the program again, and this time when prompted, type the following:

```
Enter path to the book file: data/fairy-tales.txt
```

You won't see an error message this time because your program found a file at the file path given. Java started at your current working directory, then went to the `data` subdirectory, and then found the file `fairy-tales.txt`.

Step Three: Create a read loop and process the lines

You've successfully opened the file. Now add the code to read each line of text, count it, and display it to the user.

Before the `try` block, declare and initialize a variable to hold the count of lines:

```
int lineCount = 0;          // Count of lines between the start and end markers.
```

Inside the `try` block, add this code:

```
/*
Step 3: Create a read loop and process the lines
*/
// Loop until the end of file is reached
while (fileInput.hasNextLine()) {
    // Read the next line into 'lineOfText'
    String lineOfText = fileInput.nextLine();
    // Increment the line count.
    lineCount++;
    // Print the line
    System.out.println(lineCount + ": " + lineOfText);
}
```

`fileInput.hasNextLine()` returns `true` if there are more lines to read from the file. If so, you read the next line using the `nextLine()` method, and place its contents into the string `lineofText`. You increment the counter, and then you print both the line number and the text to the screen.

After the try-catch block, display the total count to the user:

```
// Tell the user how many lines of content were found.
System.out.println("Found " + lineCount + " lines of text in " + filePath);
```

Run the program, and type a path to one of the books at the prompt. You see output that looks something like this:

```
Enter path to the book file: data/fairy-tales.txt
1: Project Gutenberg's Andersen's Fairy Tales, by Hans Christian Andersen
2:
3: This eBook is for the use of anyone anywhere at no cost and with
4: almost no restrictions whatsoever.  You may copy it, give it away or
```

```
5: re-use it under the terms of the Project Gutenberg License included
...
Found 6206 lines of text in data/fairy-tales.txt
```

The output continues until you reach the end of the file, and then it prints the summary line.

Step Four: Skip the header information before book content

You have created a basic book reader. However, notice that there are about 20 lines at the top of the book file, and around 300-400 lines at the bottom, which aren't book content. They're added by the Gutenberg Project, and you don't need to display them to the user.

In the next two steps, you'll ignore the starting lines, then print book content, and then ignore the ending lines.

First, create a variable that tracks whether the line you're currently reading is book content or not. When you begin reading the file, the variable `inBookText` is false, because you're reading header lines.

Before the `try` block, declare and initialize this variable:

```
boolean inBookText = false; // Are you reading between the start and end markers?
```

Now you must look for the "begin marker" that ends the header, and change the variable to `true` because after the header, you're reading book content.

Inside the read loop (the `while` block), after you read the next line (`String lineOfText = fileInput.nextLine();`), add:

```
/*
Step 4: Skip the header information before book content
*/
if (lineOfText.startsWith(BEGIN_MARKER)) {
    inBookText = true;
    continue;  // No need to process this line...go to the next
}
```

Finally, you only want to count and print lines if they're part of the book content. So make those lines conditional by wrapping an `if` statement around them. Replace the existing count and print code with this:

```
if (inBookText) {
    // Increment the line count.
    lineCount++;
    // Print the line
    System.out.println(lineCount + ": " + lineOfText);
}
```

Now when you run the program, the output is slightly different. Approximately the first twenty lines aren't shown, and the total number of lines is smaller:

```
Enter path to the book file: data/fairy-tales.txt
1:
2:
3:
4:
5: Produced by Dianne Bean
6:
7:
8:
9:
10:
11: ANDERSEN'S FAIRY TALES
12:
13: By Hans Christian Andersen
...
Found 6186 lines of text in data/fairy-tales.txt
```

Step Five: Skip the footer information after book content

You've suppressed the header information, but the footer information still shows. Add logic to look for the "end marker", and stop printing after that.

Inside the read loop, after checking for `BEGIN_MARKER`, add code to check for `END_MARKER`:

```
/*
Step 5: Skip the footer information after book content
*/
if (lineOfText.startsWith(END_MARKER)) {
    break;  // Once the program finds the end, break out of the loop.
}
```

Notice the use of the `break` statement. Once you reach the end of book content, there's no need to read lines anymore. So instead of continuing to read and ignoring those lines, you break out of the loop.

When the code exits the loop, it completes the `try-with-resources` block. At that time, Java closes the file.

This time when you run the program, you see no footer information, and the line count is hundreds of lines fewer:

```
Enter path to the book file: data/fairy-tales.txt
1:
2:
3:
4:
5: Produced by Dianne Bean
```

```
6:
...
Found 5827 lines of text in data/fairy-tales.txt
```

Next steps

If you want further practice, think about how you might make the following enhancements to make your program even better:

- In the printout, there are blocks of blank lines that don't look very good on the screen. How can you enhance the program to print the first of a series of blank lines, but if there are any blank lines that follow the first, suppress printing those?
- It might be nice to present a screen of content at a time, and then advance to the next "page" when the user presses **Enter**. How can you enhance to program to do this?
    - Hint: this involves prompting the user inside the read loop.
    - If you do this, you also need a way for the user to quit in the middle of the book. So if the user presses only **Enter**, the book advances, but if the user types "q" and then **Enter**, the program breaks out of the loop and quits.

There are two other Project Gutenberg books in the `data` directory—`jekyll-and-hyde.txt` and `sherlock-holmes.txt`. You can test your program with those books as well.