

Debugging Files

by Sophia



WHAT'S COVERED

In this lesson, you will learn about ways to make working with files more flexible and less susceptible to errors. Specifically, this lesson covers:

Table of Contents

- [1. Working With Files](#)
- [2. File Name and Permissions Errors](#)
- [3. Text Written to File as Block Rather Than as Lines](#)
- [4. StandardOpenOption Modes](#)

1. Working With Files

It is not efficient to edit code every time that you want to process a different file. It would be more usable to ask the user to enter the file name string each time the program runs. This would ensure that they could use the program on different files without changing the Java code.

This is fairly simple to do by reading the file name from the user, using input as follows:

```
import java.io.*;
import java.nio.file.*;
import java.util.List;
import java.util.Scanner;

class EnterFileName {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter existing file name: ");
        String fileName = input.nextLine();
        File inputFile = new File(fileName);
        try {
            List<String> lines = Files.readAllLines(inputFile.toPath());
            System.out.println(inputFile.getName() + " contains " + lines.size() + " lines.");
        }
    }
}
```

```

    }
    catch(IOException ex) {
        System.out.println("File error: " + ex.getMessage());
    }
}
}

```

You would read the file name from the user and place it in a variable named `fileName`. The `Files.readAllLines()` method reads the contents of the file into a `List` named `lines`. The size of the `List` corresponds to the number of lines in the file. Now the program can run repeatedly on different files to count the number of lines in the file.



CONCEPT TO KNOW

Keep in mind that the file names are case sensitive (and spelling counts).

2. File Name and Permissions Errors

What if our user types something that is not a file name? In the following example, the user has typed the name of a file that doesn't exist:

➞ EXAMPLE

```
nonexistentFile.txt
```

This is the expected result:

ConsoleShell

```

➤ java EnterFileName.java
Enter existing file name: nonexistentFile.txt
File error: nonexistentFile.txt
➤ 

```

As you have seen, using the methods provided by the `Files` utility class can result in an `IOException` being thrown, so using `Files.readAllLines()` requires using `try` and `catch` blocks for exception handling. If the file entered by the user does not exist, an `IOException` is thrown and the `System.out.println()` statement in the corresponding `catch` block runs and produces the message:

➞ EXAMPLE

```
File error: nonexistentFile.txt
```



CONCEPT TO KNOW

Note that an `IOException` may also be thrown if the file exists, but the user does not have read permission on the file. If we wanted to check that the file exists before trying to read it, we could use the `Files.exists()` method, which returns a boolean.

To check if the user running the program has read permission on the file, we could use the `Files.isReadable()` method, which also returns a boolean. These methods allow us to produce better error messages and also avoid the overhead cost of throwing an `IOException`.

```
import java.io.*;
import java.nio.file.*;
import java.util.List;
import java.util.Scanner;

class EnterFileName {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter existing file name: ");
        String fileName = input.nextLine();
        File inputFile = new File(fileName);
        // Warn if file doesn't exist
        if(! Files.exists(inputFile.toPath())) {
            System.out.println("The file " + fileName + " does not exist.");
        }
        // Then warn if user doesn't have read permission
        else if(! Files.isReadable(inputFile.toPath())) {
            System.out.println("User doesn't have read permission on " + fileName + ".");
        }
        // If file exists and can be read by user, try to read it.
        // Other I/O errors may be possible, so try/catch needed, but such
        // errors are much less likely.
        else {
            try {
                List<String> lines = Files.readAllLines(inputFile.toPath());
                System.out.println(inputFile.getName() + " contains " + lines.size() + " lines.");
            }
            catch(IOException ex) {
                System.out.println("File error: " + ex.getMessage());
            }
        }
    }
}
```

The result should look like this:

```
Console Shell
> java EnterFileName.java
Enter existing file name: wrong.file.name.txt
The file wrong.file.name.txt does not exist.
> 
```

The specifics of setting file and directory permissions varies by operating system, but Replit's Linux environment uses the `chmod` command to set permissions. If you run the following command on a text file that already exists:

➔ EXAMPLE

```
chmod a-r wrong.permissions.txt
```

Read access is taken away for all users, so running the code above should produce output like this:

```
Console Shell
> chmod a-r wrong.permissions.txt
> java EnterFileName.java
Enter existing file name: wrong.permissions.txt
User doesn't have read permission on wrong.permissions.txt.
> 
```

3. Text Written to File as Block Rather Than as Lines

When writing text to a file, it is important to be mindful of the presence or absence of newline (`\n`) characters in the String values being written to the file. Unlike the `System.out.println()` method for writing text to the terminal, the `Files.write()` and `Files.writeString()` methods do not automatically add a newline character.

The following code writes two statements about flowers to a file called `flowers.txt`:

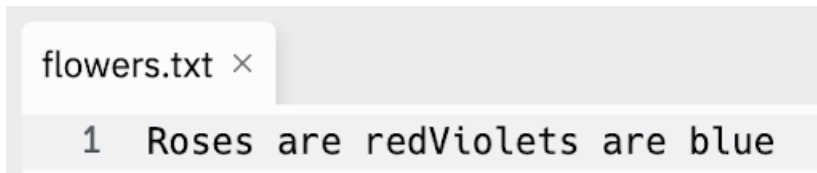
```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.StandardOpenOption;

public class FileNewLine {
    public static void main(String[] args) {
        File flowerFile = new File("flowers.txt");
        try {
            // Write each line without added new line
            Files.writeString(flowerFile.toPath(), "Roses are red", StandardOpenOption.CREATE);
```

```

Files.writeString(flowerFile.toPath(), "Violets are blue", StandardOpenOption.APPEND);
}
catch(IOException ex) {
    System.out.println("Error: " + ex.getMessage());
}
}
}

```



```

flowers.txt x
1 Roses are red
Violets are blue

```

If the two calls to `Files.writeString()` add `"\n"` at the end of each string, the text is written on separate lines. Here are the two modified lines:

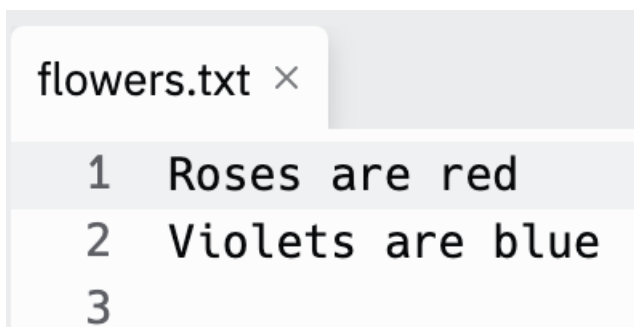
➞ EXAMPLE

```

Files.writeString(flowerFile.toPath(), "Roses are red\n", StandardOpenOption.CREATE);
Files.writeString(flowerFile.toPath(), "Violets are blue\n", StandardOpenOption.APPEND);

```

To see the new results, delete the existing `flowers.txt` file. Since the text includes newline characters, the results will appear like this:



```

flowers.txt x
1 Roses are red
2 Violets are blue
3

```

4. StandardOpenOption Modes

When writing to a file using `Files.write()` and `Files.writeString()`, one of the arguments passed is a value defined in the `StandardOpenOption` enumeration. An enumeration is a set of constant values with names.

We have already seen `StandardOpenOption.CREATE`, which creates a file if it doesn't already exist. If the file already exists, the contents will be overwritten. Note, though, that if the new text is shorter than the previous file contents, any lines not overwritten by new text will be unchanged. This may not be the behavior you would expect.

The argument `StandardOpenOption.APPEND` will cause the text to be written starting at the end of an existing file, but it will fail if the file doesn't already exist. As we saw in the previous lesson, these modes can be combined simply by passing both arguments, as shown in this program:

```

import java.io.*;
import java.nio.*;
import java.nio.file.*;

public class AppendStringToFile {
    public static void main(String[] args) {
        File output = new File("output.append.txt");
        try {
            // Use both StandardOpenOption.CREATE and StandardOpenOption.APPEND so that
            // file is created if it doesn't exist. If the file already exists, text is
            // appended.
            Files.writeString(output.toPath(), "Hello, world\n", StandardOpenOption.CREATE,
                             StandardOpenOption.APPEND);
        }
        catch(IOException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }
}

```

When using `StandardOpenOption.CREATE` to overwrite an existing file, it is commonly paired with the `StandardOpenOption.TRUNCATE_EXISTING` so that the current contents are cleared before the new text is written to the file. This line shows how to pass these two as arguments:

➞ EXAMPLE

```

Files.writeString(flowerFile.toPath(), "Roses are red\n", StandardOpenOption.CREATE,
                  StandardOpenOption.TRUNCATE_EXISTING);

```

It is worth noting that `StandardOpenOption.READ` and `StandardOpenOption.WRITE` are also defined, but the `Files.ReadAllLines()` and `File.Write()` methods that we have used handle opening the file in the correct read or write mode behind the scenes, so the programmer doesn't have to be concerned with them.

There are a few more modes defined in the `StandardOpenOption` enumeration.

`StandardOpenOption.CREATE_NEW` is another one for beginners to be aware of. When using this mode with `Files.write()`, the file is created only if it does not already exist. It throws an `IOException` if the file already exists.



SUMMARY

In this lesson, you learned about handling file-related issues and ways to make **working with files** more flexible. One common issue is dealing with files that don't exist. We can use Java's normal exception handling, but the `Files` class also provides methods that can help us steer clear of common **errors related to file names and permissions**. You learned that when **writing text to a file as block rather than as lines**, it is important to be mindful of the presence or absence of newline (`\n`) characters in the `String` values being written to the file. Lastly, you learned about the various **`StandardOpenOption` enumeration modes** used when writing to a file using `Files.write()` and

`Files.writeString()`.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source py4e.com/html3/