

PLEASE COMPLETE THE SOCRATIVE Pulse Survey

URL: gosocrative.com

ROOM NAME: JAVAGOLD

Please do a 'git pull upstream main' to get today's review materials

Objectives

- **Review**
- File writing

Exceptions Review

- An exception is an error that occurs when the program is running
 - Exceptions can be handled in 2 ways:
 - With a `try-catch-finally`
 - Thrown up the call stack to be handled (with a `try-catch-finally`) by one of the calling methods.
 - Checked exceptions must be caught, Unchecked does not
- Try-catch-finally allows an error to be recovered
 - Create a `try-catch` block around the code that may cause the error
 - Use the `catch()` block to check for a specific exception
 - Recovery routine done within the `catch()` code block
 - Use a `finally` block to close a Scanner file stream if necessary
- Custom exceptions extend from `Exception` (Checked) or `RuntimeException` (Unchecked)
 - `throws` passes the exception up the call stack.
 - `throw` creates a new exception object that must be handled by the calling code.

Checked vs. Unchecked Exceptions

- Checked are compile-time exceptions
 - If code in a method throws a checked exception, method must handle it
 - Handle in method or pass up to parent

File inputFile = getInputFileFromUser();

```
try(Scanner fileScanner = new Scanner(inputFile)) {  
    while(fileScanner.hasNextLine()) {  
        String line = fileScanner.nextLine();  
        String rtn = line.substring(0, 9);  
  
        if(checksumIsValid(rtn) == false) {  
            System.out.println(line);  
        }  
    }  
}
```

Unhandled exception type FileNotFoundException
2 quick fixes available:
1 Add throws declaration
2 Add catch clause to surrounding try
Press F2 for focus

- Unchecked are run-time exceptions

- User or code does something that causes program to stop running

Cincinnati

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3  
at com.techelevator.exceptions.ExceptionsLecture.main(ExceptionsLecture.java:22)
```

File and Scanner: Example

Consider this example:

```
public static void main(String[] args) throws FileNotFoundException {  
  
    File inputFile = new File("resources/testFile.txt");  
  
    if (inputFile.exists()) {  
        System.out.println("found the file");  
    }  
  
    try (Scanner inputScanner = new Scanner(inputFile)) {  
  
        while (inputScanner.hasNextLine()) {  
            String lineInput = inputScanner.nextLine();  
            String [] wordsOnLine = lineInput.split(" ");  
  
            for (String word : wordsOnLine) {  
                System.out.print(word + ">>>");  
            }  
        }  
    }  
}
```

Review Questions

- What is an abstract method?
- What is an exception?
- How would you handle an exception?
- How are a class and an object different?
- What is a method call stack?
- What is the difference between throw and throws keywords?

Review Questions 2

1. What, if anything, is wrong with the code below? How would you fix it?

```
try {  
    int[] intArr = {1, 2, 3, 4};  
    intArr[4] = 42;  
}
```

```
System.out.println("After the try block");
```

Objectives

- Review
- **File writing**

Java Output

Java has the ability to communicate data back to the user. Consider some of these methods:

- Using `System.out.println()` that sends a message to the console.
- Write data to a database (Module 2).
- Transmit data to an API (Module 2).
- Send a HTML view back to the user (Module 3).

For today, we will focus on something simpler, writing data back to a text file.

File class: create a directory.

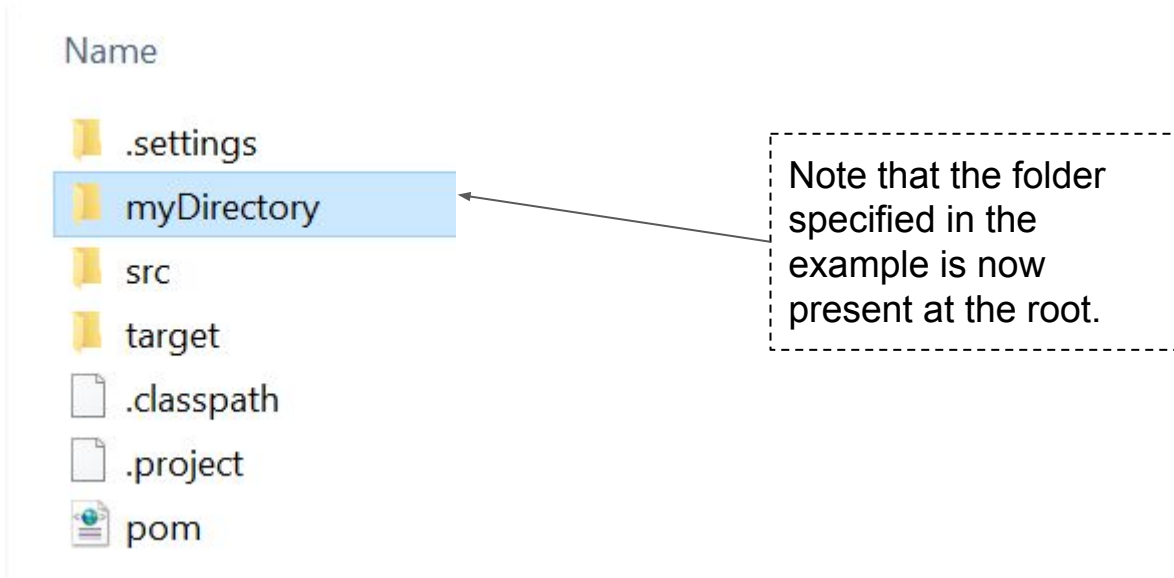
```
public static void main(String[] args) {  
    File newDirectory = new File("myDirectory");  
  
    if (newDirectory.exists()) {  
        System.out.println("Sorry, " + newDirectory.getAbsolutePath() + " already exists.");  
    }  
    else {  
        newDirectory.mkdir();  
    }  
}
```

We won't create a new directory if it exists.

Otherwise, the `.mkdir()` method will create a new directory.

File class: create a directory.

Just like with reading from files, writing is done with respect to the project root.



File class: create a file.

The file constructor can take only a file name, the file will be created at the root of the project.

```
File newFile = new File("myDataFile.txt");  
newFile.createNewFile();
```

The File constructor can also take in two arguments, the directory, and the file name, causing the file to be created at the specified folder.

```
File newFile = new File("myDirectory","myDataFile.txt");  
newFile.createNewFile();
```

Writing to a File

- Just like with reading data from a file, writing to a file involves bringing in an object of another class.
 - In this case, we will need an instance of the `PrintWriter` class.

Writing a File Example

```
public static void main(String[] args) throws IOException {  
    File newFile = new File("myDataFile.txt");  
    String message = "Appreciate\nElevate\nParticipate";  
  
    PrintWriter writer = new PrintWriter(newFile);  
    writer.print(message);  
    writer.flush();  
    writer.close();  
}
```

Create a new file object.

Create a PrintWriter object.

print the message to the buffer.

flush the buffer's content to the file.

The expected result:

- There will be a new text file in the project root.
- The file will be called myDataFile.txt
- The file will contain each of the three words in its own line.

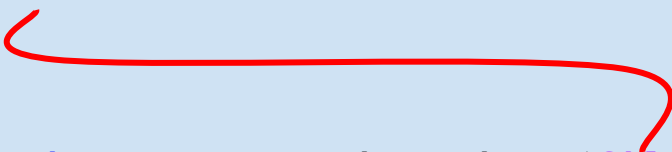
PrintWriter

java.io.PrintWriter

JavaDoc

```
File file = new File(pathToFile);

try(PrintWriter writer = new PrintWriter(file))
{
    writer.println(text);
}
```



The connection to the File must be closed once writing is complete, so it is important to use a try-with-resource, which allows Java to auto-close the file once the PrintWriter is out of scope.

The File is referenced by a File object using the path to the File.

The File object is then passed as an argument to the PrintWriter, which opens it for writing.

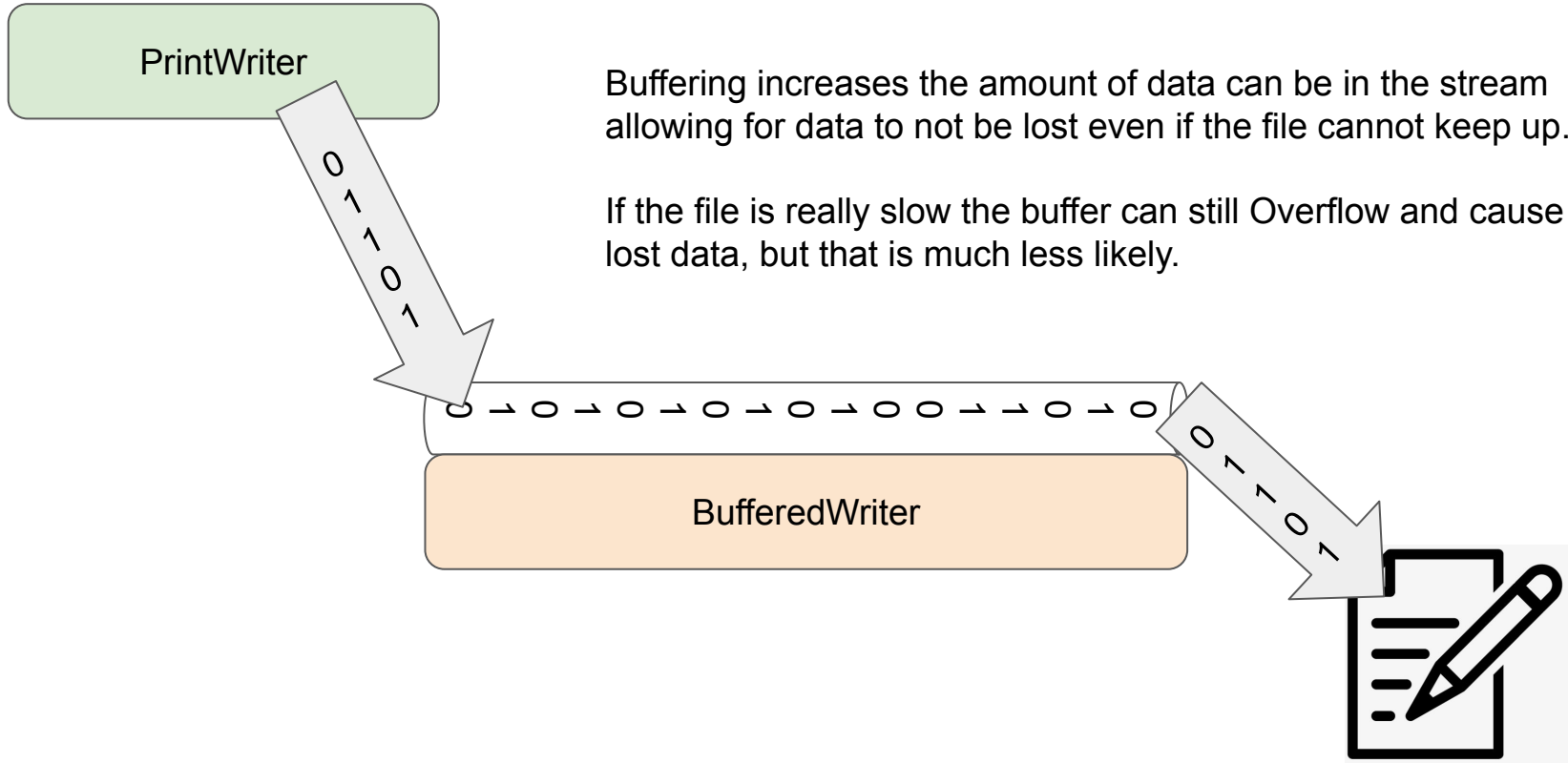
While the File is open the printWriter can then write data to the file by pushing it onto the output stream.

What is a buffer?

A buffer is like a bucket where the text is initially written to. It is only after we invoke the **.flush()** method that the bucket's contents are transferred to the file. **The flush() and the close() can be performed automatically if the “try with resources” is used:**

```
public static void main(String[] args) throws IOException {  
    File newFile = new File("myDataFile.txt");  
    String message = "Appreciate\nElevate\nParticipate";  
  
    try( PrintWriter writer = new PrintWriter( newFile.getAbsolutePath() ) ) {  
        writer.print(message);  
    }  
}
```


Buffering



BufferedWriter

java.io.BufferedWriter

JavaDoc

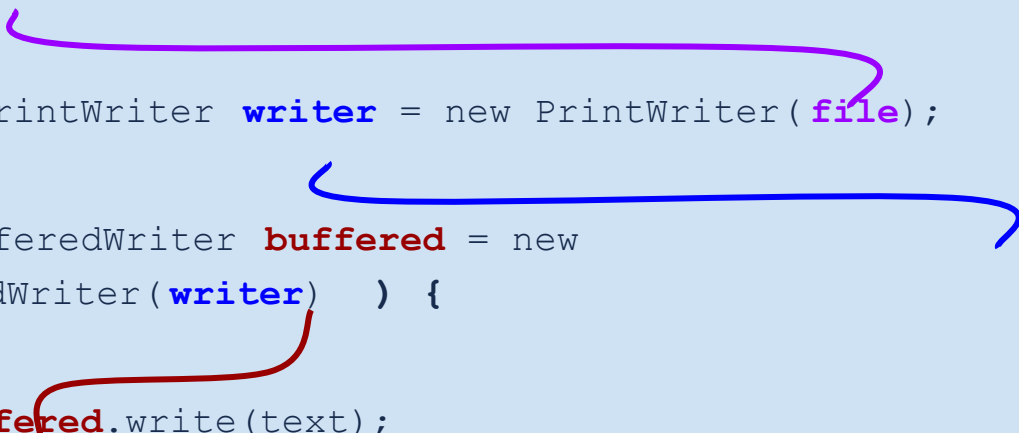
```
File file = new File(pathToFile);

try( PrintWriter writer = new PrintWriter(file);

    BufferedWriter buffered = new
    BufferedWriter(writer) ) {

    buffered.write(text);

}
```

A diagram with three curved arrows illustrating the flow of data. A purple arrow starts from the 'file' variable in the first line and points to the 'file' argument in the 'PrintWriter' constructor. A blue arrow starts from the 'writer' variable in the second line and points to the 'writer' argument in the 'BufferedWriter' constructor. A red arrow starts from the 'buffered' variable in the third line and points to the 'buffered' object in the 'write' method call.

The File is referenced by a File object using the path to the File.

The File object is then passed as an argument to the PrintWriter, which opens it for writing.

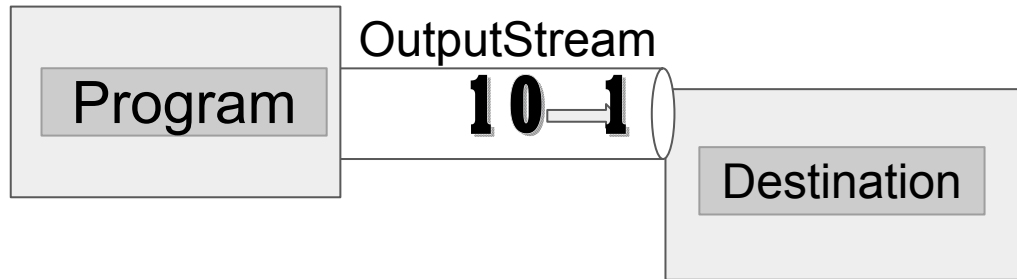
The PrintWriter is then passed as an argument to the BufferedWriter.

The BufferedWriter provides buffering in the case that the file cannot keep up with the speed of the stream.

Flushing a Stream

Sometimes other operations occurring on the system or in the application can cause data to remain in the stream without an opportunity to write it to the destination.

Flushing forces all the data in the stream to be immediately sent from the Stream to the destination.



Streams have a `flush()` method that can be called to immediately flush them.

```
System.out.flush()  
PrintWriter.flush()  
BufferedWriter.flush()
```

A stream should never be closed without first being flushed. If the stream is opened with a try-with-resource, it will automatically flush the stream before closing it.

Appending to a File (Optional Technique)

The previous example regenerates the file's contents from scratch every time it's run. Sometimes, a file might need to be appended to, preserving the existing data content. The `PrintWriter` supports two constructors:

- **`PrintWriter(file)`**, where `file` is a file object.
- **`PrintWriter(new OutputStream(file,true))`**

Appending a File Example (Optional Technique)

```
public static void main(String[] args) throws IOException {  
    File newFile = new File("myDataFile.txt");  
    String message = "Appreciate\nElevate\nParticipate";  
  
    PrintWriter writer = null;  
  
    // Instantiate the writer object with append functionality.  
    if (newFile.exists()) {  
        writer = new PrintWriter(new FileOutputStream(newFile.getAbsolutePath(), true));  
    }  
  
    // Instantiate the writer object without append functionality.  
    else {  
        writer = new PrintWriter(newFile.getAbsolutePath());  
    }  
  
    writer.append(message);  
    writer.flush();  
    writer.close();  
  
}
```

The expected result is that *myDataFile.txt* will be continuously appended with the message text each time it runs.

If the second parameter of the `FileOutputStream` constructor is false, the file will be overwritten