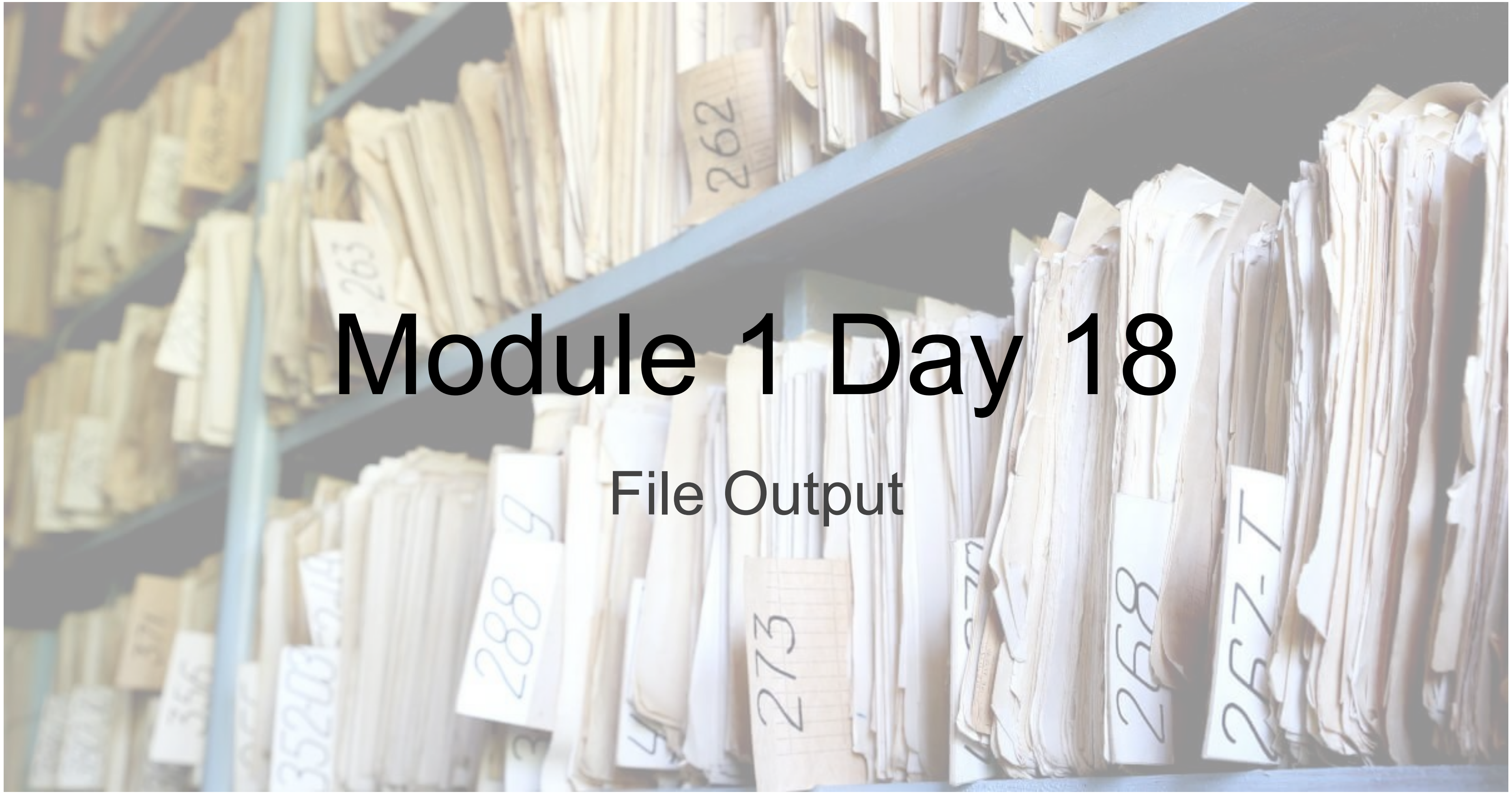


Module 1 Day 18

File Output

Module 1 Unit 18 File Output

Can you … ?

- … describe the concept of exception handling
- … implement a try/catch structure in a program
- … use a try-with-resources block
- ... handle File I/O exceptions and recover from them
- … explain what a character stream is
- … use and discuss the `java.io` package File and Directory classes
- … talk about ways that File I/O might be used on the job

Java Output

Java, like all languages, can communicate data, as output, to the user. This output can occur in various ways:

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

Today, we will focus on writing data back to a text file.

File class: create a directory.

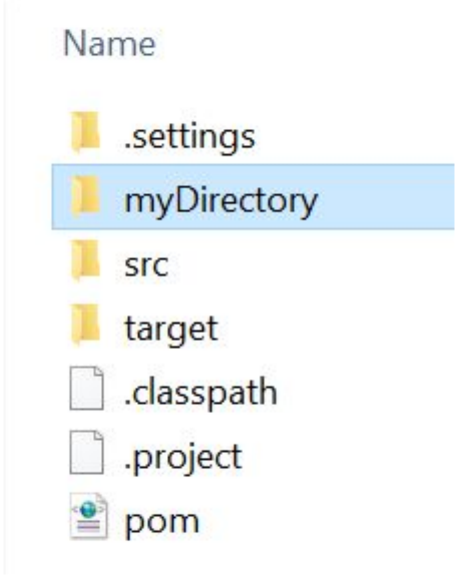
```
public static void main(String[] args) {
        File newDirectory = new File("myDirectory");

        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 reading files, writing is relative to the project root *unless* an absolute path is provided for a directory.


Name

.settings
myDirectory
src
target
.classpath
.project
pom

```
        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();
                }
        }
```

File class: create a file.

```
public static void main(String[] args) throws IOException {
        File newFile = new File("myDataFile.txt");
        newFile.createNewFile();
}
```

File class: create a file within a directory.

```
public static void main(String[] args) throws IOException {
        File newFile = new File("myDirectory","myDataFile.txt");
        newFile.createNewFile();
}
```

Writing to a File

- Writing to a file involves the use of an instance of the PrintWriter class.
- When more than one classes are used to perform a task, those classes are referred to as **collaborators**.
- In this case, the File and Printwriter classes are collaborators.

Buffered Streams

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.getAbsoluteFile());
        writer.print(message);
        writer.flush();
        writer.close();
}
```

Create a new file object.

Create a PrintWriter object.

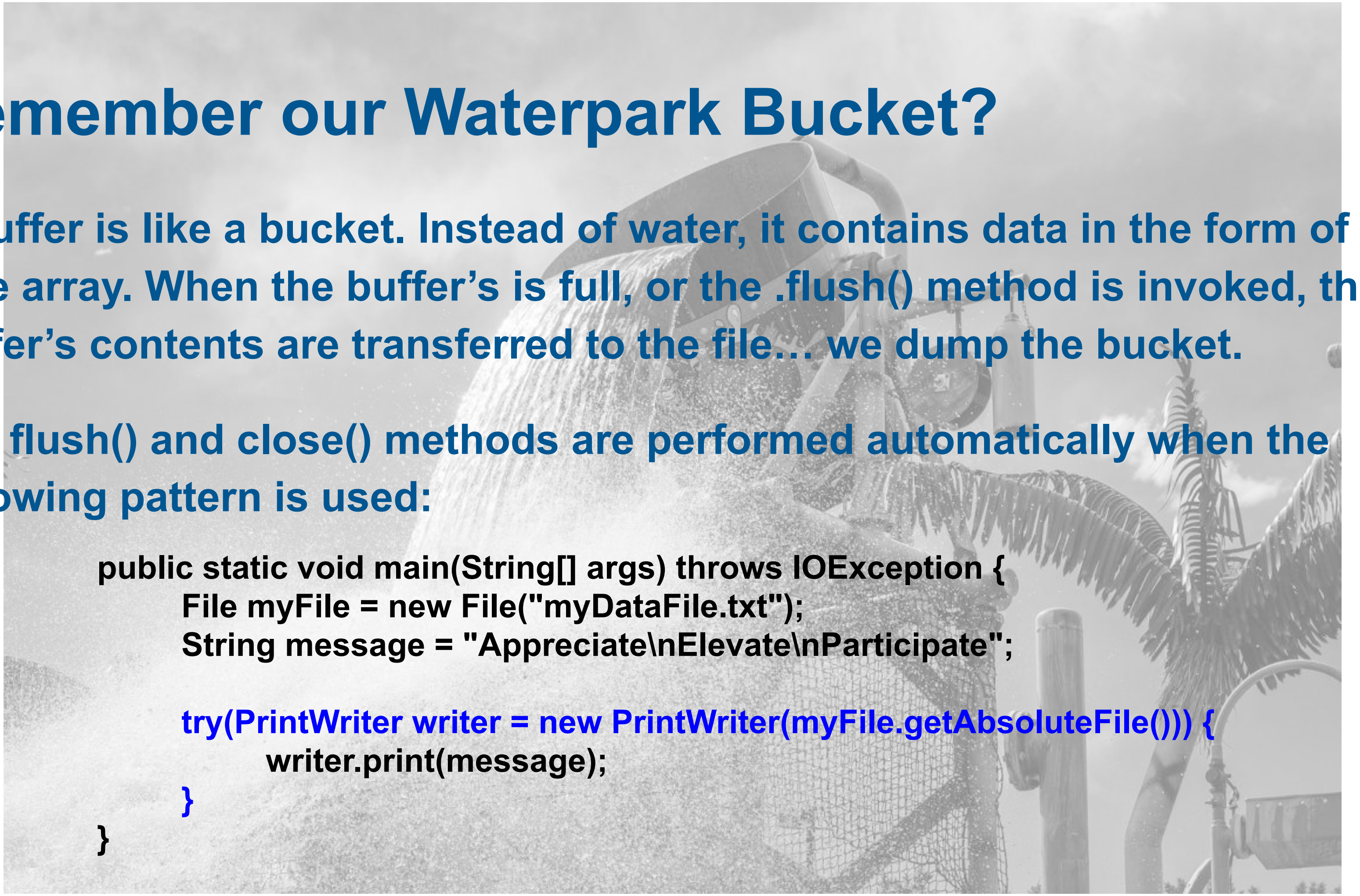
print the message to the buffer.

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 the text of ***message*** each of the three words on its own line due to the ***\n*** newline escape character..

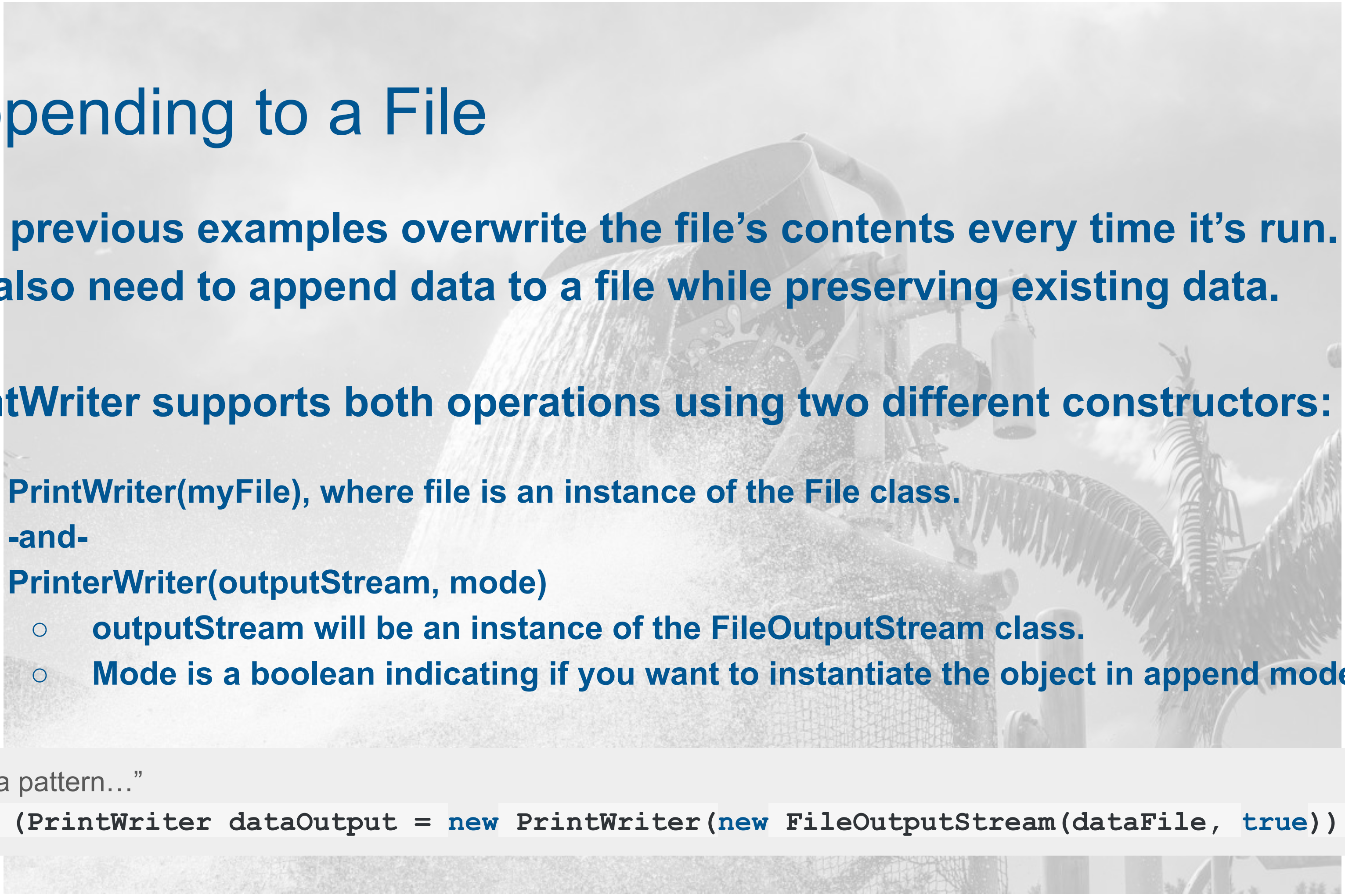
Remember our Waterpark Bucket?



A buffer is like a bucket. Instead of water, it contains data in the form of a byte array. When the buffer's is full, or the .flush() method is invoked, the buffer's contents are transferred to the file… we dump the bucket.

The flush() and close() methods are performed automatically when the following pattern is used:

```
public static void main(String[] args) throws IOException {
        File myFile = new File("myDataFile.txt");
        String message = "Appreciate\nElevate\nParticipate";

        try(PrintWriter writer = new PrintWriter(myFile.getAbsoluteFile())) {
                writer.print(message);
        }
}
```



Appending to a File

The previous examples overwrite the file's contents every time it's run. But we also need to append data to a file while preserving existing data.

PrintWriter supports both operations using two different constructors:

- **PrintWriter(myFile), where file is an instance of the File class.**
  -and-
- **PrinterWriter(outputStream, mode)**
  - **outputStream will be an instance of the FileOutputStream class.**
  - **Mode is a boolean indicating if you want to instantiate the object in append mode**

"It's a pattern…"

```
try (PrintWriter dataOutput = new PrintWriter(new FileOutputStream(dataFile, true)))
```

Appending text to a File: Example … what's missing?

```
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.getAbsoluteFile(), true));
		}
		// Instantiate the writer object without append functionality.
		else {
			writer = new PrintWriter(newFile.getAbsoluteFile());
		}
		writer.append(message);
		writer.flush();
		writer.close();
	}
}
```

The expected result is that myDataFile.txt will be continuously appended to with ***message*** *each time this code runs.*