Reading from a File in Java

Reading from a file is a fundamental operation in Java, where you retrieve data from an existing file and process it. The classes typically used for reading a file in Java include:

1. **FileReader**: A class used for reading character files.
2. **BufferedReader**: A wrapper around `FileReader` that improves efficiency by buffering the input and providing methods like `readLine()`.

Let's break down the process step by step.

Example Code

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class ReadFileExample {
    public static void main(String[] args) {
        try {
            // Step 1: Create a FileReader object
            FileReader fileReader = new FileReader("example.t
xt");

            // Step 2: Wrap the FileReader in a BufferedReade
r for efficient reading
            BufferedReader bufferedReader = new BufferedReade
r(fileReader);

            // Step 3: Read each line of the file until the e
nd
            String line;
            while ((line = bufferedReader.readLine()) != nul
l) {
```

```
                System.out.println(line); // Print each line
to the console
            }

            // Step 4: Close the BufferedReader
            bufferedReader.close();
        } catch (IOException e) {
            System.out.println("An error occurred while readi
ng the file.");
            e.printStackTrace();
        }
    }
}
```

Step-by-Step Breakdown

1. **Creating a `FileReader` Object:**

   ```
   FileReader fileReader = new FileReader("example.txt");
   ```

   - The `FileReader` class is designed to read character data from a file.
   - It takes the file name (or path) as a parameter. In this example, we're reading from a file called `"example.txt"`.
   - If the file does not exist, a `FileNotFoundException` (a subclass of `IOException`) will be thrown.

2. **Using `BufferedReader` for Efficient Reading:**

   ```
   BufferedReader bufferedReader = new BufferedReader(fileRea
   der);
   ```

   - The `BufferedReader` class wraps around `FileReader` to improve performance by reducing the number of I/O operations.
   - It reads larger chunks of data at once and stores them in a buffer, allowing for faster access.

Why Use `BufferedReader`?

- Without buffering, each `read()` operation in `FileReader` directly interacts with the file system, which is slow.
- `BufferedReader` allows us to use the `readLine()` method, which reads one line at a time, making it easier to work with text files.

3. **Reading Each Line of the File:**

```
String line;
while ((line = bufferedReader.readLine()) != null) {
    System.out.println(line);
}
```

- The `readLine()` method reads a line of text and returns it as a `String`. It returns `null` when the end of the file is reached.
- The `while` loop continues reading lines until it hits the end of the file.

4. **Closing the `BufferedReader`:**

```
bufferedReader.close();
```

- It is important to close the `BufferedReader` after reading the file to release resources and avoid potential memory leaks.
- The `close()` method also closes the underlying `FileReader`.

Handling Exceptions

- The `IOException` is a common exception when dealing with file I/O operations. In our code:

```
catch (IOException e) {
    System.out.println("An error occurred while reading th
e file.");
    e.printStackTrace();
}
```

- If any issue occurs (e.g., file not found, I/O error), the exception is caught, and a relevant message is displayed.

Example Output

Suppose the `example.txt` file contains the following content:

```
Hello, World!
This is a sample file.
Java File Handling is important.
```

When you run the code, the output will be:

```
Hello, World!
This is a sample file.
Java File Handling is important.
```

Important Notes

- The `BufferedReader` provides better performance when reading large files or multiple lines of text.
- You can also use `Scanner` to read files, but `BufferedReader` is more efficient for reading line-by-line content.
- Always remember to close the reader in a `finally` block or use a try-with-resources statement to ensure the reader is closed even if an exception is thrown.

This detailed explanation should help clarify the steps involved in reading from a file in Java and how each part of the code works together.