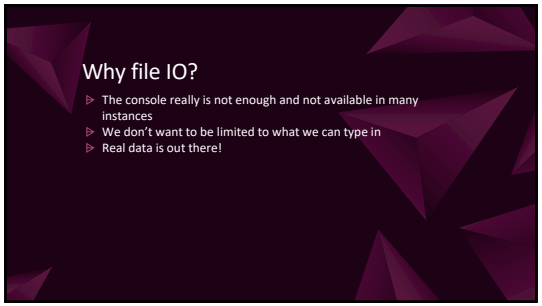
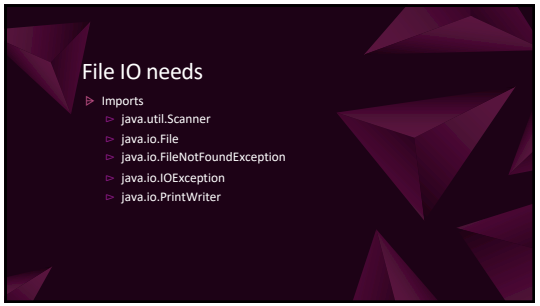

Java File IO
CS 1400 / CS 1410 2024
Cody Henrichsen

1


Why file IO?
The console really is not enough and not available in many instances
We don't want to be limited to what we can type in
Real data is out there!

2


File IO needs
Imports
java.util.Scanner
java.io.File
java.io.FileNotFoundException
java.io.IOException
java.io.PrintWriter

3

Try/Catch

- File access is NOT guaranteed. As such we need to write code that can deal with errors that could happen when we attempt to access items not immediately available.
- The try with resources block provides a space for the potentially error causing code to run
- Using the try with resources block also supports good practice so that all associated data sources are properly closed
  - Safety
  - Security
- The associated catch block allows our program to do something other than crash when an exception occurs

4

Static methods

- We want to minimize the code exposure, so we will be using static methods for our IO operations
- Our Controller class is passed in as a parameter so that error handling which is not the responsibility of the IOController can be passed along to the handleError method.
- No internal state is needed for this type of utility method so making it explicitly static makes sense.

5

Reading from a file

- Our read method has a parameters for a reference to the Controller, a String for the path of the file we want to read, and returns a String
- Since we are returning the text from a file the return variable is initialized to an empty String.
- Inside the resource for the try, create a Scanner instance from the File specified by the path.
- Use a while loop in the block to read each line from the file and append a new line symbol to maintain consistency
- Catch the associated exceptions
- Return the text

6

Reading example

```
public static String readTextFromFile(Controller app, String path)
{
    String text = "";

    try (Scanner fileScanner = new Scanner(new File(path)))
    {
        while (fileScanner.hasNextLine())
        {
            text += fileScanner.nextLine() + "\n";
        }
    }
    catch (NullPointerException | FileNotFoundException error)
    {
        app.handleError(error);
    }

    return text;
}
```

7

Writing to a file

- Writing to a file is almost a reverse of reading
- This is a void method with a Controller reference, String for path, and a String for the text to save
- Create a PrintWriter from the path parameter and a Scanner instance from the text parameter in the resources for the try
- While the scanner has text use the println method to write each line to the file
- Catch a FileNotFound exception if needed

8

Writing example

```
public static void writeTextToFile(Controller app, String text, String path)
{
    try (PrintWriter writer = new PrintWriter(new File(path));
         Scanner stringScanner = new Scanner(text))
    {
        while (stringScanner.hasNextLine())
        {
            writer.println(stringScanner.nextLine());
        }
    }
    catch (FileNotFoundException error)
    {
        app.handleError(error);
    }
}
```

9

Using the IO Methods

- The `Controller` class has instance methods that call the associated `IOController` methods
- The `load` method accepts the path and calls `IOController.readTextFromFile` and returns the results
- The `save` method accepts the text and path and calls the `IOController.writeTextToFile` method
- The `handleError` method displays an error if called

10

Using example

```
public void handleError(Exception error)
{
    JOptionPane.showMessageDialog(window, error.getMessage(), "Ooops", JOptionPane.ERROR_MESSAGE);
}

public String load(String path)
{
    String results = IOController.readTextFromFile(this, path);
    return results;
}

public void save(String text, String path)
{
    IOController.writeTextToFile(this, text, path);
}
```

11