Text File IO in Java

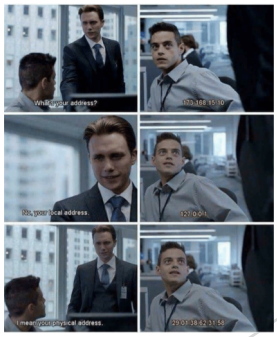
When you want to save and load to a "local" directory

Cody Henrichsen

IO Operations

- Require a try/catch block since IO access is NOT guaranteed
  - All the code in a try is valid, the compiler just doesn't know if the executing environment has the associated access/files/permissions
  - Exceptions should be processed from specific to general
- The biggest difference between file input and output is the order/direction of steps. Most of the steps are the same
  - String: The text to save/load
  - String: file path
  - While loop: Repeat the process until finished (Simon says)
  - Scanner: Process the File or String contents by line

Paths



Why Java Static Methods

- No state
  - Can be `void` or return type
  - No use of `this`.
  - **No data members**
    - All code needs to be included in the method or via parameter
- Simple execution of non-OOP code
  - It is not a thing that does stuff, rather just an action that can be called
  - Always `public`!
- Great for utility actions like IO operations or calculations
  - See also `Math.random()`, `Math.abs(value)`, etc. `ClassName.methodName(paramsAsNeeded);`

Input Process - Scanner

- Get the path to the file
- Check that the file exists
- Use a `Scanner` to read each line from the file in a loop
  - Append a new line marker since `.nextLine()` consumes them
- Add that line to a `String` variable
  - `+=` is easiest
  - `.concat` requires more work
    - `contents = contents.concat(fileScanner.nextLine() + "\n");`
- Close the `Scanner`
- Return the `String` variable

```
public static String loadFile(ChatController app, String path)
{
    String contents = "";

    try
    {
        File saveFile = new File(path);
        Scanner fileScanner;
        if(saveFile.exists())
        {
            fileScanner = new Scanner(saveFile);
            while (fileScanner.hasNext())
            {
                contents += fileScanner.nextLine() + "\n";
            }
            fileScanner.close();
        }
    }
    catch(IOException error)
    {
        app.handleErrors(error);
    }
    catch(Exception genericError)
    {
        app.handleErrors(genericError);
    }
    return contents;
}
```

Output Process - Scanner

- Create a file from the path parameter and time info
- Create the resources needed in the try block
- Parse the `String` parameter with a `Scanner`
- Use the `.hasNext()` method to loop over the text from the parameter
- Write one line at a time with a `PrintWriter`
- Close the `PrintWriter` and `Scanner` automagically

```
public static void saveTextWithResources(DataController app, String path, String textToSave)
{
    String filename = path;
    LocalDateTime date = LocalDateTime.now();
    filename += "_" + date.getMonth() + "_" + date.getDayOfMonth();
    filename += "_" + date.getHour() + "-" + date.getMinute();

    filename += " data structures save.txt";
    File saveFile = new File(filename);

    try(Scanner textScanner = new Scanner(textToSave);  PrintWriter saveText = new PrintWriter(saveFile))
    {
        while(textScanner.hasNext())
        {
            String currentLine = textScanner.nextLine();
            saveText.println(currentLine);
        }
        //Note the lack of a .close() call for the Scanner and PrintWriter objects
        //The close methods are handled as part of the AutoCloseable interface the
        //Try with Resources approach uses.
    }
    catch(IOException error)
    {
        app.handleErrors(error);
    }
    catch(Exception genericError)
    {
        app.handleErrors(genericError);
    }
}
```

Input Process - Bytes

- Create a `String` variable
- Get the path for the file from the parameter
- Check that the file exists
- Create a `FileInputStream` from the file
- Loop over each byte in the file with the `.read()` method until it returns -1
- Cast the resulting byte as a `char`
- Append the `char` to the `String`
- Close the `FileInputStream`
- Return the `String`

Output Process - Bytes

- Build a file path based on date and supplied directory
- Initialize stream and scanner as resources
- While scanner has text retrieve each line, and append a new line then convert to a byte array
- Write the bytes to the file
- Since the stream is appendable add the time at the end
- Catch exceptions as needed