file handling

# design

There are two parts to developing this section: constructing the system for organising and storing files, and the set of methods that will go alongside that.

## file management

There are going to be 3 main classes in the new file system:

FileStorage 🡺 Stores all of the files inside it, with methods concerning them all  
FileItem 🡺 Equivalent to a single file, with contents and a name and methods for reading and writing  
FileHandler 🡺 Used for opening files, and interacting with them

Overall, the system will use a lot of abstraction. There will be no actual usage of files when the program is running only the FileItem instances, which will contain a name and contents to abstractively behave as a file. There will be ways to import and export files to allow for proper interaction, however this will be implemented alongside the user interface. Overall, the whole idea behind this system is abstraction.

### file storage

This will behave very much like the SymbolTable class, except it will work with file names instead of Identifier tokens. There will only ever be a single instance of it, and the plan is that it will not reset each time a script is executed, however for now that is not possible as the code is currently completely restarted each time.   
The class will contain the following properties and methods:

🡺 files: an array of FileItems which belong to the storage  
🡺 get(): will accept the file name, and will return the corresponding file name  
🡺 addNew(): will accept a filename, and will create a new file with the corresponding name

Overall, this is very simple, however not that if the file passed to addNew() already exists, then it will reset the contents to be empty instead of creating a second one, to ensure no duplicates occur, as well as this simulating how file management works in actual languages.

### file item

As explained, this is the template for a file, and will have the following properties and methods:

🡺 name: a string which represents the file name  
🡺 contents: a string representing the contents, containing \ns instead of being split  
🡺 reset(): will set the contents to “”, being called by addNew if it already exists on creation  
🡺 write(): accepts a single argument string, and will write the argument to the contents.

Note that these methods will be very simple and have little to no error handling, because they are going to be called by the corresponding custom properties and functions which will do the type checking.

### file handler

The purpose of the file handler may seem obsolete, but the same file can be opened in multiple times, and when using readLine() then we need a way to keep track of the multiple different positions for the different locations, and this is where the handler comes in place.

When running some tests in python, it allowed for these multiple file readers to operate this, so this is what I will base the system off, as it is a close relative to the ERL. Therefore, to simulate this, the FileHandler class will have the following properties:

🡺 file: will store the corresponding FileItem to the handler, given in the constructor.  
🡺 contents: the file’s contents, split on \n to be separated into lines  
🡺 position represents the current reading position, starting at -1  
🡺 closed: starts as false, and represents if the handler has been closed by .close()

Then, the methods that go alongside this are:

🡺 readLine(): increments the position and returns the corresponding string or null  
🡺 endOfFile(): returns true if the entire contents has been iterated through  
🡺 write(): will accept an argument of what to write, updating its own contents and calling the file’s write()

Again, very minimal validation will occur, as this will be dealt with in the properties, which I can now explain. This should be the overall structure of the File system complete, and it is mainly comprised of lots of very simple methods which should be easy to implement.

## the methods and functions

Now, the corresponding methods and functions need to be implemented. This will build off the previous framework, so all the parsing will remain the same, with only some additions to build\_global() and make\_property(). They will all have a call() method because they are all methods or functions.

### open

The Open class will extend Subroutine.

It will have a call() method, and will accept one argument, ensuring that it is a StringType. It will use the find() method on the global FileStorage to get the corresponding FileItem. If it receives null, it means the file does not exist so an error will be returned, so Open will return a new Evaluation error.

If a FileItem is returned, then a new FileHandler will be created, and will be passed the FileItem to use, and then this will be returned so that it can be used by the user.

This FileHandler is what all the properties will act on: so, their callees must always be a FileHandler, so this is why open() will be stored in a variable, so that the FileHandler that is returned can be used later.

### FURTHER SUBHEADING

Close will extend Property.

Therefore, it must be added to to make\_property() with the “close” name.

This property will change the FileHandler closed property to True. Now, every single other property that acts on a method, including close(), will check to see if the FileHandler is closed before performing a check. If it is, then it will return an Error. This will therefore create the illusion that the file has been unloaded from the program, even though only a property has changed.

I am not concerned with the performance issues for not properly closing a file, because the files used with the interpreter will be so small that it shouldn’t make a huge difference to the speed or memory usage, because the files are going to need to be stored in memory anyway for the file storage system to function correctly, and there is no way to easily use secondary storage on a web-page.

### read line

The ReadLine class will extend Property, being created by “readLine” in make\_property()

After ensuring that its callee is an opened FileHandler, it will call the FileHandler’s readLine() function to get the next string. If the result is null, then it will return an error saying the end of the file has been reached. Otherwise, it will create a new StringType, and set the value to the result and will return that.

The FileHandler code manages the position, so not a lot of work must be done here.

### write line

WriteLine will also extend Property and will be made by “writeLine”.

The call will accept a single argument this time, being the string that needs to be written. So, after ensuring that the argument is a string, and that the callee is a FileHandler, it will call write() on the FileHandler, passing the value but not the entire StringType as an argument. Write() will handle the implementation, changing the FileHandler’s contents and the FileItem’s contents, and then WriteLine will return the null return value.

### end of file

EndOfFile will be created by “endOfFile”.

Again, after checking that the callee is an open FileHandler, it will call the FileHandler’s endOfFile() method, which will return a boolean based on the FileHandler’s position and the file length.

EndOfFile will then create a new BooleanType, setting its value to the result of endOfFile(), and will then return it, which will then allow the method to function.

### new file

NewFile will extend Subroutine.

It will also be added to make\_global(), with the Identifier name being “newFile”.

Whenever it is called, it will ensure that the argument is a StringType, and I could also implement some additional validation into this method: to ensure that it has a valid file format and that it ends with a specific file extension, however I will probably add this later in development.

Then, it will call addNew() on the global symbol table, passing the value of the StringType as the argument. This will then handle adding the new file: either creating the file if it does not exist, or resetting the contents if it does, as explained earlier. New file will then return the null value because the file then needs to be opened afterwards to be used, as explicitly stated.

This concludes the plan for the file system, overall being generally simple in concept, but with a lot of different Classes to add to complete the functionality.

# development

## opening and creating files

A screen shot of a computer program

Description automatically generatedA screen shot of a computer code

Description automatically generatedA computer screen shot of a program code

Description automatically generatedFirstly, I added the basic storage system, as described in the plan. It is very similar to a symbol table but does not need identifiers to function, and instead uses the file names as strings.

The basics of the file handler was also added, so it can be created for open().

A screen shot of a computer code

Description automatically generated  
The open method follows the same format as a lot of the other native functions.

After ensuring it is passed 1 string, it will search the file storage for the file name and will either return an error or return a new file handler.  
I have saved the global file storage as a static property in Evaluator for consistency in the program.

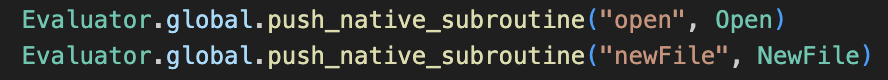
### creation

A screen shot of a computer code

Description automatically generatedThe new file function is also very simple, and just calls addNew on the global file storage, which then handles the creation.

### testing

A black background with colorful text

Description automatically generatedFirstly, I had to add both new native subroutines into make\_global()

A screen shot of a computer

Description automatically generatedInitially the only errors were in Open, where I had not used the let keyword, and was using self instead of the “this” keyword.

A screen shot of a computer

Description automatically generatedThe test case shows that files can now be created, and open() will return an error if they do not exist, as shown the output of the entire file storage added at the end of the output.

However, this is a very poor test case, and I really need to add some more functionality to ensure that the whole system will work.

## reading files

I am manually adding this test file below to help with this section:

A screen shot of a computer program

Description automatically generatedA computer screen shot of a code

Description automatically generatedThen the FileHandler is given a readLine() property, which will increment its position and return the corresponding line or null if it is beyond the end of the file.

A black background with blue and green text

Description automatically generatedThe new property will ensure it has no arguments and will call readLine() on the file handler. If the result is null, it will return an error, and otherwise will return a string with the contents.

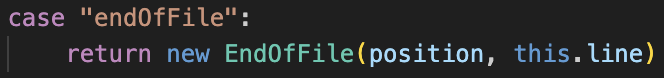
A black screen with white text

Description automatically generatedA screen shot of a computer program

Description automatically generatedAfter adding the new property to make\_property(), it works as anticipated, as shown in the result below.

### A computer screen shot of text Description automatically generatedend of file

A black background with blue text

Description automatically generatedThis property is very similar to the previous, but instead calls the file’s endOfFile() method, and returns the result in a boolean.

A screen shot of a computer program

Description automatically generatedAfter adding it as a property and setting endOfFile() to return based on the position being less than the length, I ran a test.

A black screen with white text

Description automatically generatedA black background with blue text

Description automatically generatedInitially, this test case failed because the next line was still read, however after realising it was because readLine() incremented the position when it was run, the endOfFile is technically met at one before the length, so this needed a small changed to the comparison.

Now, all the different lines are correctly displayed and it stops at the end of the file without an error.

## writing

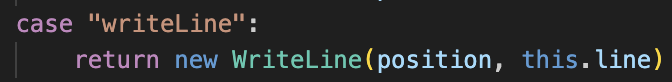
A black background with blue text

Description automatically generatedA black screen with white text

Description automatically generatedFirstly, FileItem is given a very simple method that will add the argument string to its contents. This is not completely needed currently; however, I will add some implementation where it may be helpful later.

A screen shot of a computer program

Description automatically generatedFileHandler’s write method will call write() on its file, passing its argument. It also then updates its contents to ensure that it stays up to date with its current contents.

The new property follows a very similar format: ensuring that it belongs to a FileHandler, and that is has a single argument that is a string. It will then call write() on its FileHandler, which will then update the file. It is also added to the Lexer as shown:

Now we should be able to write to files.

### test cases

A screen shot of a computer program

Description automatically generatedA black background with white text

Description automatically generatedA screen shot of a computer program

Description automatically generatedA black background with white text

Description automatically generatedAs shown, the file type can now be written to and read from again, and the values will be stored in the file correctly.

A black background with colorful text

Description automatically generatedHowever, when trying to use \n to write multiple lines, the program would not recognise this, and would write it as plaintext instead of adding newlines.

A black background with white text

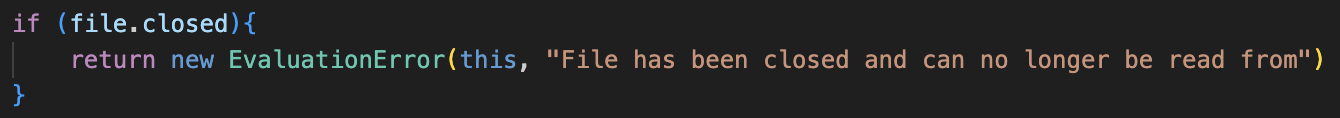
Description automatically generatedThe solution was using the replace() method inside FileHandler’s write() method.

This would replace any plaintext \ns, with real ones, which would therefore actually separate them, therefore the two separate readLines() now both yield a separate line as expected.

## final features

### closing

First, the new close property is added to FileHandler, as planned.

From here, every single property is given a check to see if its FileHandler is closed. If it is, then it will return an error message and not allow the method to execute.

A screen shot of a computer program

Description automatically generated  
Now, the Close property itself is made being very simple: ensuring that it belongs to a FileHandler and has no arguments, and it then sets the closed property to true.

A screen shot of a computer program

Description automatically generatedTherefore, this will stop the other properties from functioning: therefore, closing the file.

A black sign with white text

Description automatically generatedThe test case for this works as expected, and the program will no longer allow these files to be operate on.

A black background with white text

Description automatically generatedOf course, this does not really change anything: the Interpreter is only intended for small, GCSE questions, so needing to clear system resources is not really needed because no insanely huge files are going to be used.

However, I can make a slight addition in Close to clear the contents of the FileHandler, which should increase performance slightly.

### A computer screen shot of text Description automatically generatedimport and export

As more of a temporary measure, I have added two methods to FileStorage which allow for files to be imported and exported from the local storage, so that they can be used for the program.

This is very temporary, as when the proper interface is implemented there will need to be a different system for uploading and managing files. However, for now, this is a good solution. The methods are not intended to be referenced by any code, and only by the developer – hence why they have no error handling.

A screenshot of a computer

Description automatically generatedA computer code on a black background

Description automatically generatedIn this example, at the bottom of the sourcecode, I add the line to export the test.txt file after running.

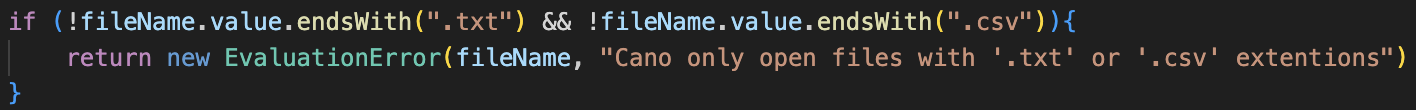
It correctly created the file with the same contents as what was written in the ERL.

### name checking

A computer screen shot of numbers

Description automatically generatedCurrently, using import(), it is possible to open different types of files: such as this png on the left. This creates a completely broken output, as it is only intended for text files.

Therefore, I might limit the interpreter so that it can only open certain types of files. When discussing this with Tanish, he agreed as the example on the left is extremely confusing and should be avoided.

The implementation for this is simple, by checking the extension that the filename ends with.

I added this check to both Open and NewFile and limited the two extensions to “.txt” and “.csv” files, as they are both plaintext-based and are the only types I’ve ever used in gcse questions.

However, this could easily be expanded in the future to include other file types, if I decided that others should be allowed based on feedback.