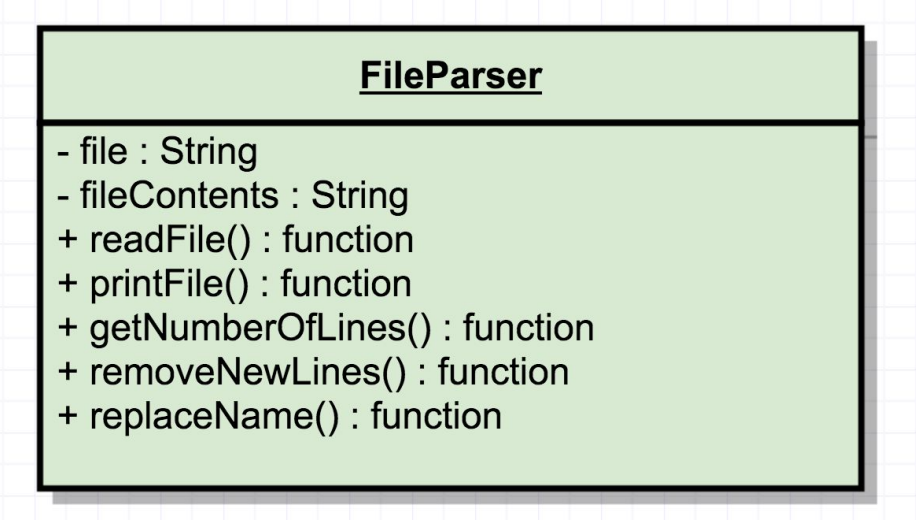
Mini Lab 1

The purpose of this mini lab is to practice encapsulation in JavaScript by using the concept of *Closures*. Encapsulation is essentially the selective hiding of properties and methods in an object to prevent access and/or data corruption. A basic form of encapsulation in a traditional programming language is to use Private or Protected members inside the class. In JavaScript, all members are public and accessible to anyone. However, private members can be achieved through *closures*, which is when a function will have access to its parent scope even after the parent function has finished execution and been garbage collected.

In this class you will implement the following class diagram in JavaScript. The private members in the diagram will have to be implemented using “Closures”. The FileParser object will read in a filename that is passed in by the user and store the contents in a private variable. A number of public utility methods are available to either retrieve data about the file or manipulate the file.

Use the the Node module FileStream to read in the file. FS is an I/O wrapper around POSIX functions. To use FileStream, simply call *require(‘fs’)* in your project. All the FileStream methods have an asynchronous and synchronous form.



Step 1: Download the Mini Lab 1 skeleton project from Github.

Step 2: Inside the FileParser function object, create two private variables, 1) **file** - to store your filename, and 2) **fileContents** - to store the contents of the file. The

FileParser return object will contain 5 methods that will be publicly available anytime you instantiate a new instance.

Step 3: You will need to implement each of the public methods of the FileParser:

1) *readFile()* - To avoid having to use asynchronous callbacks you can use the synchronous form, **fs.readFileSync(filename, 'utf8')**. Pass in **File** variable that you created beforehand. This will return a buffer which you will need to call **toString()** on and store in your **fileContents** private variable.
2) *printFile()* - simply log the **fileContents** to the console.
3) *getNumberOfLines()* - Use a for loop and loop through each character of the content data and check for a new line ('\n'). Keep track of this and return the number of newlines in the file.
4) *removeNewLines()* - Use the *String.replace()* method and a regex **(/\r?\n|\r/g)** to replace all the instances of new lines with a single whitespace. *String.replace(regex | subStr, subStr)* accepts two arguments, the substring or regex to look for, and the new subStr you want to replace it with. Use the above regex as the first argument, and just a single whitespace character (' ') as the second argument.
5) *replaceName(name)* - This method should replace all the instances of 'Sam' in the file contents with the name parameter passed in. This can be done using a regular expression, but please try it manually by splitting the words into an array and using a for loop and doing a comparison on each word manually. Instead of doing a direct comparison use the *string.indexOf(subStr)* method to check for 'Sam'. For example, string.indexOf('Sam') will return -1 if the letters "Sam" **do not** appear in sequence in the string, or it will return an integer of 0 or greater if they do. If you encounter an instance of the word 'Sam', replace it with the name variable passed in using the following *string.replace('Sam', name)*.

Step 4: Create a new instance of FileParser. Read in the file and print out the number of lines to the console. Next, remove all newlines from the file contents, and replace the name "Sam" with your name. Then print the contents of the file out to the console.