## Block Class

-The Block class is used to simplify the use of 128 byte segments.

When a block is needed simply construct a block with the data requested.

then use getBlock to get the byte array from the block. setBlock is made private as

to avoid unregistered changes to the Block.

### Methods

Block(byte[] NewContents) - constructor for the Block class simply calls setBlock.

getBlock() - returns the Block as an array of 128 bytes.

setBlock(byte[] New Contents) - confirms that the array of bytes is less than or equal to the 128 bytes. If it is too high the block will not be made and will return a -1. If it is too small it will write the bytes to an block of size 128 bytes and return a 1. If it is the perfect size then the block will be created and return a 0.

## PartitionControlBlock Class

The PartianControlBlock class is used to reprecent the PCB. The PCB

holds:

- a pointer to the Root Directory (represented by an int that represents the block it is held at)

- the size of the FAT

- the total number of data blocks

- a pointer to the first free block in the FAT(the list of free blocks is its own class)

- a linked list representation of the free space blocks

### Methods

PartianControlBlock(int FirstBlock, int TotalSize) - This is the constructor for the PCB. First I remove the first to blocks the first for the reserved space, and the second for the PCB. the total number of blocks is used to determine the number of blocks in the FAT. this is then used to determine the number of blocks taken up by the FAT which are then also removed from the free blocks. another free block is removed and acts as the pointer to the root directory. Finally the remaining free block us used as the start of the free space list

getRootDir()- returns a pointer to the root directory(as an index in the FAT)

getFirstFreeBlock() - returns the firstFreeBlock and replaces it with the next free block

UpdateFirstFreeBlock(freespaceList FreeSpace) - sets the first free block to the next block in the list

addFreeBlock(int IndexOfBlock) - adds a block with the specified index

getTotalDataBlock() - returns the number of total data blocks.

getSizeOfFAT() - returns the size of the fat.

### Internal Class

freespaceList - this represents all of the free blocks as a linked list of ints representing the index of the next free block

#### Methods

freespaceList(int FirstFreeBlock, int lastBlock) - this is the constructor for the list. When called it

initializes a linked list of blocks between the first and last blocks.

addBlock(int FreeBlockLocation) - adds a block to the free space list. The block is located at the specified location.

getFreeBlock() - returns the next free block.

## FileAllocationTable Class

The FileAllocationTable class is used to represent the FAT. The FAT holds:

- an entire table consisting of the location on the next block in this file

- a Block table consisting of the block associated with each location on the EntrieTable

- total number of entries in the table

### Methods

FileAllocationTable(int TableSize)- this is the FAT constructor when called it initializes both the block and entry array based on the specified size.

changeBlockAndEntrie(int Index, int Entrie, Block NewBlock) - simultaneously changes the block and entry at the specified index.

setBlockTotable(int Index, Block NewBlock) - block at the specified index.

getBlockFromTable(int Index) - returns the block at the specified index

ChangeEntrie(int Index, int Entrie) - Changes the Entry at the specified index.

getEntrieFromTable(int Index) returns the entry at the specified index.

getEntryTable() - returns the full entry table(represented as an array of ints)

## Directory Class

the Directory holds the following:

- a FileAttributes which describes the directory

- a linked list of FileAttributess that each represent a file or directory these attributes can then be used to find the appropriate directory using the FAT(this feature is not yet fully implemented)

### Methods

Directory(String FileName, int startBlock, int fileSize) - This is the constructor for the directory when called it creates a directory with the specified attributes.

addFile() - when called it adds a file to this directories linked list

addDirectory() - when called it adds a new directory to this directories linked list. This is done by making a addition to the linked list and specifying it as a directory in its file attributes.

getFile(String fileName) - when called it looks for an entry in this directory with the specified name. When found the file attributes are returned and can be used to construct the file using the FAT. If this file is a directory(specified in the file attributes) the constructed file will be able to act as a directory.

findIndex(String fileName) - when called it looks for an entry in this directory with the specified name. When found it returns the index in the linked list that it is located at.

getDirectoryInfo() - returns the file attributes of the this directory;

## FileAttributes Class

the FileAttributes class holds the following information(based on the Project requirements):

- if it is a file or directory

- the name of the file/directory as an array of 15 bytes

- the first block in the FAT for this file(if its a file)

- the file name as a string

- the length of the name

- the size of the file/directory

### Methods

FileAttributes(boolean isfile, String FileName, int startBlock, int fileSize) - when called it constructs the class with the specified information. If the name exceeds the specified 15 bytes in size it will be denied.

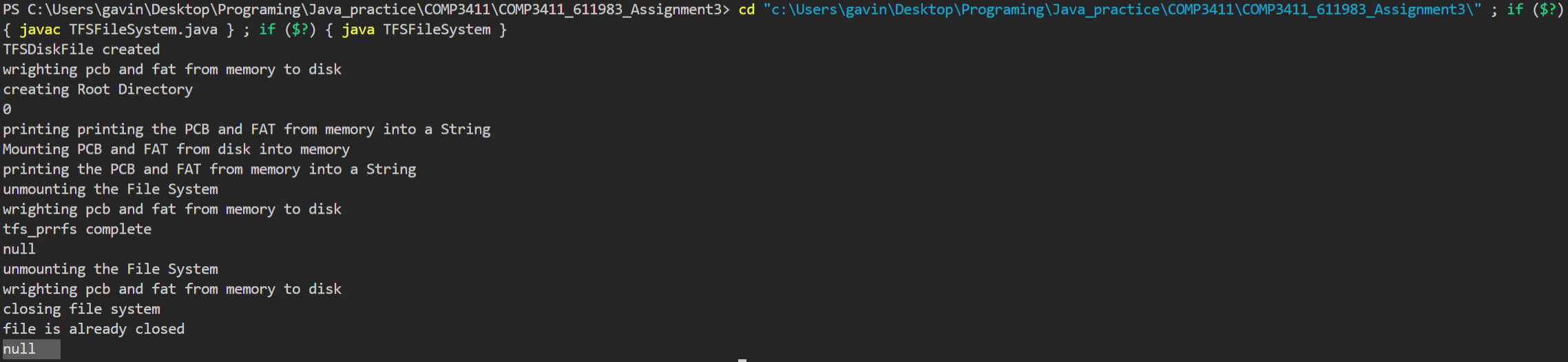
getName() - returns the file name as a string.

# Testing

Here are each of the tests demonstrating the functionality of the program.

## Test #1

To better demonstrate the process each action prints what it is doing during the process.



When running the main in TFSFileSystem.java it executes the following commands

TFSFileSystem fs = new TFSFileSystem();

System.out.println(fs.tfs\_mkfs());// if a 0 is printed no errors occur if its 1 then the filesystem

//already existed

System.out.println(fs.tfs\_prrfs());// if a null is printed an errors occur since

System.out.println(fs.tfs\_exit());// the file should already be closed since it was not left open

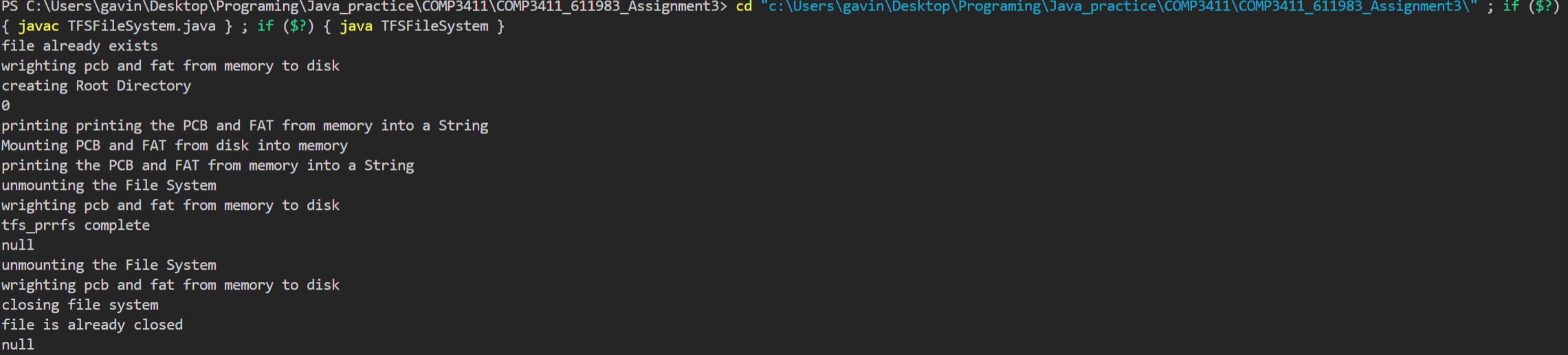
Basically it

- makes the file system

- prints the FAT and PCB

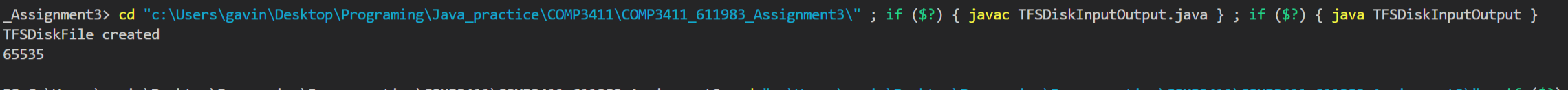
- exits the filesystem

## Test #2



When the same main is executed the program changes since the FIle System already exists.

## Test #3



When running the main in the TFSDiskInputOutput.java the following commands are executed:

TFSDiskInputOutput fileIO = new TFSDiskInputOutput();

String name = "hello.txt";//since the disk file would not be readable i am using a txt file

fileIO.tfs\_dio\_create(name.getBytes(), name.length(), 65535);//create the file (name is hello.txt, the total size is only 12 bytes)

fileIO.tfs\_dio\_open(name.getBytes(), name.length());//open the disk for i/o

fileIO.tfs\_dio\_get\_size();//check the size of the diskfile

System.out.println(fileIO.CurrentFileSize);

//try to read from the diskfile

byte[] bufferBlock = new byte[BlockSixeLimit];

fileIO.tfs\_dio\_read\_block(0, bufferBlock);

String BlockText = new String(bufferBlock);

System.out.print(BlockText);

//try to write to file

String data = "inside of file";

bufferBlock = data.getBytes();

fileIO.tfs\_dio\_write\_block(0, bufferBlock);

//close disk file

fileIO.tfs\_dio\_close();

Basicly

-it creates a file on your computer called hello.txt

-Opens it

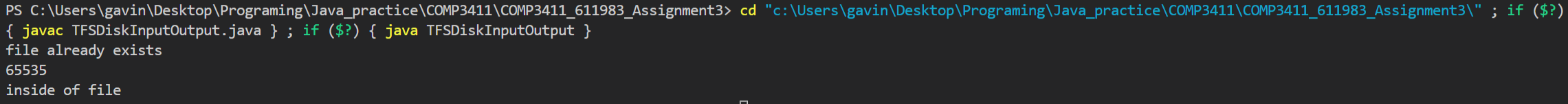
-Checks the size of the Disk file

-reads the first block

-writes “inside of file” in the first block

-closes the Disk File

## Test #4



To prove that the File was actually created and changed the program was ran again with the same main.