Cultima is a role playing game that needs completion: read <CultimaManuals.pdf> to learn about what it is and how to play it.

The ability to save one's progress in the game is critical for developing a character and playing beyond a single sitting. As the program currently sits, the mechanic for reading the player's saved state always returns ***null*** as a flag to let the program know that something has gone wrong: the player file has not been read in correctly. Until the code is fixed, a new randomly assigned player character will be generated every time the game is run. When done correctly, the game will say ***“Welcome back”*** when the player file is read in completely.

Your assignment is to complete the method that will correctly read in the player's saved progress from the text file so that they can continue their adventure with the same character. Here are the details:

Standard classes needed: (you may not have learned all of these, so look them up in the Java API)

Scanner methods: hasNextLine(), nextLine()

String methods: trim(), substring(start), substring(start, end),

equals(other), split(delimeter), indexOf(part)

Wrapper-class: Integer.parseInt(word), Double.parseDouble(word)

Boolean.parseBoolean(word), Byte.parseByte(word)

List methods: add(object)

java.awt.Point: constructor Point(x, y)

***CultimaDriver.java*** contains the main function that will run the game and ***FileManager.java*** contains the method that must be completed for the assignment:

public static Player readPlayerFromFile(String fileName)

This method is expecting the player's information to be in a file called ***<data\player.txt>***

Some values read in as a String from the file need to be stored in a primitive numeric type, like an int, double, or byte (a very small, memory light int). To verify that a given string could be represented as a numeric type (comprised of characters that only contain digits, possibly with a negative in the front and possibly with a decimal point), you may invoke the following helper methods in the ***FileManager*** class:

//post:returns true if word could be parsed into a whole number, positive or negative

// i.e., the word is comprised of all digits, perhaps with a “-“ as the 1st character

public static boolean wordIsInt(String word)

//post:returns true if word could be parsed into a double, positive or negative

public static boolean wordIsDouble(String word)

If at any time the data from the file is determined to be corrupted, the method should return a ***null*** value to let ***CultimaPanel*** know to create a new, random character. As you complete the method, one field at a time, you can step-trace the process to see if it is doing things correctly. The character’s progress in game will not be able to be saved until the entire method is completed correctly.

Here is what needs to be completed in the ***readPlayerFromFile(String filename)*** method from the player data file, in order. Look for the comment string “TO DO” in FileManager.java to find where to complete the code. Line numbers represent the lines from data/player.txt.

Line 3: Three integers that mark the location that the player logs on their map for Teleportation spells. They are to be read into an array of 3 integers. Line 3 might look like this: -1 -1 -1 \*\*mapMark

The numbers from the string “-1 -1 -1” need to be extracted out and stored in an array of 3 ints.

int [] mapMark = new int[3]; //this will store the int value from the 3 elements from line 3 separated by spaces

Note: The numbers from the string read in can be any three numbers. If the string contains “0 34 173”, then the resulting int array called mapMark should contain the int values 0, 34, and 173.

Line 4: a collection of < Potion.NUM\_POTIONS > ints separated by a space that represent the number of potions that the player has among the different types of potions. These will be placed into an array:

int [] potions = new int[Potion.NUM\_POTIONS];

This is similar code to what you did for mapMark.

Line 5: a collection of < Player.NUM\_FLOWERS > ints separated by a space that represent the number of flowers that the player has among the four types of florets. These will be placed into an array of ints:

int [] flowerBoxCount = new int[Player.NUM\_FLOWERS];

Line 6: a series of 4 ints separated by a space that represent the number of experience points that the player has accumulated in specific skills, used to give an appropriate skill increase when the player levels up. The 4 values need to be read in to the int array:

int [] specificExperience = new int[4];

Line 7: a series of <Player.NUM\_STATS> ints separated by a space used to keep track of various player statistics. They need to be read into the int array:

int [] stats = new int[Player.NUM\_STATS];

Line 8: a series of <Player.NUM\_INFO\_INDEXES> ints separated by a space used to keep track of the information the player has learned about different topics, used to populate the player’s journal. They need to be read into the int array:

int [] infoIndexes = new int[Player.NUM\_INFO\_INDEXES];

Line 9: a series of four ints separated by a space that represent the spell hot-keys that the user assigns to quickly cast a spell. They are to be read into an array of four ints:

int []spellHotKeys = new int[4];

Line 10: a series of <Potion.NUM\_POTIONS> ints separated by a space that represent the potion hot-keys that the user assigns to quickly drink a potion. They are to be read into an array of ints of the size Potion.NUM\_POTIONS.

int []potionHotKeys = new int[Potion.NUM\_POTIONS];

Line 11: a series of 8 ints separated by a space that represent the weapon hot-keys that the user assigns to quickly draw a particular weapon. They need to be read in to an array of 4 java.awt.Point objects, each of which has successive values read in for it’s x and y.

6 0 8 1 6 0 10 0 \*\*weaponHotKeys

Given the data above, four Point objects need to be created and placed in an array of four Points, the first of which has the data (6,0) for its x and y, the second stores (8,1) , the third Point stores (6,0) and the last Point stores (10,0). Note that we are pulling 8 ints out of the string, but filling up four Point objects in an array of Points.

Point []weaponHotKeys = new Point[4];

Line 12: a collection of Strings to fill an ArrayList of String objects that represent the effects that the player was plagued with at the time of being saved, such as being cursed, or engulfed in flames.

[cursed, poisoned] \*\*effects or if the player is healthy, the line might be: [] \*\*effects

Note that you will have to remove the first and last characters from the String (the square brackets) and separate the contents by a comma (“,”).

Line 13: an ArrayList of String that represents which image files the player uses (multiple images for the animation frames of the player). It might look like:

[images/characters/player/saber1.Gif, images/characters/player/saber2.Gif, images/characters/player/saber3.Gif, images/characters/player/saber4.Gif] \*\*images

Given the example described above, the resulting ArrayList should be of size 4, containing *"images/characters/player/saber1.Gif*" at index 0

*"images/characters/player/saber2.Gif*" at index 1

*"images/characters/player/saber3.Gif*" at index 2

and "*images/characters/player/saber4.Gif*" at index 3

Make sure you call the String method .trim() for any element that you store into the ArrayList to clean out any white-characters from the beginning or end of the string.

Line 14: the inventory of player items and their respective frequencies:

[holdall, lockpick:3, viper-assassin-bounty] \*\*items

The line will need to be parsed to fill a single ArrayList of String objects.

ArrayList<String> items = new ArrayList<String>();

Given the example above, line 14 tells us that ***items*** should store 3 Strings with the values *"holdall", "lockpick:3"* and *"viper-assassin-bounty".* The mechanics of the frequencies for each item is handled within the Player object. Again, the player might not have any items, in which case line 14 looks like: [] \*\*items

The resulting ArrayList should exist, but be of size zero.

Line 15: a collection of String objects separated by a comma that represent the spells that the player has learned. They need to be placed into an ArrayList of ***Spell*** objects:

ArrayList<Spell> spells = new ArrayList<Spell>();

A spell has a 1-argument constructor where you send it the name as a String. If the name is wrong, the spell will be assigned the name “!!!UNKNOWN!!!”. If you send to the constructor a String that happens to be the name of a valid spell in the game, the correct Spell object is built that can be placed in the ArrayList. Convenient, right?

Line 16: a LinkedList of ***Teleporter*** data, which might look like this:

[(6 7 17 small\_castle), (0 338 26 world)] \*\*teleporter-memory

The example above represents a Linked List containing two teleporter memories.

A ***Teleporter*** Object has the following constructor:

public Teleporter(int toIndex, int toRow, int toCol, String locType)

If the player saves their game while navigating the greater world, the Teleporter memory will be empty. In that case, Line 16 from data/player.txt will look like this: [] \*\*teleporter-memory

For this situation, the LinkedList of Teleporter objects must exist as an LinkedList of size zero.

Note that you need to first remove the set of square-brackets enclosing the whole String. Then separate the elements by the comma (“,”), then for each element around the comma, separate by space (“ “) to extract the three pieces of information that the Teleporter constructor needs: 3 ints and a String.

Line 17: the inventory of the player's armor: the line consists of elements that has each armor type name followed by the armor frequency:

[Clothes:1, Scalemail:1, Snake-skin:2, Wolf-pelt:3] \*\*armor:armorFrequency

The line needs to be parsed to fill two parallel arrays: an ArrayList of Armor objects and an ArrayList of Integer objects (representing each armor's frequency in the inventory).

ArrayList<Armor> armorSet = new ArrayList<Armor>();  
 ArrayList<Integer> armorFreq = new ArrayList<Integer>();

Like Spell object, Armor has a 1-argument constructor where you send it the name as a String. If the name is wrong, the Armor object will be assigned the name “!!!UNKNOWN!!!”.

In the example above, line 17 tells us that our ***armorSet*** ArrayList needs to contain 4 Armor objects with the names *"Clothes", "Scalemail", "Snake-skin"* and *"Wolf-pelt".* The ***armorFreq*** ArrayList also needs to store 4 Integer objects with the values 1, 1, 2 and 3.

Lines 18-32: the inventory of the player's weapons. Each line consists of the inventory of each of 15 weapon types, and consist of the weapon name followed by the weapon frequency.

[Fists:1] \*\*weapon:weaponFrequency  
[] \*\*weapon:weaponFrequency  
[Dagger:2, Poison-Dagger:1] \*\*weapon:weaponFrequency

These 15 lines need to be parsed to fill two parallel arrays: an array of ArrayLists of ***Weapon*** objects and a parallel array of ArrayLists of Integer objects (representing each weapon's relative frequency in the inventory).

ArrayList<Weapon> [] weapons = new ArrayList[Player.NUM\_WEAPONS];  
 ArrayList<Integer> [] weaponFrequencies = new ArrayList[Player.NUM\_WEAPONS];

Like Armor, Weapon has a 1-argument constructor where you send it the name as a String. If the name is wrong, the Weapon object will be assigned the name “!!!UNKNOWN!!!”.

In the example data above, the 1st line will need to store the Weapon with the name "*Fists*" in the ***weapons*** array at index 0, and the Integer value 1 in the ***weaponFrequencies*** array at index 0. Both ArrayLists of Weapons and Integers stored at index 1 will be empty. But index 2 in the ***weapons*** array and ***weaponFrequencies*** array will have ArrayLists with two elements: The ***weapons*** array will contain an ArrayList that stores the Weapon objects with the names *"Dagger"* and *"Poison-Dagger",* and the ***weaponFrequency*** array will store the Integer values 2 and 1.

Run the main function in FileManager.java to test your code. If it passes both tests, you are almost done! The next page follows with ways to better test your code to make sure it works.

Testing your code:

1) Try setting hot-keys for changing weapons and casting spells. Save the game and quit. Make sure the hot-keys still work upon loading the game again. To get to the hot-keys assignment menu, go to the Inventory with “I”, then the hot-keys menu with “k”. You can assign a weapon to a particular function key from F1 to F4 by first selecting a weapon (Insert/Delete to scroll weapons), then assign a particular key by holding SHIFT-F1 to assign that weapon to the F1 key. SHIFT-F2 would assign it to the F2 key, etc.

2) Acquire a new weapon, armor, spell, potion and item. Make sure that upon saving, you still have those items when the game is loaded up again.

3) Try saving your player while in a location, like a city or a cave. Exit the program. Make sure that upon starting again that you are restored to the same spot in which you saved the game. Also try entering an embedded location, like a dungeon inside of a temple or castle. Again, make sure that you are restored back to the same location where you saved.

4) Try taming a horse and dog. Save the game with your trained animals, and make sure that they are restored with you upon running the game again.

5) Try saving the game while poisoned or cursed, and make sure you are restored with those effects when the game is restarted again.

Extension) Look at how the conversation trees are established in ***NPC.java***. Add a new branch to a conversation tree for something new and interesting.

Extension) Look at how weapons are defined in ***Weapon.java***. Create a new weapon type, and find a way in which it can be acquired in the game. Will it be able to be purchased in an armory, taken from a slain opponent who could randomly spawn with it, or received as a prize for completing a mission? Weapon names that contain the substring ***“flame”*** or ***“bright”*** (or cast the affect ***“flame”*** ) will automatically light the area around a player when in the dark. Weapons with ***“flame”*** in their name will also give the player the option to ignite flammable terrain when close. Weapon names containing the substring ***“blessed”*** will invoke a blessing effect on the player, healing them incrementally over time. Warning: do not make weapons too powerful – they ruin the game by making it too easy to play. New armor, spells and items can also be added, but care must be taken to make each one focus on a very particular advantage, and nothing over-powered.

Extension) Can you create a new monster or NPC type? Some simple graphics for your new monster can easily be added as well. In ***NPC.java***, the array ***public static final String [][] characters*** is in an important order, echoed in the final bytes declared under the array. The byte ***public static final byte TROLL = 25*** means that the troll’s images are at index 25 in the ***characters*** array. The last monster defined is at index 100, denoted by ***public static final byte BRIGANDRIDER = 100***. If you add a new monster or NPC, add the image declaration at the end of the array, and the new byte after value 100 so that you do not disrupt the order of the images in the ***characters*** array.

Extension) ***Mission.java*** defines how missions are assigned and resolved in the game. Adding a new mission type takes some thought and careful code. Try to add a new mission type or variation in the game. Carefully consider what type of NPC would have need for the new mission to be completed. Is there a reputation requirement for the type of NPC that has the mission?

Challenge) Try to complete the main mission without cheating by directly manipulating the player file or reading all the clues in the java files. The game can still be played after completing the main mission, but it should be obvious when you do (resulting in acquiring a special “***crown”***).

Note on encryption and comments:

***CultimaPanel.java*** contains much of the game logic. There is a data field called:

protected static final boolean ENCODE\_FILES = false;

If the data field is set to true, then the program will apply a simple encoding device to make the player file difficult to understand, and less prone to client shinannigans.

The first thing that ***readPlayerFromFile*** does is look to see if the player's data is coming in encoded form by reading a single integer as the first line from the file. If the integer is even, then it will set a flag to let the programknow whether or not it should decode any String that is read in:

String line = decode(input.nextLine().trim(), encoded);

If encoded is true, ***decode*** will return the decoded version of the string that was read in from the file. If encoded is false, ***decode*** will return the unaltered string read in from the file. While working on this assignment, it is suggested that we keep encoding off to make your job easier. Once the method is completed, we can turn encoding on from ***CultimaPanel*** by setting the data-field ***ENOCDE\_FILE***S to true.

Every time you attempt to read a line from ***<data\players.txt>***, you should use the string sent back from the decode method as shown here:

line = decode(input.nextLine().trim(), encoded);

Each line of data from ***<data\players.txt>*** may be followed by a comment describing what the data is.

Because of this, each line from the file should be initially read in as a String. Each comment is preceded by "\*\*". Therefore, each line of text read from the file should ignore any text following "\*\*". The string's substring method can help with this task. Then the information in the remaining string can be examined to see if it is in the correct format, and transformed into the correct type to be sent to the constructor for the ***Player*** object.