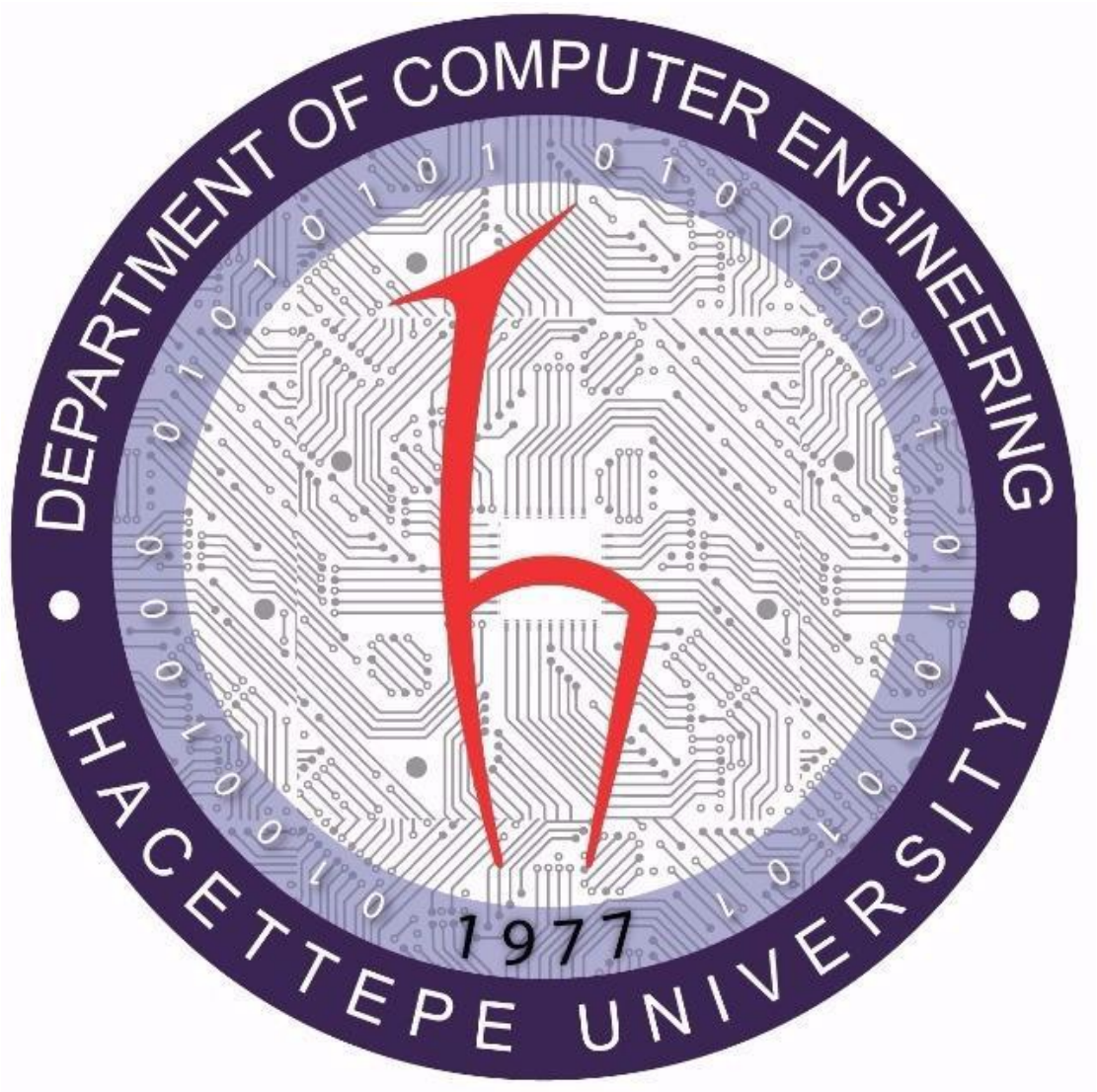

HACETTEPE UNIVERSİTY
DEPARTMENT OF COMPUTER ENGINEERING

GÖKHAN ÖZELOĞLU
21627557

-CALORIE CALCULATION FOR HEALTHY LIFE-

2.1 SOFTWARE USAGE

This program have 4 different input files in text format. Also, it has one output file in the same file type. Input files' names are:

1- **sport.txt**
2- **food.txt**
3- **people.txt**
4- **command.txt** (This file's name can be change. User can enter a name in different name. Command file is determined by user.)

sport.txt, *food.txt* and *people.txt* are read directly by the program. These are not taking from command line as an argument. Only command.txt file takes from command line by user.

1-sport.txt: This file contains up to 100 items. It has three attributes like *sport ID, sport name* and *amount of calorie* that burns if you do sport 1 hour.

2-food.txt: This file contains up 100 items. It has 3 attributes like *food ID, food name* and *amount of calorie* that takes if you eat 1 portion.

3-people.txt: This file contains up 50 items. It has 6 attributes like, *people ID, people name, people genre, people height* and *people's date of birth.*

4-command.txt: This file has no any limitation. Number of item depends on the user. This file writes the appropriate message to the output file.

*All of them files are separated by tab characters.

*There is no any output on the screen. Program just writes to text file as "**monitoring.txt**".

monitoring.txt: This is the output file which writes output after executing commands. It has some kind of special messages. Output file is created by considering command.txt. Program just follows the commands. Every command is written line by line. Lines are separated by asterisk. Number of asterisk is 15 in one line. Last line has to have just message. There is no asterisk on last line.

2.2 ERROR MESSAGES

No error message is expected by this program. Error or exception handling does not the case of this assignment. Nevertheless, I had to use try-catch structure to handle exceptions in unexpected situations. While reading or writing the files, I had to catch ***IOException, FileNotFoundException*** and ***ArrayIndexOutOfBoundsException.*** *IOException* and *FileNotFoundExceptions* are handled because something can goes wrong and can occur unexpected situations. Also, *ArrayIndexOutBoundsExceptions*

occur if the user does not enter the *command.txt* on command line. All of exceptions print out error messages to the screen and execution of program is terminates immediately.

3. SOFTWARE DESIGN NOTES

3.1 Description of the Program

3.1.1. Problem

The problem is creating different classes related to input file, parsing the problem into sub classes and doing every operation in own class.

3.1.2. Solution

I created classes related to input files (`Sport.java`, `People.java`, `Food.java`, `Command.java`). These classes have lots of attributes in field. I choose the attributes according to input files. For example, people have *ID, name, genre, weight, height, birth of date*, so that field were created by these attributes. Also, I defined 3 classes to read files and store the ID's of food, people and sport. These classes are `FoodReader.java, SportReader.java, PeopleReader.java.` I just read the files and stored their ID's in array. Also, these classes have an accessor method to call these arrays in different class'. I used ***`BufferedReader`*** to read files line by line. I checked to see if the files were open in `try-catch` blocks. If something goes wrong and file cannot open or read, program gives an error messages and terminates the running of the program. I provided the terminating the program with `exit` method. So, in order to use exit method, I had to `import static java.lang.System.exit` package. Furthermore, I did import `import java.io.*` package to do file operations. I did import all methods under java.io package.

While reading the files, I used **`while`** loop. This loop implements until the end of file. I read line by line and assign the lines a variable in string type. Then, I splitted the line with **`split()`** method by tab character. Then, I created objects with using the constructor. These constructors have includes all of the attributes of input files. Finally, I appended the ID's of the classes to the array to store them.

On the other classes like **`People, Sport, Food`**, I just defined attributes on field. For instance, food file have 3 different attributes. So, I defined them in `private` situation on the field. I constructed my classes private. In this way, I blocked to change my variables from other classes. Due to constructing private my variables, I had to write *constructor* and *accessor* methods. When I need to create new object from *food, people* or *sport* class, I used class' constructor. After that, I defined accessor methods to get object's information. Also, I defined methods to calculate and update the calories when taking or burning.

While processing the command file, I used `while` loop. Program reads the file line by line, control the which command will be processed and then, calls related methods. The while loop continues until null character. These all operations implements in `readCommandFile` method. This method calls from the `Main` class. In addition, I created an array which names `peopleOnCommandFile`. The program stores the people names which names are used before the `printList` command in this array. I controlled and added the names used at first time on `controlPeopleCommandArray` method. If the name is added before this array, program does not add the name again. Also, I defined `peopleNum` variable to count number of names which used in command file. If program must increment this variable, `controlPeopleCommandArray` method calls the `incrementPeopleNum` method whenever it needs. I also defined `printList` method to implement `prinList` command. I used nested *for loops* and *if-else* conditions because this method takes two parameter which are ***array*** that includes names are used in command file and *peopleArr* includes ID's of *people.txt* file. Firstly, program finds how many names are there in array, because there are *null* characters in the array and program has to know that number of people in this array. This number will be using putting *newline* character. After that, *for loops* are searching until program conditions are ensure. When conditions are ensure, loops are terminated with ***break*** keyword. While writing on the file, there is an *if-else* conditions. The difference between two of them is just putting *"+"* signature. If the calorie result is positive, program puts the *"+"* in front of the number. Other methods are printing asterisk character, calculate the taken and burned calories and count the number of *command.txt* lines. The number of line is using the putting *newline* character end of the *monitoring.txt* file. Program controls the last line of *monitoring.txt* file.

3.1.3. ALGORITHM

Main.java

- Takes command file's name from command line
- Creates a file object from ***`File`*** class and assign the argument to this object as a file name.
- Create an object from ***`Command`*** class and call the ***`readCommandFile`*** method with file object.

```
File file = new File(args[0]);
/* Created an object from Command class to pass argument*/
Command cmd = new Command();
cmd.readCommandFile(file);
```

Command.java

- Creates an array to store people ID's that implemented on command file.(***`peopleOnCommandFile`***)

```
private String[] peopleOnCommandFile = new String[100];
```

- Defines a variable to count how many different people implemented on command file. (***peopleNum***)

```
private int peopleNum = 0;
```

- Defines 4 different object. Three of them are created for reading the people, food and sport files. Last one is for writing the output on the file.

```
FoodReader fr = new FoodReader();
PeopleReader pr = new PeopleReader();
SportReader sr = new SportReader();
 FileWriter fw  = null;
```

Command Class

- Initialize the *lineNumber* variable to know how many line are there in command file. Program puts the *newline* characters with comparing this variable. In order to count line number, *countCommandLineNumber* method calls.

```
int lineNumber = countCommandLineNumber(file);
```

- Creates the "monitoring.txt" file to write output

```
 fw = new FileWriter("monitoring.txt");
```

- Reads the input files by calling the reader methods in order.

```
fr.readFoodFile();
Food[] foodArray = fr.getNewFoodArray();

pr.readPeopleFile();
People[] peopleArray = pr.getNewPeopleArray();

sr.readSportFile();
Sport[] sportArray = sr.getNewSportsArray();
```

- Creates the arrays to store files in their own type's and gets this arrays with accessor methods.
- Program controls the files that open successfully or not in *try-catch* blocks. If program cannot open due to any kind of problem, gives an error message on the screen and terminates the program.
- Command file reads with BufferedReader class.

```
 BufferedReader br = new BufferedReader(new FileReader(file));
```

- Due to not knowing how many lines are there on command file, command files reads and implements in while loop until *null* character.

```
 while ((str = br.readLine()) != null) {
```

- Command file reads line by line and assign a string variable.
- Then, this string variable compare with *if-else* conditions to which command will be processed.
  - If the first 5 character is “print” and 5th character is “(“, program understand that command says that print the person whose ID is interval of parenthesis.

```
if (str.substring(0, 5).equalsIgnoreCase("print") && str.substring(5, 6).equalsIgnoreCase("("))
```

  - Creates the temporary string variable to get ID to find on people ID array.

```
String id = str.substring(6, 11);
```

  - Then, program look for all people ID’s until finding the person. When it finds the ID, writes on the *monitoring.txt* file and for loop is terminated immediately.
  - Then, controls the which line is implementing to put newline character and asterisks.

```
if ( i == lineNumber-1) {
    continue;
} else { //Puts the newline character if this command is not in the last line
    fw.write("\n");
    printAsteriks();
}
```

  - Increment the line number variable after processed this command.
  - If the command is ***`printList`***, program calls the `printList` method.

```
else if (str.equalsIgnoreCase("printList")) {
  printList(peopleOnCommandFile, peopleArray);
```

  - `printList` method takes two parameters. One of them is array which contains all people ID’s and implemented people ID’s on command file.

```
public void printList(String[] array, People[] peopleArr) throws IOException{
```

  - In `printList` method, firstly, count the how many people are there on this array except *null* characters. Then, goes to nested for loop to searching ID’s and comparing two arrays. If two elements are paired, writes on the *command.txt* file. Then, puts the *newline* character if last array’s element is not being written on command file.

```
int x = 0;
int counter = 0;

while ( array[x] != null ) {
    counter++;
    x++;
}
```

```
for ( int i = 0 ; i < array.length; i++ ) {
    if ( i < peopleNum ) {
        for (int j = 0; j < peopleArr.length; j++) {
            if (array[i].equals(peopleArr[j].getPeopleID())) {
```

- After the printList method, asterisks are put with calling *printAsterisk* method.

```
if ( i == lineNumber-1) {
    continue;
} else { //Writes on the out
    fw.write("\n");
    printAsteriks();
}
```

- If the command is not `printList` or print, program goes to else condition.
- Firstly, defines `lineOfCommand` array to store line in string type.

```
String[] lineOfCommand;
```

- Then, line is splitted into 3 elements and added automatically `lineOfCommand` array.

```
lineOfCommand = str.split("\t");
```

- After that, looks at which number there is `lineOfCommand's` second element's first number.

```
if (lineOfCommand[1].substring(0, 1).equals("2")) {
 else if (lineOfCommand[1].substring(0, 1).equals("1")) {
```

- If number is starting with 2, this means person is doing sport, so calories are burned.
- Defines two different variables to looking at indexes.

```
int j, x;
```

- There are two for loops which first one trying to match command's sport ID and sport file's sport ID and second one trying to match command's person ID and people file's person ID.

```
for (j = 0; j < sportArray.length; j++) {
    if (sportArray[j].getSportID().equals(lineOfCommand[1])) {
        break;
    }
}
```

```
for (x = 0; x < peopleArray.length; x++) {
    if (peopleArray[x].getPeopleID().equals(lineOfCommand[0])) {
        break;
    }
}
```

- When they are matched, they terminated for loops with break statement.
- Then, program calculates the calorie that is burned from sport.
- In order to calculate the burned calorie, program calls the `calculateBurnedCalorie` method. This method takes two parameters which are sport's calorie and time of doing sport in minute.

```
double calorie = calculateBurnedCalorie(sportArray[j].getSportCalorie(), Integer.parseInt(lineOfCommand[2]));
```

- This method divides time of sport to 60 and returns in double time.

```
public double calculateBurnedCalorie(int calorie, int sportsMinute) {
    return ((double)sportsMinute / 60) * calorie;
}
```

- After calculating calorie program updates amount of burned calorie.

```
peopleArray[x].calorieBurn(calorie);
```

- Then, writes on the file and adds the asterisks if it is needed.
- Finally, if the person is eating something and taking calorie, last "else if" condition starts processing.
- The similar things are implementing in this block.
- Program tries to match person ID's.

```
for (j = 0; j < foodArray.length; j++) {
    if (foodArray[j].getFoodID().equals(lineOfCommand[1])) {
        break;
    }
}
```

```
for (x = 0; x < peopleArray.length; x++) {
    if (peopleArray[x].getPeopleID().equals(lineOfCommand[0])) {
        break;
    }
}
```

- Then, calculates the taken calorie.

```
int calorie = calculateTakenCalorie(foodArray[j].getFoodCalorie(), Integer.parseInt(lineOfCommand[2]));
```

- Updates the new taken calorie

```
peopleArray[x].calorieUp(calorie);
```

- Writes on the file and adds the asterisks characters if it is needed.