Name: **Matthew Vella**

ID: **0428698M**

Course: **Bachelor of Science (Honors) (Computing Science)**

Department of Computer Science

Object Oriented Programming Project

Study-Unit: **Object Oriented Programming**

Code: **CPS2004**

Plagiarism Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I / We* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Student Name | Signature

Student Name | Signature

Student Name | Signature

Matthew Vella
Student Name | Signature

CPS2004
Course Code

OOP Assignment
Title of work submitted

14/01/2019
Date

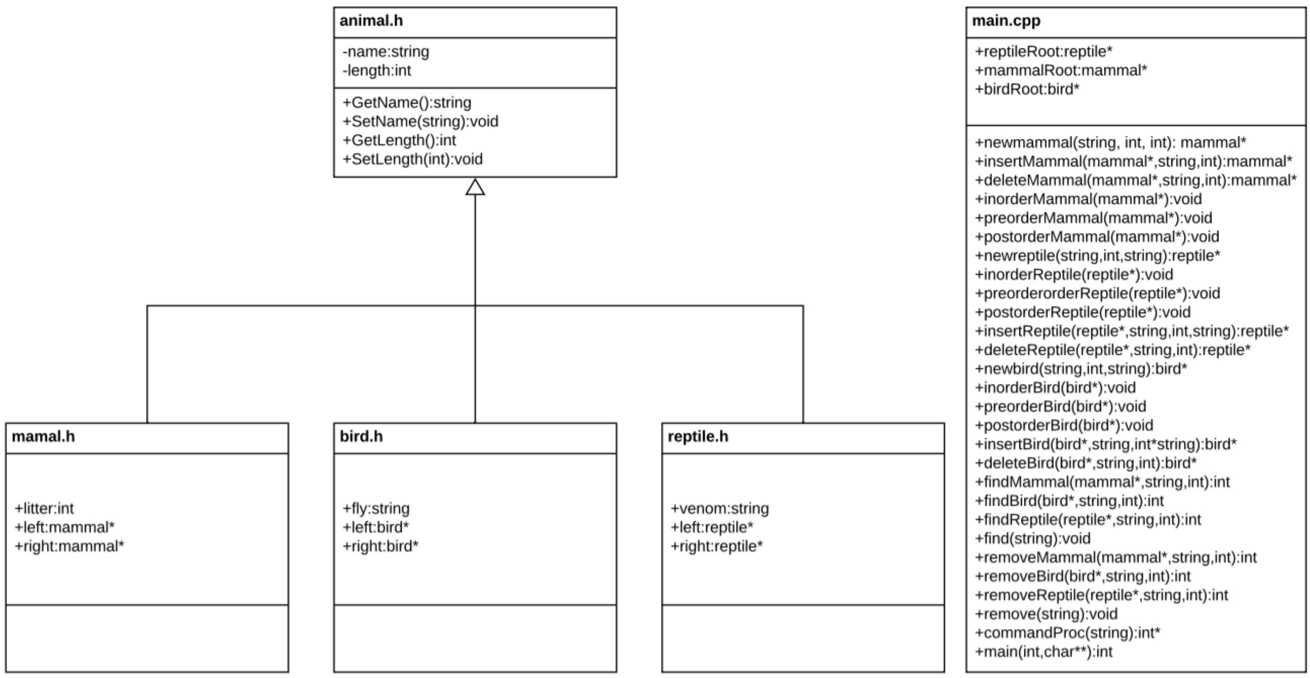
Table of Contents

Task 1 – C++

Design

The program is designed to store Animals of categories Mammals, Reptiles and Birds in a Binary Search Tree. Each type of animal has their own property. Mammal has the average litter size, reptiles can be either venomous or non-venomous and birds can-fly or cannot-fly. The Binary Search Tree has methods for insertion, searching, deletion and traversal algorithms; pre-order, in-order and post-order traversals.

The program is designed with 5 classes, a main class and 4 other classes including a super class and 3 sub classes to take advantage of the object-oriented concept; inheritance where similar properties for the sub classes can be acquired from the parent class. This design is heavy discussed in the limitation section. The system is explained clearer in the UML below.



Implementation

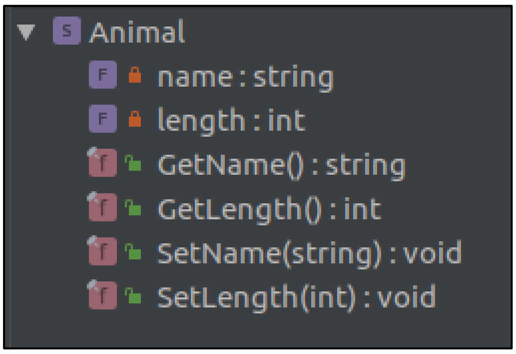
Summary

This program contains 5 classes:

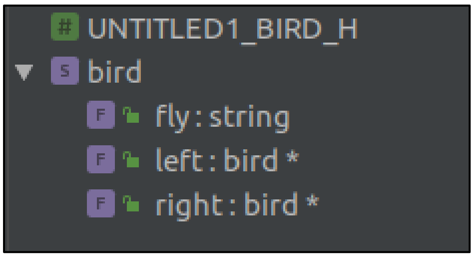
main.cpp: The main class

```
reptileRoot : reptile *
mammalRoot : mammal *
birdRoot : bird *
newmammal(string, int, int) : mammal *
insertMammal(mammal *, string, int, int) : mammal *
deleteMammal(mammal *, string, int) : mammal *
inorderMammal(mammal *) : void
preorderMammal(mammal *) : void
postorderMammal(mammal *) : void
newreptile(string, int, string) : reptile *
inorderReptile(reptile *) : void
preorderReptile(reptile *) : void
postorderReptile(reptile *) : void
insertReptile(reptile *, string, int, string) : reptile *
deleteReptile(reptile *, string, int) : reptile *
newbird(string, int, string) : bird *
inorderBird(bird *) : void
preorderBird(bird *) : void
postorderBird(bird *) : void
insertBird(bird *, string, int, string) : bird *
deleteBird(bird *, string, int) : bird *
findMammal(mammal *, string, int) : int
findBird(bird *, string, int) : int
findReptile(reptile *, string, int) : int
find(string) : void
removeMammal(mammal *, string, int) : int
removeBird(bird *, string, int) : int
removeReptile(reptile *, string, int) : int
remove(string) : void
commandProc(string) : int *
main(int, char * *) : int
```

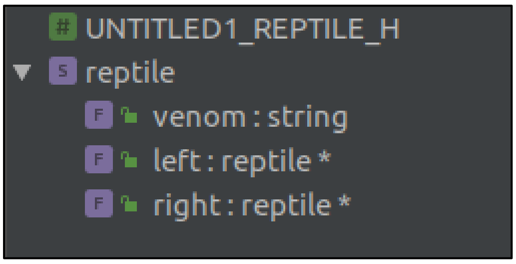
animal.h: Super class of bird.h, reptile.h and mammal.h



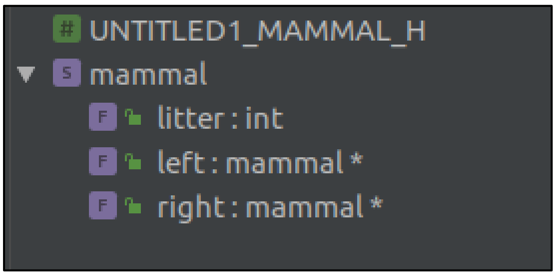
bird.h: subclass



reptile.h: subclass



mammal.h: subclass



Detailed explanation of the code

main.cpp

The subclasses header files are included in the main class. I/O stream to read input/output. String Is to use the string data type. Fstream to read/write to files. Bits/stdc++.h is an implementation file for a precompiled header.

The roots of each type of animal; reptile, mammal and bird were set to null.

```
#include <iostream>
#include <bits/stdc++.h>
#include <string>
#include <fstream>
#include "bird.h"
#include "mammal.h"
#include "reptile.h"

using namespace std;


reptile *reptileRoot = NULL;
mammal *mammalRoot = NULL;
bird *birdRoot = NULL;
```

<u>Creation</u>

**newmammal :* This functions creates a new mammal pointer, name, length, litter and the left and right nodes are set to null. The mammal object, current, is then returned.

```
//a function that creates a new a BST mammal
mammal *newmammal(string n, int item, int l)     //Creates a new mammal pointer, name, le
{                                                //and right nodes to NULL
    mammal *current = new mammal;
    current->SetName(n);
    current->SetLength(item);
    current->litter = l;
    current->left = current->right = NULL;
    return current;                     //Returns the mammal object
}
```

Insertion

**insertMammal:* This function inserts an animal object in the BST. If the tree is empty, a new mammal is returned, creating a new tree for the mammal objects. Else, if the length of the new mammal is smaller than the length of the root node, a recursive call is made to insert the node on the left part of the tree, until a leaf node is found. Else, the recursive call is made on the right part of the tree and finally returning the mammal pointer.

```
// a function that inserts a new mammal with some length in a BST
mammal *insertMammal(mammal *mammal, string n, int length, int v) {
    // When tree is empty, a new mammal is returned
    if (mammal == NULL)
        return newmammal(n, length, v);

    // Otherwise, recursive call
    if (length < mammal->GetLength())          //If length is smaller than the
        mammal->left = insertMammal(mammal->left, n, length, v);
    else
        mammal->right = insertMammal(mammal->right, n, length, v);

    // returning mammal pointer
    return mammal;
}
```

Deletion

**deleteMammal:* This function deletes a node and returns the root. It takes 3 parameters; the root of the tree, the name and the length. The syntax used in the text file for deletion is as follows: Remove Dog. Whenever the 'Remove' token is read from the textfile, the object is searched in the BST. If the object Is found, the length is returned and the deleteMammal function is executed, finally returning the new mammal pointer.

There are 3 conditions for deletion;

- When the node to be deleted is leaf: In this case, the node is simply removed from the tree
- When the node to be deleted has only one child:In this case, the child is copied to the node and the child is deleted
- When the node to be deleted has two children: In this case, firstly the inorder successor of the node is found. Then the contents are copied to the node and the inorder successor is deleted. The inorder successor is only needed when the right child is not empty.

```
mammal *deleteMammal(mammal *mammalRoot, string n, int k) {
    // base case
    if (mammalRoot == NULL)
        return mammalRoot;

    // recursive calls, checking ansistors
    if (mammalRoot->GetLength() > k) {
        mammalRoot->left = deleteMammal(mammalRoot->left, n, k);
        return mammalRoot;
    } else if (mammalRoot->GetLength() < k) {
        mammalRoot->right = deleteMammal(mammalRoot->right, n, k);
        return mammalRoot;
    }

    // We reach here when mammalRoot is the node
    // to be deleted.
```

```
    // If one of the children is empty
    if (mammalRoot->left == NULL) {
        mammal *current = mammalRoot->right;
        if (mammalRoot->GetName().compare(n) == 0) {
            delete (mammalRoot);
        }
        return current;
    } else if (mammalRoot->right == NULL) {
        mammal *current = mammalRoot->left;
        if (mammalRoot->GetName().compare(n) == 0) {
            delete (mammalRoot);
        }
        return current;
    }
```

```
        // If both children exist
    else {

        mammal *succParent = mammalRoot->right;

        // Find successor
        mammal *succ = mammalRoot->right;
        while (succ->left != NULL) {
            succParent = succ;
            succ = succ->left;
        }

        // Delete successor.  Since successor
        // is always left child of its parent
        // we can safely make successor's right
        // right child as left of its parent.
        succParent->left = succ->right;

        // Copy Successor Data to mammalRoot
        mammalRoot->SetLength(succ->GetLength());
        mammalRoot->SetName(succ->GetName());
        mammalRoot->litter = succ->litter;
        // Delete Successor and return mammalRoot
        if (succ->GetName().compare(n) == 0) {
            delete (succ);
        }
        return mammalRoot;
    }
}
```

Traversals

Traversal algorithms perform the; pre-order, in-order and post-order traversal on each type of BST; mammal, reptile and bird.

```
// a function that does inorderMammal traversal of a BST
void inorderMammal(mammal *mammalRoot) {
    if (mammalRoot != NULL) {
        inorderMammal(mammalRoot->left);
        cout << mammalRoot->GetName() << " " << mammalRoot->GetLength() << " " << mammalRoot->litter << " Mammal"
             << endl;
        inorderMammal(mammalRoot->right);
    }
}

// a function that does preorder traversal of a BST
void preorderMammal(mammal *mammalRoot) {
    if (mammalRoot != NULL) {
        cout << mammalRoot->GetName() << " " << mammalRoot->GetLength() << " " << mammalRoot->litter << " Mammal"
             << endl;
        preorderMammal(mammalRoot->left);
        preorderMammal(mammalRoot->right);
    }
}

// a function that does postorder traversal of a BST
void postorderMammal(mammal *mammalRoot) {
    if (mammalRoot != NULL) {
        postorderMammal(mammalRoot->left);
        postorderMammal(mammalRoot->right);
        cout << mammalRoot->GetName() << " " << mammalRoot->GetLength() << " " << mammalRoot->litter << " Mammal"
             << endl;
    }
}
```

Searching

findMammal(): This functions finds a particular node recursively. If the root node is null, the value is returned. Else, the node is found recursively, either to the left or right part of the BST.

```
int findMammal(mammal *mammalRoot, string n, int val) {
    if (mammalRoot != NULL) {
        val = findMammal(mammalRoot->left, n, val);
        val = findMammal(mammalRoot->right, n, val);
        if (mammalRoot->GetName().compare(n) == 0) {
            cout << mammalRoot->GetName() << " " << mammalRoot->GetLength() << " " << mammalRoot->litter << " Mammal"
                 << endl;
            return val + 1;
        }
    }
    return val;
}
```

Command Processor

**commandProc():* This function processes the commands read from the text file. The program searches first searches for the main tokens; Insert, Find and Remove. If the Insert token is found, the program checks the next token whether it's a 'reptile', 'mammal' or bird and after checks for their properties. If the commands are in correct format, for each of these cases, the functions; insertReptile(), insertMammal() or insertBird() are called respectively, passing the respective properties to the functions. For the find token, the find() is called whilst for the Remove token, the remove() is called. All data was carefully validated so that the program does not crash when encountering an unknown token.

Main Method

main(): This is the main method of the program. This method opens the file test.txt and starts executing the commandProc() function inside a while loop to start processing the commands until the end of the file. When all the commands have executed, this method prints 3 BSTs for each animal type; reptile, mammal and bird, using an in-order traversal. The file is then closed.

```
int main(int argc, char *argv[]) {
    string n = string("Jewel");
    string s;
    ifstream infile;
    infile.open("test.txt");   //open text file
    if (argc == 2) {
        infile.close();
        infile.open(argv[1]);
    }
    while (getline(infile, s)) {
        commandProc(s);      //Calling commandProc which processes commands.
    }
    //Printing the traversals:
    cout << endl;
    printf("Reptile BST InOrder Travesal -> \n");     //Reptile Traversal
    inorderReptile(reptileRoot);
    cout << endl;
    printf("Mammal BST InOrder  -> \n");              //Mammal Traversal
    inorderMammal(mammalRoot);
    cout << endl;
    printf("Bird BST InOrder Traversal -> \n");      //Bird Traversal
    inorderBird(birdRoot);
    cout << endl;


    infile.close();    //closing the file
    return 0;
}
```

animal.h:

This is the superclass of the subclasses; reptile.h, bird.h and mammal.h. It has the common attributes 'name' and 'length' and enforces encapsulation to give access to the private variables only via getters and setters.

```
#pragma once

#include <memory>
#include <string>

class Animal {

private:

    std::string name;
    int length;

public:

    std::string GetName() {
        return name;
    }
    int GetLength(){
        return length;
    }

    void SetName(std::string name) {this -> name = name;}
    void SetLength(int length){this-> length = length;}


};
```

bird.h: This is a subclass which inherits from animal.h.

```
#ifndef UNTITLED1_BIRD_H
#define UNTITLED1_BIRD_H
#include "animal.h"


class bird : public Animal {


public:
    std::string fly;

    bird *left , *right;


};


#endif //UNTITLED1_BIRD_H

```

mammal.h: This is a subclass which inherits from animal.h.

```
#ifndef UNTITLED1_MAMMAL_H
#define UNTITLED1_MAMMAL_H

#include "animal.h"


class mammal : public Animal {

public:
    int litter;
    mammal *left ,*right ;


};

#endif //UNTITLED1_MAMMAL_H
```

reptile.h: This is a subclass which inherits from animal.h.

```
#ifndef UNTITLED1_REPTILE_H
#define UNTITLED1_REPTILE_H
#include "animal.h"

class reptile : public Animal {


public:
    std::string venom;
    reptile *left, *right;


};

#endif //UNTITLED1_REPTILE_H
```

Testing

To test the system, the test file given by the lecturer was used. The test file was saved as a .txt file and stored in the same directory as the project file.

```
Insert mammal cat 60 4
Insert reptile viper 200 venomous
Insert reptile chameleon 12 non-venomous
Insert bird eagle 80 can-fly
Insert bird ostrich 150 cannot-fly
Find viper
Remove eagle
```

There are two ways to run this system:

- By running manually, the commands from the Terminal:

```
matt@ubuntu:~/Desktop/CPP-Final$ g++ -std=c++11 -Wall main.cpp
matt@ubuntu:~/Desktop/CPP-Final$ ./a.out
viper 200 venomous Reptile
Found Results: 0, 1, 0 in Mammals, Reptiles and Birds respectively
eagle 80 can-fly Bird
Deleting 80

Reptile BST InOrder Travesal ->
chameleon 12 non-venomous Reptile
viper 200 venomous Reptile

Mammal BST InOrder  ->
cat 60 4 Mammal

Bird BST InOrder Traversal ->
ostrich 150 cannot-fly Bird
```

- By running the run.sh file in the project directory.

  The contents of the shell file:

```
#!/bin/bash
g++ -std=c++11 -Wall main.cpp./a.out

/bin/bash
```

Both ways work Successfully and output the correct result.

Limitations and possible improvements

The above program implements a binary search tree capable of storing and sorting different types of Animals (birds, reptiles and mammals) that all inherit from the superclass 'animal.h'. This approach allows common attributes like length to be written once in the superclass , the ninherited throughout the rest of the program.

Although the given solution provides the correct functionality, an improvement to this program would be to extend this concept further while also adding generalization using templates. This would have allowed methods for insertion, removal, searching and traversal algorithms to be written just once, accepting a template and therefore would work for any type of object; instead of having to rewrite the same algorithm for all the different objects as done for this system. Example: instead of having to write the functions: insertReptile(), insertMammal() and insertBird(), a single template would replace these example:

```
template <typeName myType> void display(myType x) {
        cout << “you have passed ” << x << endl;
}
```

The above function can accept any type of object. Therefore, it can be called with any parameter, such as; display(10), display(“Hello”) etc. This proves the power of generelisations using these templates, and this would have stepped the game in this program, as initially intended.

Also, an improvement to this program would be a separate 'BST.h 'class, which encapsulates all the general functionality of inserting, finding, deleting and traversing through the BSTs, allowing for any generic object. This would increase code reusability and would reduce code repetition where a function was needed for each animal type.

Such an improvement would have also made it easier to make changes to the logic of the program, as this would only need to be implemented to a single function, rather than every variation of the same function.

The proposed improvement was the first choice of the design solution but was not successfully implemented due to time and difficulties incurred. The concept of generics and template was implemented in Part 2 of this assignment to generalize a linked list and worked very well.

References

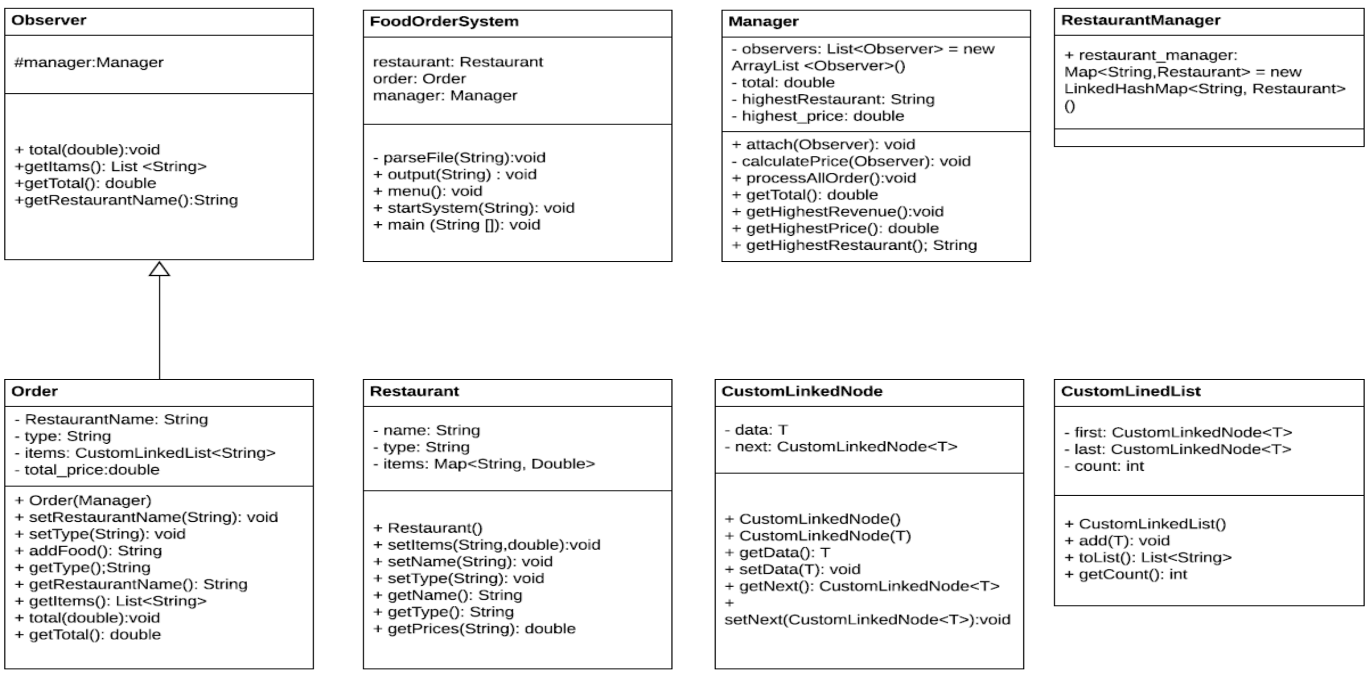
- Class notes

Task 2 – Java

Design and OOP Principles

The program is designed to process food orders from multiple restaurants entered through a text file. Each restaurant has a menu with food and prices and can offer deliveries, takeaways or both. The system will finally output the total price of all the restaurants and the highest revenue restaurant along with the amount selling. The output is also displayed in a text file inside the folder as Receipt.txt.

The program is written in Java, an object-oriented language, which promotes code reuse and a clear modular structure for programs along with garbage collection for memory management. The Observer Design pattern was also used when designing the system structure to note the state of objects and get notified whenever there is any change. This promotes loose coupling. Full advantage of OOP principles was also taken when designing and building this system. Encapsulation was used in the classes: Resturant.java, Manager.java and Order.Java, by declaring all the variables as private and provide getters and setters to modify and view the variable values. This enforces data hiding. Abstraction was used in the Observer.java class and Order.java. Observer.java was declared as an abstract class, with abstract methods without implementations. The Order.java class inherits from the Observer.java and provides implementations for the abstract methods. Using this technique, the implementation details are hidden from the user and the abstract class cannot be instantiated meaning that no objects can be made of the class. Inheritance was also used between these 2 classes, so that the Order.java (sub class) class could acquire the properties of the Observer.java class (the superclass). This is shown clearer in the UML below.

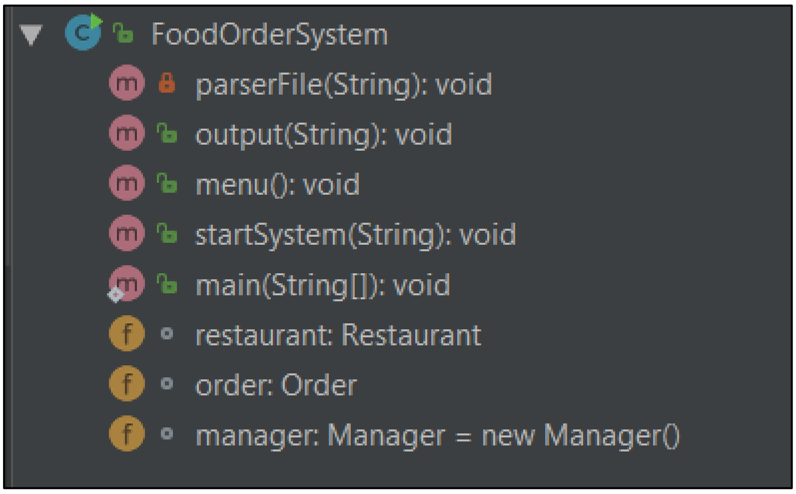


Implementation

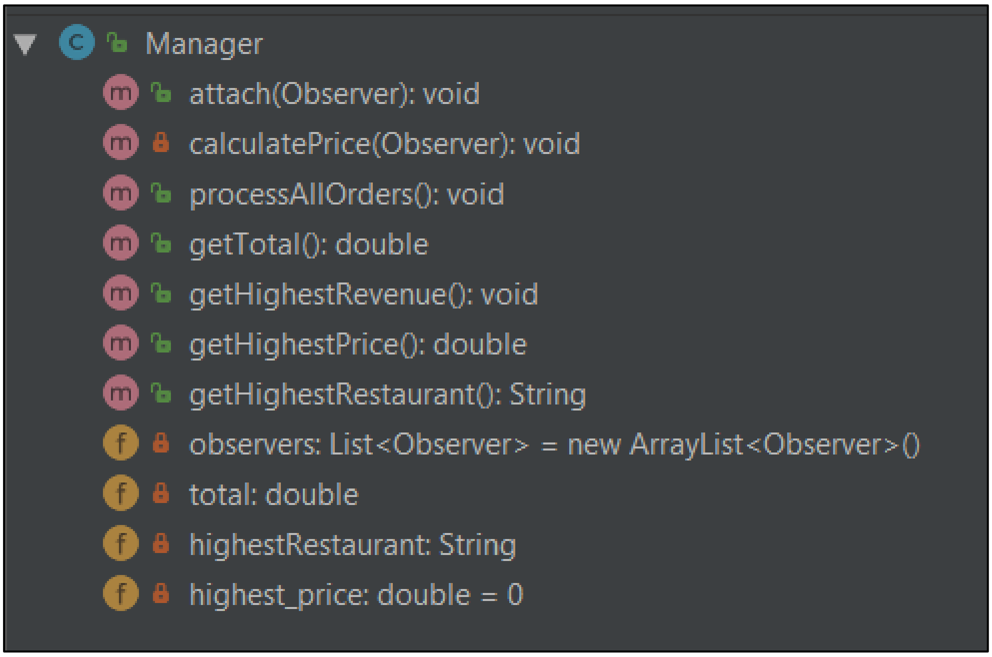
Summary

The program has 8 classes:

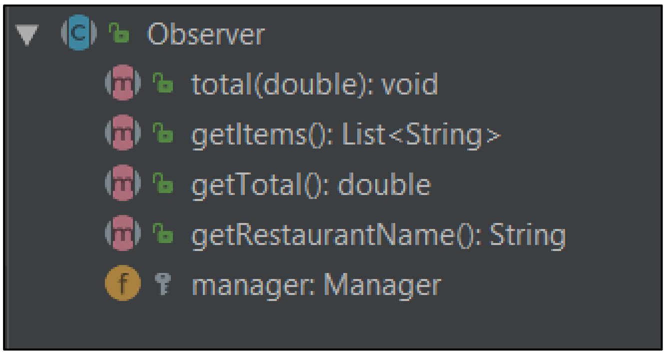
- **FoodOrderSystem.java:** The Main class



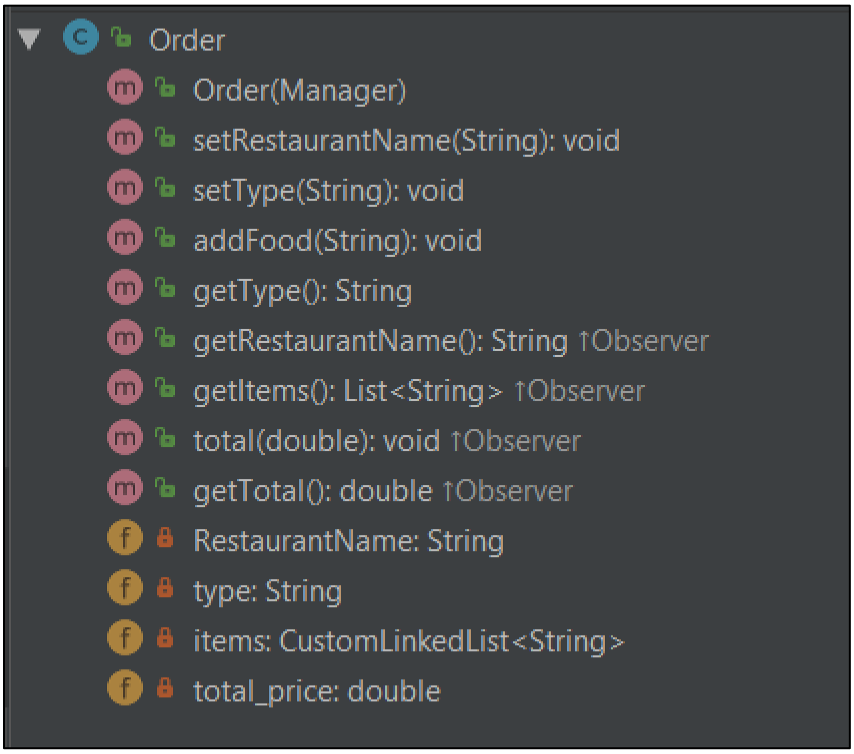
- **Manager.java:** Works out the total revenue, the restaurant with highest revenue and manages all the orders.



- **Observer.java:** An abstract class for implementing the observer design pattern

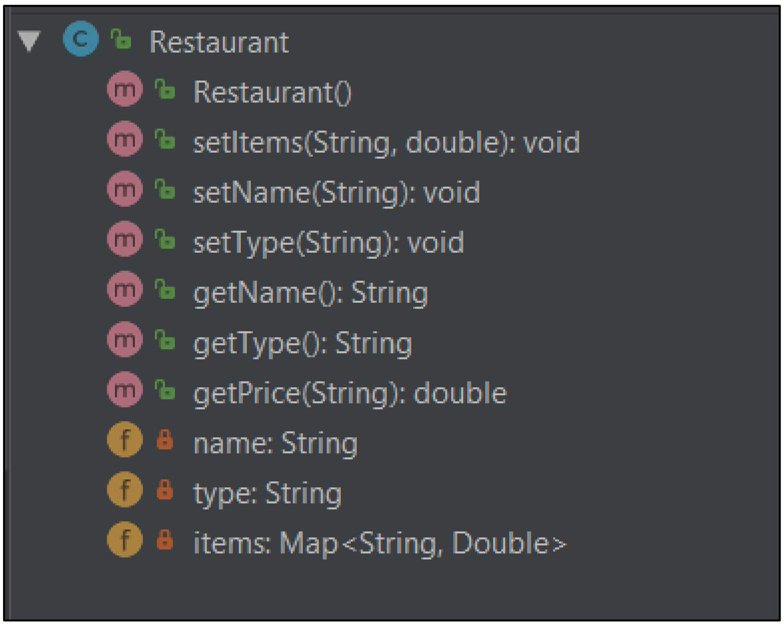


- **Order.java:** Initializes all the orders and passes the orders to Manager.java

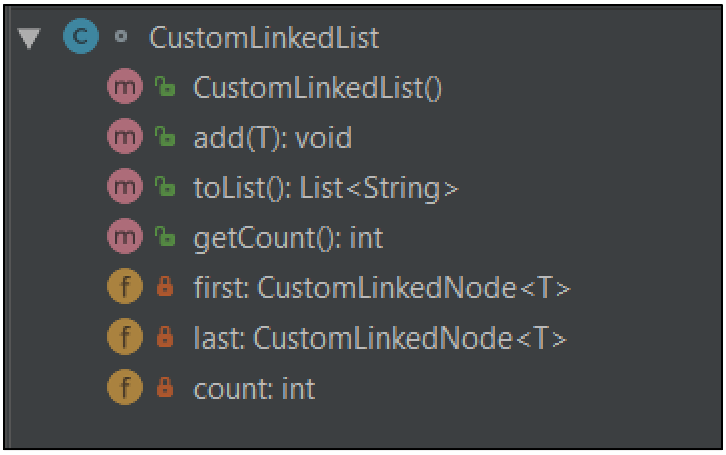


- **RestaurantManager.java:** Class which stores all the restaurants

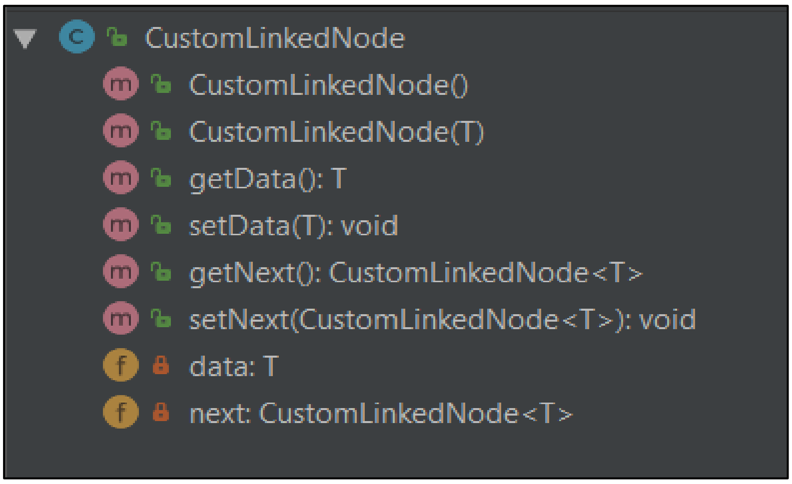
- **Restaurant.java:** Restaurant data class



- **CustomLinkedList.java:** The class with the custom generic linked list.



- **CustomLinkedNode.java**



Detailed explanation of the code

FoodOrderSystem.java

This is the main class of the system. The program is compiled and ran, by running the shell script 'Run System', where in the folder there's a script available both Windows and Unix based systems. The program first checks whether an argument for a test file was given or not. If no argument is given with the test file, the program outputs, "No test file was inputted". If an argument for a test file is given but the there are no such text files, an error is outputted, "Test file does not exist.". Else, if the test file is correct, the system will run the method 'startSystem()', having the test file as the argument. The method 'startSystem()' first calls the function 'usage()', which prints a menu. Then, the method tries to open the test file in a try-catch block, for safety to prevent the program from crashing if something goes wrong. In such case, the stack trace is printed so that it's easier to understand what went wrong. If the test-file is opened successfully, the functions 'processAllOrders()' and 'getHighestRevenue()' are called from the Manager.java class, which works out the total amount of all the orders and the calculates the highest revenue restaurant respectively. These values; total amount, and the highest revenue restaurant amount & name are returned to this method and printed out to console.

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;

public class FoodOrderSystem {

    Restaurant restaurant;
    Order order;

    Manager manager = new Manager();

    private void parserFile(String filename) throws Exception {
        File inputFile = new File(filename);

        //If input file doesn't exist
        if (!inputFile.exists()) {
            output( str: "Input File is not exist !");
            return;
        }

        //If exists, file is read
        BufferedReader reader = new BufferedReader(new FileReader(inputFile));
        String line = null;

```

```java

                if (line.startsWith("OrderItem")) {        //Token = OrderItem
                    String[] split = line.split( regex: " ");
                    String foodName = split[1].trim();
                    order.addFood(foodName);           //Add food in CustomLinkedList
                }
            }
        }

        public void output(String str) {

            System.out.println(str);
        }

        public void menu() {
            output( str: "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|");
            output( str: "|                                              |");
            output( str: "|            Food Ordering System              |");
            output( str: "|                                              |");
            output( str: "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|");
        }

```

```java
        while ((line = reader.readLine()) != null) {
            if (line.startsWith("BeginRestaurant")) {   //If Token = BeginRestaurant
                restaurant = new Restaurant();
                String[] split = line.split( regex: " ");
                String restName = split[1].trim();  //First string = restName
                String typeName = split[2].trim();  //Second string - typeName
                restaurant.setName(restName);
                restaurant.setType(typeName);
                RestaurantManager.restaurant_manager.put(restName, restaurant); //Saves Restaurant in LinkedH
            }

            if (line.startsWith("Item")) {             //If Token = Item
                String[] split = line.split( regex: " ");
                String foodName = split[1].trim();
                double price = Double.parseDouble(split[2].trim());
                restaurant.setItems(foodName, price);
            }

            if (line.startsWith("BeginOrderList")) {   //If Token = BeginOrderList
                continue;
            }

            if (line.startsWith("BeginOrder")) {       //If Token = BeginOrder
                order = new Order(manager);
                String[] split = line.split( regex: " ");
                String restName = split[1].trim();
                String typeName = split[2].trim();
                order.setRestaurantName(restName);     //Set restName and typeName for order
                order.setType(typeName);
            }
```

```
            if (line.startsWith("OrderItem")) {        //Token = OrderItem
                String[] split = line.split( regex: " ");
                String foodName = split[1].trim();
                order.addFood(foodName);            //Add food in CustomLinkedList
            }
        }
    }

    public void output(String str) {

        System.out.println(str);
    }

    public void menu() {
        output( str: "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|");
        output( str: "|                                                    |");
        output( str: "|              Food Ordering System                  |");
        output( str: "|                                                    |");
        output( str: "|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|");
    }

```

```
    public void startSystem(String testFile) {
        menu();  //Menu

        try {       //Try to open the text file
            parserFile( filename: "test.txt");
        } catch (Exception e) {
            e.printStackTrace();
        }

        //If successful
        manager.processAllOrders();
        manager.getHighestRevenue();

        double total = manager.getTotal();
        double highestPrice = manager.getHighestPrice();
        String highestRestaurant = manager.getHighestRestaurant();

        //Printing Results
        output( str: "Total Price : " + total + "$");
        output( str: "Restaurant with highest revenue : " + highestRestaurant);
        output( str: "Highest Total Price : " + highestPrice + "$");
    }

```

```java
    public static void main(String[] args) {
        if (args.length != 1) {     //No arguments for test file
            System.out.println("No test file was inputted.");
            return;
        }
        if (!new File(args[0]).exists()) {     //Text file doesn't exist
            System.out.println("Test file does not exist.");
            return;
        }
        new FoodOrderSystem().startSystem(args[0]);   //If successful, system starts
    }
}
```

Manager.java

This class works out the total revenue, the restaurant with the highest revenue and manages all orders. This class is also designed with the Observer design pattern. A list of observers is first created to watch over the objects in the manager class. The observers are attached via the 'attach()' method.

The 'calculatePrice()', calculates the total price. The method gets the restaurant name. and the list of items of the order which store them in a list of strings. An enhanced forloop is used to loop through all the items, get the price of each item from the restaurant class and accumulates the total price of the order. Finally, the total price is set.

```java
    private void calculatePrice(Observer order) {
        Restaurant restaurant = RestaurantManager.restaurant_manager.get(order.getRestaurantName());
        List<String> items = order.getItems();
        double total_price = 0;
        for (String item : items) {   //For each item in items, price is calculated
            double price = restaurant.getPrice(item);
            if (price != -1) {
                total_price += price;    //Total_price = accumulated price
            }
        }
        order.total(total_price);   //Total price is set
    }
```

The 'processAllOrders()' method, calculates the price of each order by calling the above 'calculatePrice()' method. For all the orders, the total price Is set.

```java
    public void processAllOrders() {
        for (Observer order : observers) {   //Foreach order, the price is calculated
            calculatePrice(order);
        }
        for (Observer order : observers) {  //For all orders, total price is set
            total += order.getTotal();
        }
    }
```

The ‘getTotal()’ method, returns the total price. The ‘getHighestRevenue()’ method finds the restaurant with the highest revenue. Using an enhanced for loop, the method iterates through all the orders, calculates the total revenue and compares the total revenue with the current highest revenue gained restaurant. The name of the restaurant with the highest revenue is saved in local variable highestRestaurant.

```
    public void getHighestRevenue() {
        for (Observer order : observers) {    //For all orders, Total price is calculcated
            double total = order.getTotal();
            if (highest_price < total) {      //The highest priced order is saved in highe
                highest_price = total;
                highestRestaurant = order.getRestaurantName();   //The name of the restaur
            }
        }
    }
```

The methods ‘getHighestPrice()’ and ‘getHighestResturant()’, return the highestPrice and the highestRestaurant respectively.

```
    public double getHighestPrice() {        //Highest priced order is returne
        return highest_price;
    }

    public String getHighestRestaurant() {      //Highest priced restaurant is
        return highestRestaurant;
    }
}
```

Observer.Java

This is an abstract class for implementing the Observer Design pattern. This class has four abstract methods: ‘total()’, ‘getItems()’,’getTotal()’ and ‘getRestuarantName()’ which are ovverriden in the subclass Order.Java and explained below.

```
import java.util.List;

public abstract class Observer {
    protected Manager manager;

    public abstract void total(double price);      //to store total price for each order

    public abstract List<String> getItems();       // to get all food items

    public abstract double getTotal();             //get total price

    public abstract String getRestaurantName();     //get restaurant name
}
```

Order.Java

This class initialised all the orders and passes the orders to the Manager.Java class. This class is also designed with the Observer Design pattern. It inherits from the Observer class. A custom linked list is created above which will store the food items.

The constructor ‘order()’, which takes a manager object, creates a new instance of the custom linked list called ‘items’, sets the manager and attaches an observer to the manager instance.

```
    public Order(Manager manager) {                                    //it
        items = new CustomLinkedList<String>();
        this.manager = manager;
        this.manager.attach( observer: this);
    }
```

The methods ‘setRestaurantName()’, ‘setType()’, set the restaurant name and restaurant type respectively. The method ‘addFood()’, accepts a food item and adds it to the custom linked list.

```
    public void setRestaurantName(String name) {
        this.RestaurantName = name;
    }

    public void setType(String type) {
        this.type = type;
    }

    public void addFood(String item) {
        items.add(item);
    }
```

Since the Order.Java class inherits from the abstract class 'Observer.Java', the Order class has to override the abstract methods; 'getResturantName()' returns the name of the restaurant, 'getItems()' returns a list of items by converting the custom linked list to a normal list, 'total()' sets the total price whilst 'getTotal()' returns the total price.

```java
    @Override
    public String getRestaurantName() {
        return RestaurantName;
    }


    @Override
    public List<String> getItems() {
        return items.toList();
    }

    @Override
    public void total(double price) {
        this.total_price = price;
    }

    @Override
    public double getTotal() {
        return total_price;
    }
}

```

CustomLinkedList.java

This class contains a custom generic linked list, which is used in the Order.Java class to store the food items. The advantage of writing generic data structures is that types which the data structure accepts can be specified later by passing them through a parameter.

The ‘CustomLinkedList’ has private 2 linked nodes: ‘CustomLinkedNode<T> first’, stores the first node of the linked list and ‘CustomLinkedNode<T> last’ stores the last node of the linked list. It also has a variable ‘count’ which holds the count of the nodes in the linked list. ‘CustomLinkedNode<T> next’ is a pointer for the next node. ‘private T data’ stores the data. The ‘CustomLinkedNode’ class is defined inside the ‘CustomLinkedList’ class.

```
class CustomLinkedList<T> {
    private CustomLinkedNode<T> first;            //s
    private CustomLinkedNode<T> last;            //st
    private int count;                            //t
```

The constructor of the CustomLinkedList creates a new linked list object. Sets the first node and sets the last node to the first node.

```
    public CustomLinkedList() {             //Creating a New Linked
        CustomLinkedNode<T> newLiked = new CustomLinkedNode<T>();     /
        this.first = newLiked;
        this.last = this.first;
    }
```

The method ‘add()’, adds a new node to the linked list and adjusts it accordingly.

```
    public void add(T data) {               //adds a new node to the
        CustomLinkedNode<T> newData = new CustomLinkedNode<T>(data);
        if (this.first.getData() == null) {        //this checks if t
            this.first = newData;
            this.last = this.first;
        } else {                  //if not it adds the node to the
            this.last.setNext(newData);
            this.last = newData;
        }
        count++;
    }
```

The method ‘toList()’, typecasts a tree into a list.

```java
public List<String> toList() {                  //Just to type cast a the tree into a List
    List<String> l = new ArrayList<String>(count);
    boolean allStored = false;
    int count = 0;
    CustomLinkedNode<T> crr = first;
    while (!allStored) {                //just to check if tree is completely stored in the list or not
        if (crr.getData() != null) {
            l.add(crr.getData().toString());
            //System.out.print(crr.getData().toString()+", ");
            if (crr.getNext() != null) {            //if the Tree is not reached it's end
                CustomLinkedNode<T> crrNext = crr.getNext();
                crr = crrNext;
            } else {                        //if it has reached the end it changes the value of allStored
                //System.out.println();
                allStored = true;
            }
        } else {
            allStored = true;
        }
    }
    return l;
}
```

The method ‘getCount()’, returns the count of nodes of the tree.

```java
    public int getCount()  {        //re
        return count;
    }
}
```

LinkedListNode.java

The 'CustomLinkedNode' class, represents a node of the linked list. The class has 2 main variables; 'data' to store the data and 'next' as a pointer pointing to the next node. The parameter-less constructor initializes both values to null. The constructor which takes an object, creates a new node object and stores the data. The method 'getData()' returns the data of a node, whilst the method 'setData()' sets the value of the node. The method 'getNext()' returns the pointer to the next node and 'setNext()' sets the pointer of the next node.

```java
    private class CustomLinkedNode<T> {            //The node of the Linked
        private T data;                            //stores the data for th
        private CustomLinkedNode<T> next;         //pointer for the next nod

        public CustomLinkedNode() {              //If the link list is initia
            this.data = null;
            this.next = null;
        }

        public CustomLinkedNode(T obj) {          //Creates the New node obje
            this.data = obj;
            this.next = null;
        }

        public T getData() {                       //returns the data in th
            return data;
        }

        public void setData(T data) {               //sets the value of the
            this.data = data;
        }

        public CustomLinkedNode<T> getNext() {       //returns the pointer
            return next;
        }

        public void setNext(CustomLinkedNode<T> next) {    //sets the point
            this.next = next;
        }

```

Restaurant.Java

This class is the Restaurant data class. It stores the restaurant name and the type, either delivery, take away or both, as Strings. A hash table is used to store the restaurant items; food name as key and price as value.

```
public class Restaurant {
    private String name; // Restaurant name
    private String type; // type <Delivery|take awa
    private Map<String, Double> items; // restauran

```

The constructor creates a new LinkedHashMap called items.

```
    public Restaurant() {
        items = new LinkedHashMap<String, Double>();
    }

```

Getters and setters

```
    public void setItems(String name, double price) {     //items a

        items.put(name, price);
    }

    public void setName(String name) {                    //name is
        this.name = name;
    }

    public void setType(String type) {                    //type is
        this.type = type;
    }

    public String getName() {                             //Name is r
        return name;
    }

    public String getType() {                             //Type is re
        return type;
    }

```

The method 'getPrice()' calculates the price for a single item by searching the item by food name (key) in the hash table, and return the value if the item is found.

```
    public double getPrice(String item) {
        if (items.containsKey(item)) {
            return items.get(item);
        } else return -1; //item not found
    }
}
```

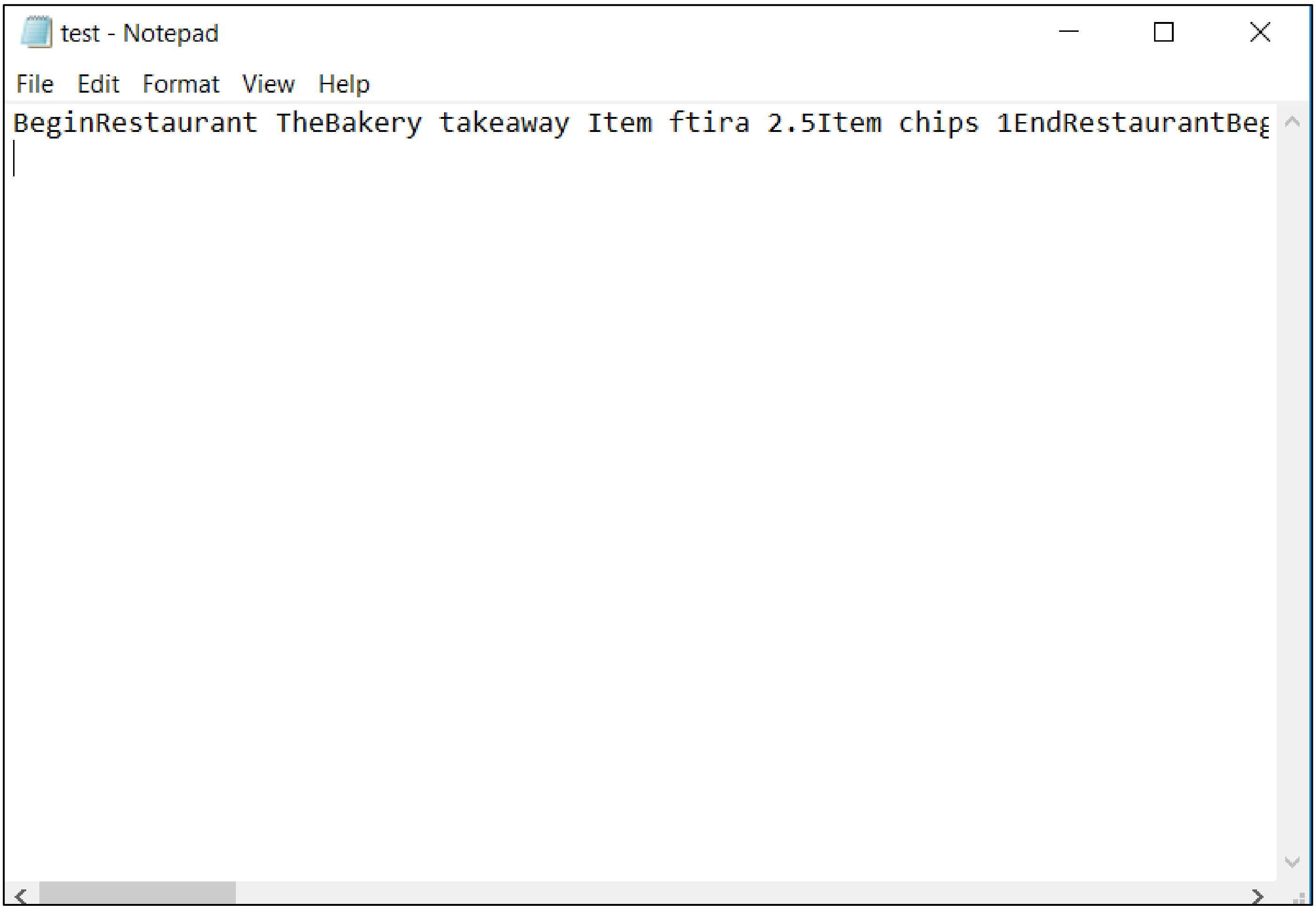
ResturantManager.Java

This class stores all the restaurants. The restaurants are stores in a LinkedHashMap aswell; the key being the restaurant name, and the value the restaurant object.

```
import java.util.Map;

import java.util.LinkedHashMap;

public class RestaurantManager {

    //Linked hashed map maintains insertion order of keys.
    public static Map<String, Restaurant> restaurant_manager = new LinkedHashMap<String, Restaurant>();
}
```

Testing

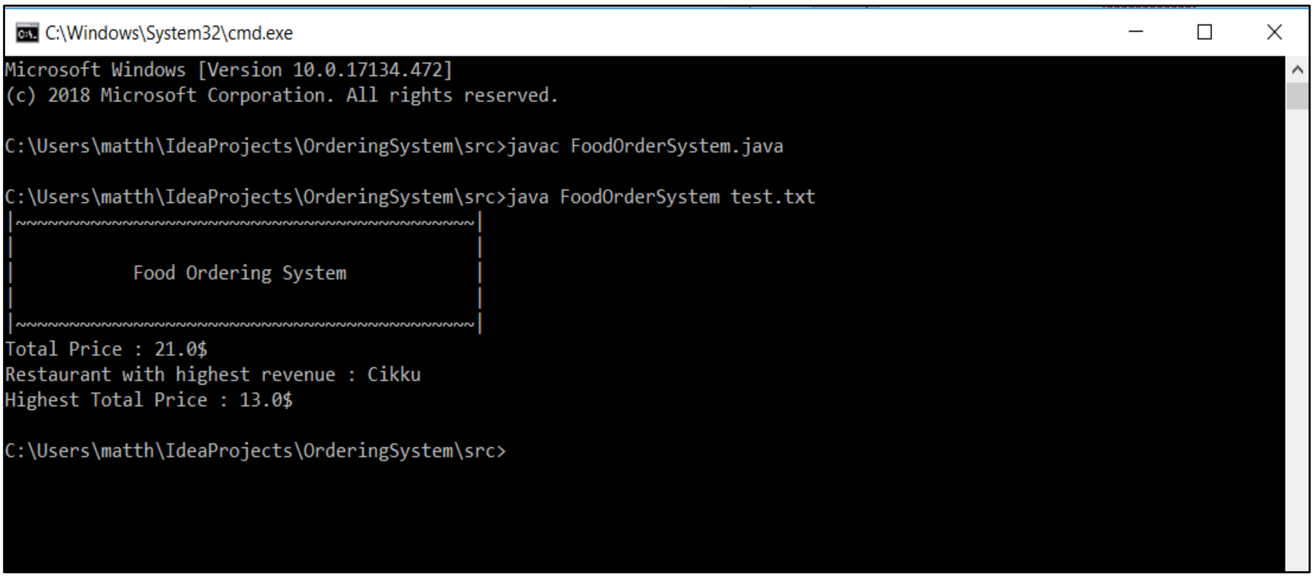
To test the system, the test file given by the lecturer was used. The test file was saved as a .txt file and passed as an argument to the program.



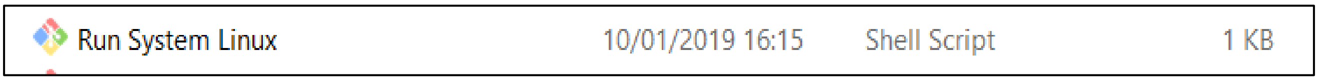
The program can be run in 2 ways:

- By compiling and running the code manually:

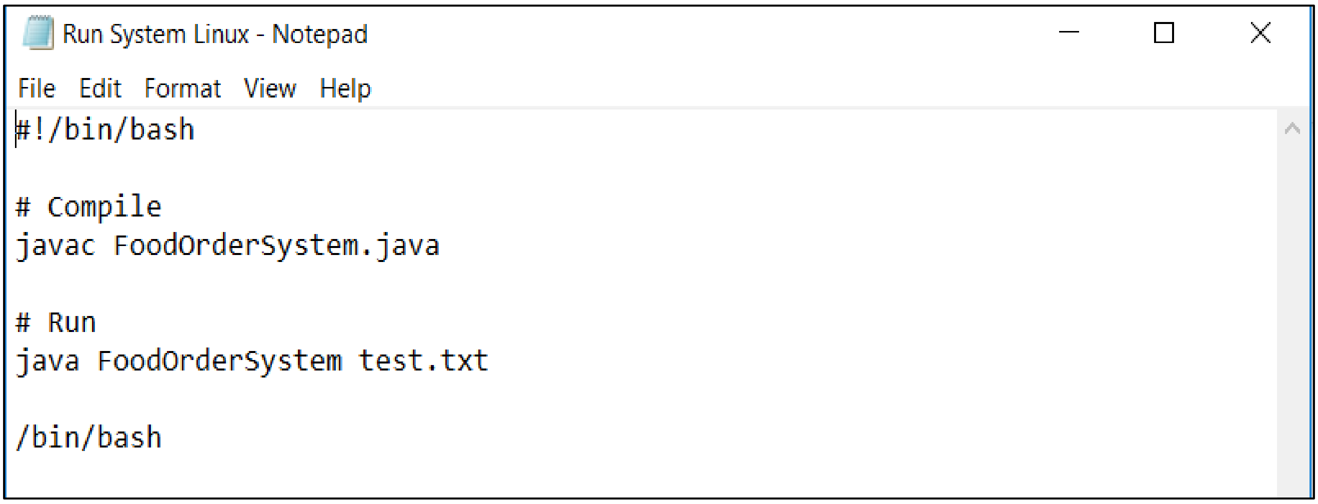
  Successful Result



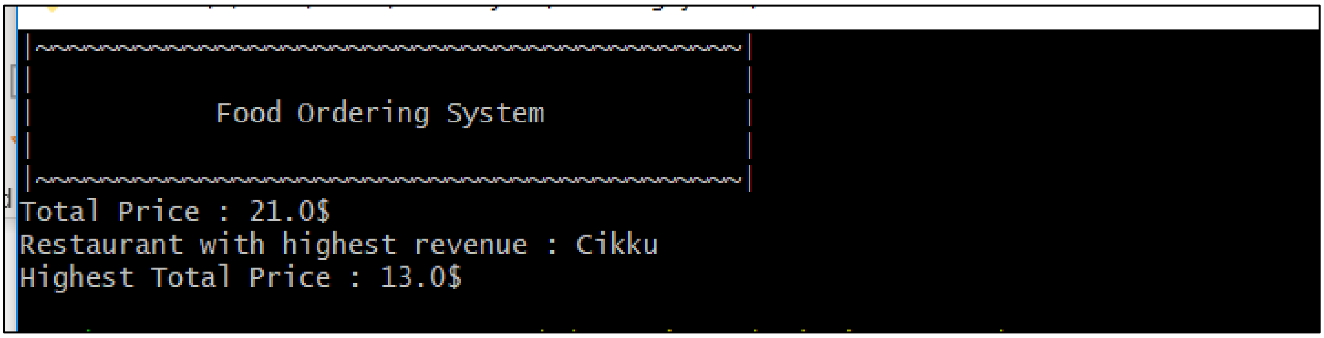
- By running the shell script written:



The shell script:



Successful Result



Both ways compiled and ran the test successfully.

Results:

- The total price is 21.0$
- Restaurant with highest revenue is Cikku
- Highest Total price is 13.0$

References

- Course Notes
- Book: Java in two semesters
- https://www.baeldung.com/java-observer-pattern