Name: Jamie Grech
ID: 123499M
Course: B.Sc. Information Technology (Hons.) (Artificial Intelligence)

Faculty of Information and Communication Technology
Department of Computer Science

L-Università ta' Malta

Study-unit: Object Oriented Programming
Code: CPS2004

Contents

Task 1: C++

Testing

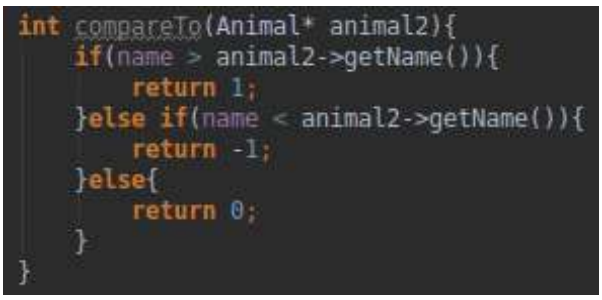
To test the program, the ‘oop-cpp-example.txt’ file provided with the assignment specification was used. In the screenshot below the name of the file was entered incorrectly as ‘oop-cpp.example.txt’ instead of ‘oop-cpp-example.txt’, and the program ran anyway using oop-cpp-example.txt as the default file. It can be seen that the program first adds the 5 animals, then finds and displays the viper node, deletes the eagle node and prints the tree using inorder traversal at the end. As visible in the screenshot, the inorder traversal in order of names is done successfully and the eagle node is not displayed since it was deleted in the last command.

```
jamie@jamie-Aspire-E5-571:~/CLionProjects/OOPCPP$ ./compile.sh
.jamie@jamie-Aspire-E5-571:~/CLionProjects/OOPCPP$ ./run.sh
Please input address of commands file:
oop-cpp.example.txt
Invalid file address, using default file instead
Node added as head
Node added
Node added
Node added
Node added
Node found
Name: viper
Length: 200
Type: Reptile
Venomous: 1
Node deleted
Name: cat
Length: 60
Type: Mammal
Average Litter Size: 4
Name: chameleon
Length: 12
Type: Reptile
Venomous: 0
Name: ostrich
Length: 150
Type: Bird
Can Fly: 0
Name: viper
Length: 200
Type: Reptile
Venomous: 1
jamie@jamie-Aspire-E5-571:~/CLionProjects/OOPCPP$
```

Header Files

animals.h

The header file animals.h contains the superclass Animal as well as its 3 subclasses Mammal, Reptile and Bird. The Animal class has private properties 'name' and 'type' of type string and 'length' of type int, along with their respective getters and setters. It also contains a virtual method printDetails to display these properties which is overridden in the subclasses so as to display their inherited properties as well as their unique properties. Finally, it has a method 'compareTo' which allows objects of type Animal to be compared by comparing their names. The method's return values are as follows.



This gives the class comparable functionality as required to be used in the generic binary search tree implementation. The subclasses simply contain the unique properties 'avgLitterSize' of type int, 'venomous' of type bool and 'flying' of type bool for Mammal, Reptile and Bird respectively, as well as their getters and setters. Moreover, the subclasses also contain more detailed constructors and printDetails methods to accommodate for these new properties.

bst.h

The bst.h header file contains a template for a class Node which has two Node pointers for its left and right child and a variable to store the data of the node, which is of type 'T' representing the argument type passed on declaration of the object, allowing for a generic binary search tree. Moreover, the template class contains getters and setters for the aforementioned properties. Aside from this class, the header file contains methods getSmallestNode, findNode, insertNode, deleteNode, inOrder, preOrder and postOrder. getSmallestNode returns the leftmost node in a given node's subtree. findNode takes an object of the same type as the node passed as a parameter and looks through the tree to find a matching node, returning the data of that node if it is found, and returning null otherwise. insertNode takes the same parameters as findNode but instead inserts it in it's appropriate position in the given node's tree, unless the data being entered is not unique. deleteNode searches for a node in a tree and deletes it if it is found. If the node to be deleted has two children, it uses the getSmallestNode method to find it's inorder successor and replace it with it so as to not throw the tree out of order. The inOrder, preOrder and postOrder recursively traverse through the tree using inOrder traversal, preOrder traversal and postOrder traversal respectively.

The Main Method

The main method prompts the user to insert the file address and attempts to open it using an ifstream. If the inputted file is not found the default file 'oop-cpp-example.txt' is used. If the default file is not found an error message is displayed and the program terminates. Otherwise, the program iterates through the file using the getLine method and a while loop, and splits every line into tokens using an istringstream and an istream_iterator, storing these tokens in a string vector. The first token of every line is checked to determine which command to execute. After all the commands are executed, the tree is displayed using inorder traversal.

Insert

If the first token of the line is 'Insert', the program first checks that there is a total of 5 tokens in the line since the insert command takes 4 parameters. It then checks whether the second token is 'mammal', 'reptile' or 'bird' and displays an error if otherwise. If the second token is one of the aforementioned values, a temporary animal is initialized with properties according to the last 3 tokens. In the case of a Mammal, the 4^{th} and 5^{th} tokens, representing length and avgLitterSize respectively, are converted to integers using the stoi method. In the case of Reptiles, the token representing length is again converted to an int using stoi, while the token representing 'venomous' is converted to a bool using the == operator (tokens[4] == "venomous"). The procedure for the bird method is the same as that for reptile, but comparing the last token with "can-fly" instead of with "venomous".

Find

If the first token of a line is 'Find', the program checks that there are two tokens in total in the line. It then then searches the tree using the findNode method for an animal with the name being the second token in the line. If such a node is found, it's details are printed. Otherwise, "Node not found" is printed.

Remove

If the first token is 'Remove', the program checks that there are two tokens in the line. It then attempts to delete an animal from the tree with name equal to the second token in the line using the deleteNode method.

OOP Principles

Inheritance and Polymorphism

The class hierarchy of the classes in the animals.h file makes use of inheritance with the subclasses Mammal, Reptile and Bird inheriting from the superclass Animal, preventing redundant code in the form of 4 classes with common properties between them being rewritten for each class. This inheritance then allows us to make use of polymorphism in the main class by declaring our binary search tree to be of type Animal thus allowing its nodes to take the form of any of its subclasses.

Comparable interface

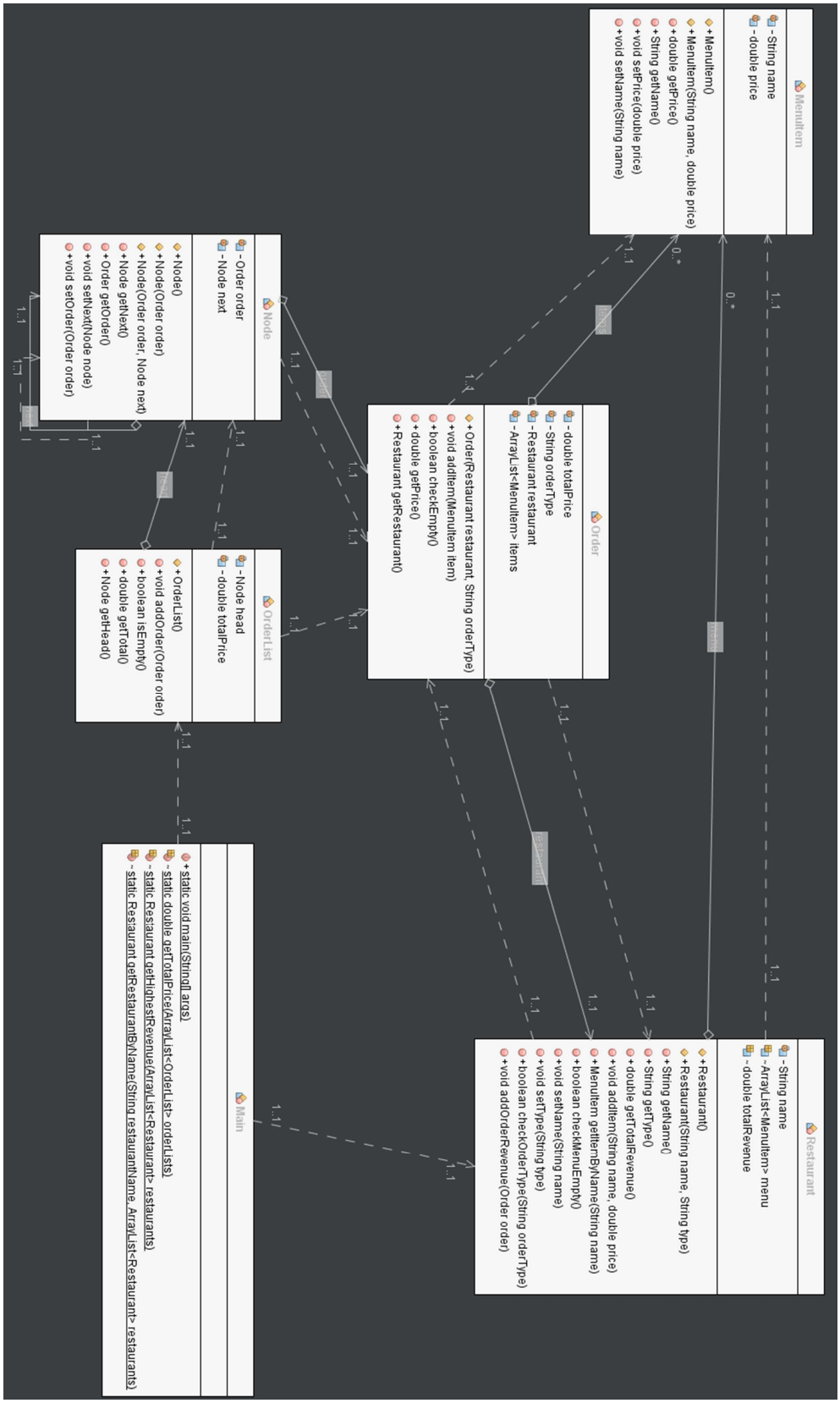
By giving our animal class a compareTo method, we can directly compare an object of type animal or one of its subtypes to another, even if the other object is of a different subtype. This method returns a 1 to represent greater than, 0 to represent equal to and -1 to represent less than, allowing us to sort these objects in an ordered data structure such as a binary search tree.

Generic implementation

By declaring the Node class and all its respective methods as a template with takes <class T> as an argument, the binary search tree implementation is a generic one which can be used to store different object types, as well as the object's subclasses. Despite being somewhat generic, this implementation is not without its limitations. For this implementation, the class being used must have a method specifically called compareTo which takes a pointer to another object of the same type as a parameter, which directly compares the two objects and returns an appropriate value as mentioned in the previous subheading (Comparable interface).

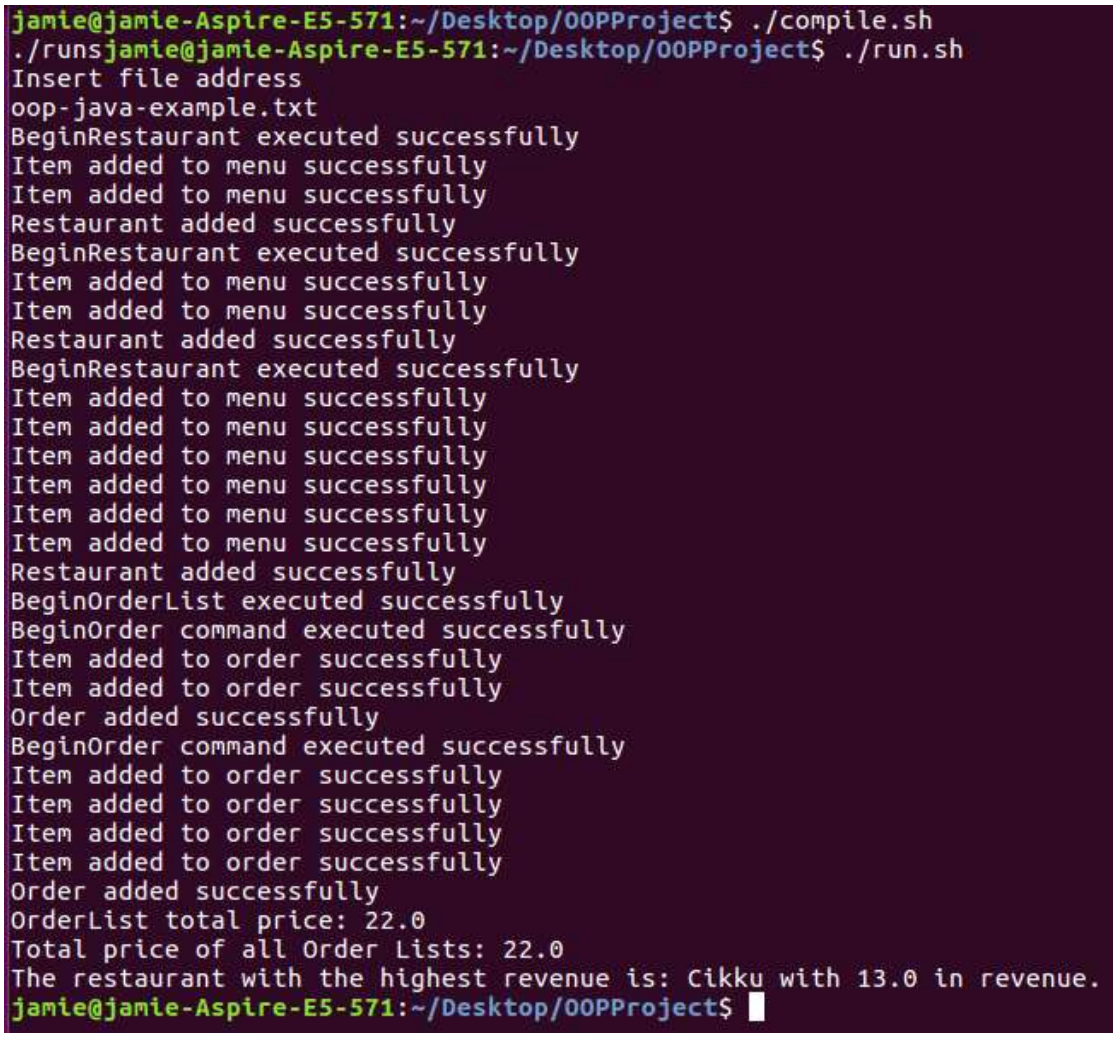
Task 2: Java

Class Diagram



Testing

For testing the core functionality of the program, the file ‘oop-java-example.txt’ provided on the VLE was used, and the program was executed successfully all the way through including the printing of the total order list price at the end of the order list, as well as the total price of all order lists (which was only one list in the case of the test file). Finally, the program prints the name of the restaurant with the highest revenue and its respective revenue amount as requested. In the case that an invalid file address is inserted, the aforementioned file ‘oop-java-example.txt’ will be used. The below screenshot shows the program executing the file.



Class Breakdown

In this section I will explain the contents of each class except the main class, listing the properties of every class and its methods, with the exception of constructors, getters and setters since they are basically the same for every class.

Restaurant

The restaurant class consists of two strings for the name of the restaurant and the type (take-away, delivery or both), a double for the restaurant's total revenue and an ArrayList of type MenuItem to store the items on the restaurant's menu. It contains a method 'getItemByName' which takes a string as a parameter and checks the restaurant's menu for a name matching the string and returning the respective item if it exists. It also contains an 'addItem' method which takes a string for the name and double for the price of an item to be added to the menu. The method first checks if an item with that name is already on the restaurant's menu, and adds the item to the menu if that is not the case. It also contains a method to check if the menu is empty in order to prevent adding restaurants to the system with empty menus, as well as a method which checks if a given order type is accepted by the restaurant. Finally, it contains an update method which takes the price of an item as a double as a parameter which increments the total revenue of the restaurant when it is notified by the notifyRestaurant method of an order placed with that restaurant.

MenuItem

The class menuItem is used to represent items on a restaurant's menu or in an order, and consists simply of a string for the item name and a double for its price, along with a default constructor, parameterized constructor and respective getters and setters.

Order

The Order class contains a double for the total price of the order, a string for the order type i.e. take-away or delivery, a Restaurant which is the restaurant the order is placed with and an ArrayList of type MenuItem containing the items in the order. It has a method to check if the list of items is empty to prevent empty orders being submitted, as well as a method to add an item to the order. This method takes a menu item passed as a parameter and adds it to the order, as well as incrementing the total price of the order by the price of the respective item. The item is fetched by name in the main method by using the getItembyName method from the restaurant associated with the order. If there is no such item on the restaurant's menu, the aforementioned method returns null, in which case the function does nothing as it first checks if the item passed as a parameter is null. The addItem method also calls the notifyRestaurant method with the price of the item as a parameter so as to call the restaurant's update method.

Node

The Node class represents one node within the linked list implementation, and thus used further in the OrderList class. It contains two properties, an Order which is the order stored in that particular node, and another Node, to point to the next node in the linked list. The most important constructor in the class is a constructor which takes one parameter of type order, since when adding a new order to the linked list, the next node does not need to be initialized at first.

OrderList

The OrderList class contains a Node which functions as the head of the list as well as a double to store the total price of all orders in the list. It only has one constructor with no parameters since the BeginOrderList command must be called before adding orders to the list. The addOrder method of this class takes an order as a parameter and initializes a node with that order as its data. It then checks if the head of the list is empty in which case it sets the aforementioned new node as the head, otherwise it runs through the list till it finds the last item in the list, i.e. the item with its next node being null, and adds the node to the end of the list. Finally, the class also contains a simple method to check whether the list is empty by simply checking whether the head of the list is null.

The Main Method

The main method of the program first prompts the user to input the address of the text file containing the commands to be executed by the program. It then attempts to open the text file using BufferedReader within a try catch block to catch FileNotFound exceptions if an invalid file address is entered. If the exception is caught, a Boolean fileFound is set to false which triggers the program to use the default file instead. The initialization of the BufferedReader with the default file is also in a try catch block in the case that the user deletes the default file, which would cause a FileNotFoundException. If this exception is caught the main method returns thus causing the program to terminate since there is no valid file to work with.

```
boolean fileFound;
try {
    file = new File(address);
    reader = new BufferedReader(new FileReader(file));
    fileFound = true;
} catch (FileNotFoundException fnfe) {
    System.out.println("File address not found, default file will be used instead");
    fileFound = false;
}
if(!fileFound){
    try{
        file = new File("oop-java-example.txt");
        reader = new BufferedReader(new FileReader(file));
    }catch(FileNotFoundException fnfe){
        System.out.println("Default file not found, possibly renamed, moved or deleted");
        return;
    }
}
```

After the file is loaded, the program goes through the text file line by line, skipping over empty lines. It uses StringTokenizer to break the commands into tokens using space as the delimiter. The program then uses a switch case ladder using the first token of every line, which should be the command, as the switch variable. This switch case ladder is also enclosed in a try catch block to catch exceptions when reading data from the file

BeginRestaurant

For the BeginRestaurant command, the program checks that 'tempRestaurant' is null to make sure that CloseRestaurant was used after the last BeginRestaurant or that this is the first BeginRestaurant command being used. It then checks that there are two tokens after the commands since it takes two parameters being the name and type. Finally, it checks that the 2nd parameter is either 'delivery', 'take-away' or 'both' and that a restaurant with that name does not already exist. If all these checks are successfully completed, a temporary restaurant with these properties and an empty menu is created.

Item

This command is used to add an item to the current restaurant's menu after initializing it with BeginRestaurant. It checks that the current restaurant is not null and that the command has two parameters after it. It then formats the second parameter two a double with two decimal places since it is the price. If this formatting is not successful, the price will be set to negative 1, thus making the item

fail the final check for a positive price. On the other hand, if this check passes, the item is added to the current restaurant's menu. It does not need to check that the item name is unique in the restaurant's menu as this check is done in the Restaurant's addItem method.

EndRestaurant

This command checks that the current restaurant is not null and that its menu is not empty. If both these checks are successful, it adds the restaurant to the ArrayList of restaurants and sets the current restaurant to null.

BeginOrderList

This command simply checks that there is no OrderList currently open by checking that tempOrderList is null, and initializes a new OrderList if that is the case.

BeginOrder

This command checks that the current order is null and that there are two parameters after the command. It then checks that the second parameter is either takeaway or delivery, followed by whether a restaurant with the name in the first parameter exists. Finally, it checks if the requested order type is accepted by that restaurant. If all these checks are successful an order with that restaurant of that type is initialized with an empty list of items.

OrderItem

This command checks that an order has been initialized and that there is one parameter after the command, being the item name. If these checks are successful it checks the current order's restaurant's menu for an item with that name. If this item is found it is finally added to the current order.

EndOrder

This command checks that the current order is not null and not empty. It then checks that the current order list is also not null because all orders need to be added to an order list. If these checks are successful, the current order is added to the current orderList. Finally, the current order is set to null.

EndOrderList

This command checks that the current order list is not null and not empty. If these checks are successful, the OrderList is added to the list of OrderLists and the total price of the OrderList is outputted. Finally, the current OrderList is then set to null.

End of File

At the end of the file a method getTotalPrice is called to get the total price of all order lists combined. A method getHighestRevenue is then called to find the restaurant with the highest individual revenue. Finally, the name and total revenue of this restaurant are outputted and the program then terminates.

OOP Principles

Observer Design Pattern

The observer design pattern is implemented to update the revenue of restaurants with items being added to orders with the particular restaurant. When an item is added to an order, the notifyRestaurant method is called which calls the restaurant's update method, passing the price of the added item as a parameter. This update method then updates the total revenue of that restaurant. The restaurant does not need to be registered as an observer since every order must already be linked to a particular restaurant.