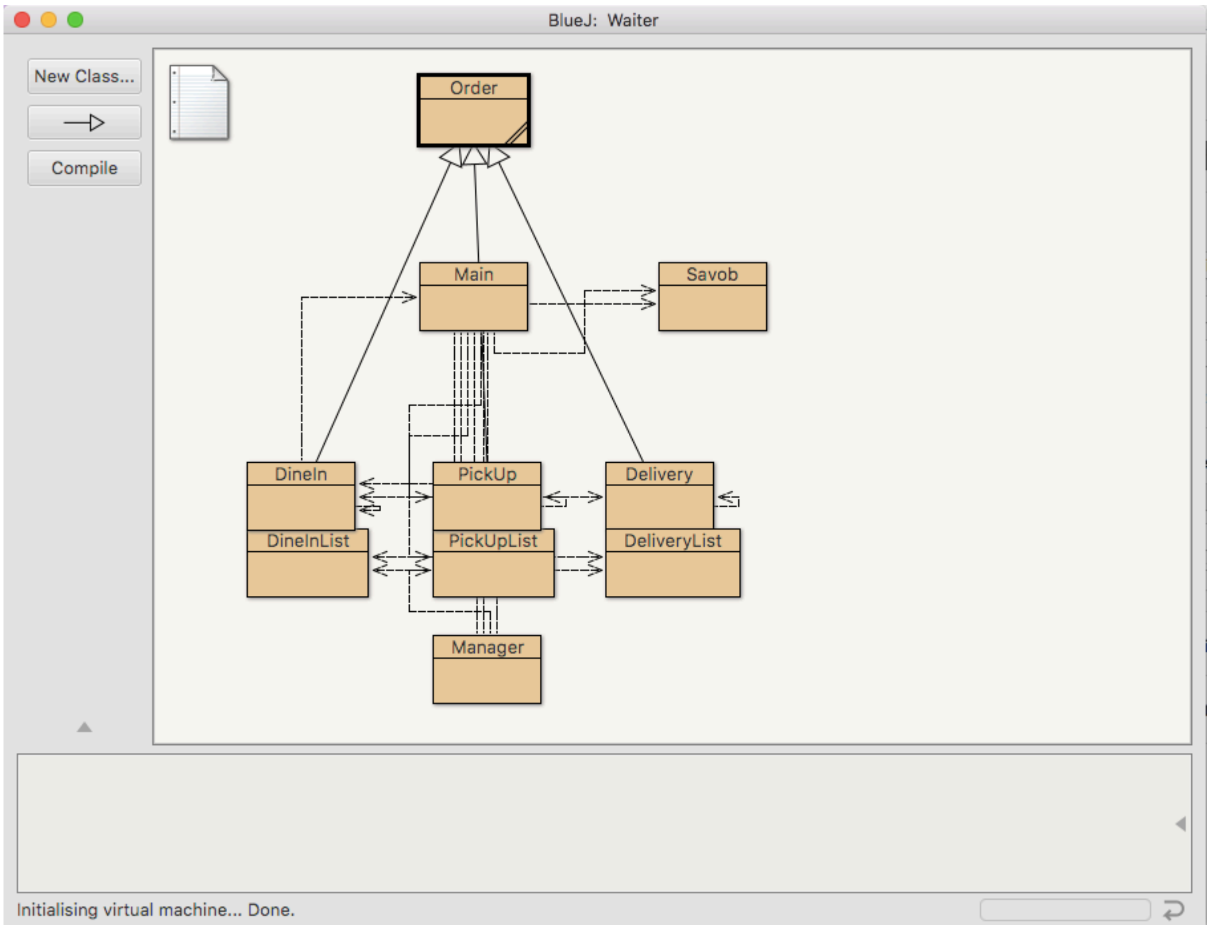
Criterion C: Development

This program was written in Java with the help of the BlueJ IDE. The program is highly object-oriented, focusing on dynamic data structures for modular programming.

Program Functions



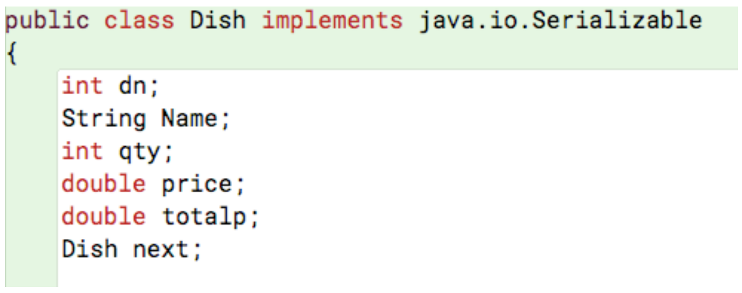
Initialising virtual machine... Done.

The techniques implemented in this program are :

- Self-Referential Classes
- Singly Linked List
- Dynamic Queue
- Serialization of objects
- Text Files
- Inheritance
- Encapsulation
- Switch Cases for Menus

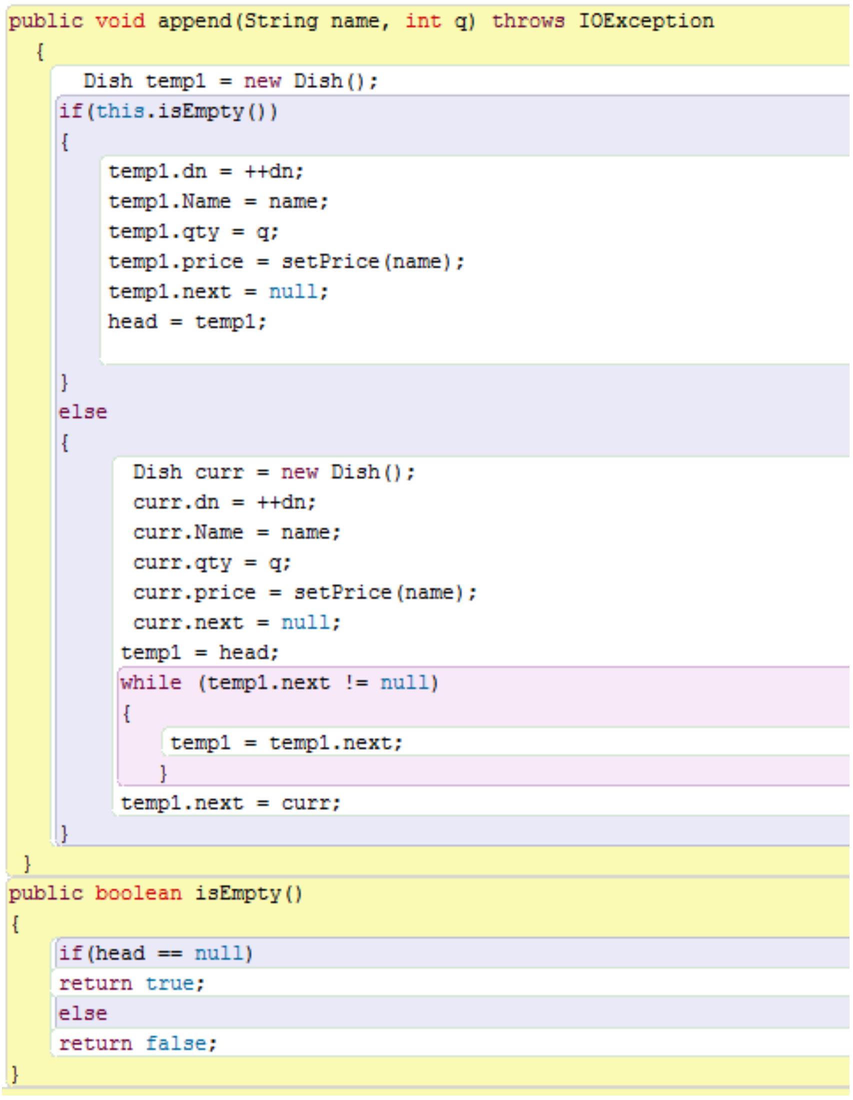
- Iterative Structures
- Core libraries
- User defined methods
- Error Handling & Validation Techniques

Self-Referential Classes : In order to implement the various Abstract Data Types into my program, I made use of Self-Referential classes which will allow for the implementation of dynamic ADTs. I have implemented this with the use of a “next” variable in all of my node classes. It has been implemented as such :



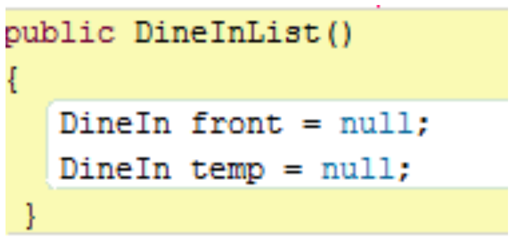
```
public class Dish implements java.io.Serializable
{
    int dn;
    String Name;
    int qty;
    double price;
    double totalp;
    Dish next;
```

Singly Linked List : I implemented the singly linked list for the order class, where it is used to list the dishes in a customer’s order. This class has a menu and encapsulated functions that allow the user to operate on the nodes of the list. Once a waiter enters the dish name, the price is calculated and it is added to the list, so that when required, the bill can be printed on the screen. Using a linked list simplifi ed the code, as I encapsulated each node into its own “Dish” object, with its own object variables and methods. Due to this abstraction, manipulating the dishes in a customer’s order became much more doable.

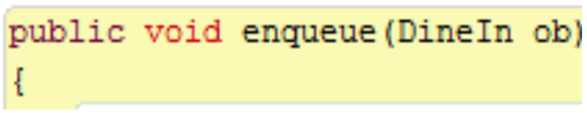


```
public void append(String name, int q) throws IOException
  {
     Dish temp1 = new Dish();
    if(this.isEmpty())
    {
        temp1.dn = ++dn;
        temp1.Name = name;
        temp1.qty = q;
        temp1.price = setPrice(name);
        temp1.next = null;
        head = temp1;

    }
    else
    {
          Dish curr = new Dish();
          curr.dn = ++dn;
          curr.Name = name;
          curr.qty = q;
          curr.price = setPrice(name);
          curr.next = null;
         temp1 = head;
         while (temp1.next != null)
         {
            temp1 = temp1.next;
           }
         temp1.next = curr;
    }
 }
public boolean isEmpty()
{
    if(head == null)
    return true;
    else
    return false;
}
```

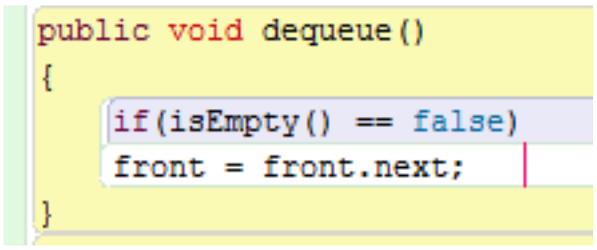
Dynamic Queues : Dynamic Queues turned out to be quite useful in my DineInList, PickUpList, and DeliveryList classes. These classes contained dynamic queues of Order objects, so that each order could be resolved on a FIFO basis, exactly like a restaurant manages customer orders. Due to the implementation of a queue, I can ensure that customers have to “wait in line” to receive their order, and saving this into a file also becomes much more efficient.



```
public DineInList()
{
    DineIn front = null;
    DineIn temp = null;
 }
```

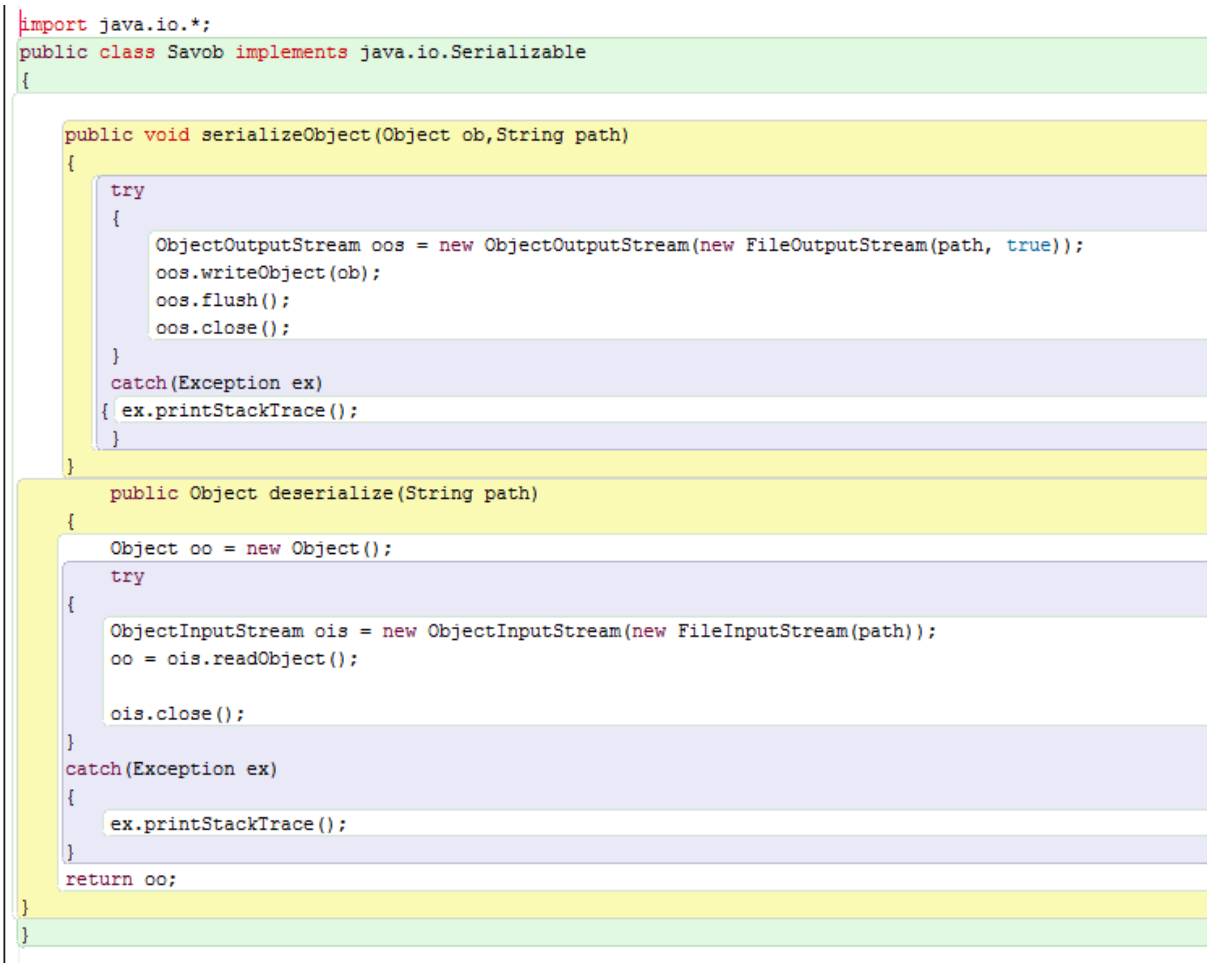


```
public void enqueue(DineIn ob)
{
```



```
public void dequeue()
{
      if(isEmpty() == false)
      front = front.next;
}
```

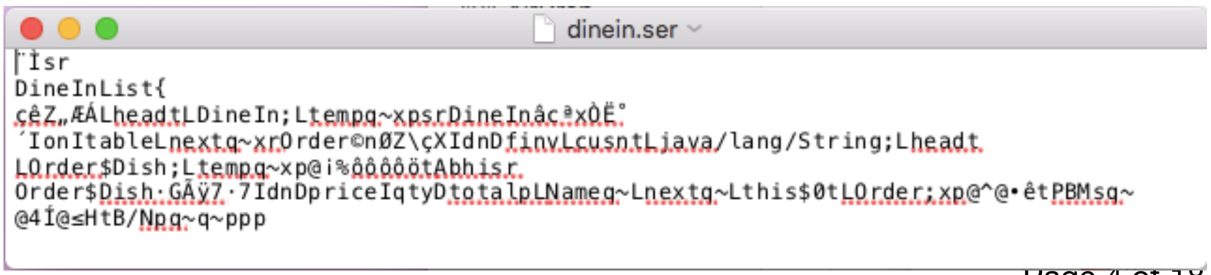
Serialisation of Objects : In order to create a permanent storage solution for my product, I decided to implement serialization, which in essence, involves converting objects into bytecode which is stored in a special file called a serialized file(with the .ser extension). I opted to use this instead of random access files, as these files are much more "permanent", and I am enabled to store and retrieve data in the form of objects without having to instantiate new objects every time the program executes ("Serialisation and deserialisation of Objects",GeeksforGeeks).



```
import java.io.*;
public class Savob implements java.io.Serializable
{

    public void serializeObject(Object ob,String path)
    {
        try
        {
            ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(path, true));
            oos.writeObject(ob);
            oos.flush();
            oos.close();
        }
        catch(Exception ex)
       { ex.printStackTrace();
        }
    }
        public Object deserialize(String path)
    {
        Object oo = new Object();
        try
    {
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(path));
        oo = ois.readObject();

        ois.close();
    }
    catch(Exception ex)
    {
        ex.printStackTrace();
    }
    return oo;
}
}
```

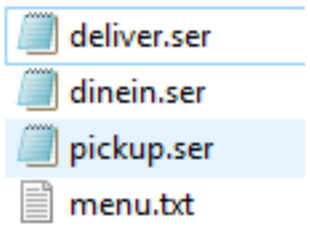
The biggest problem I encountered with creating my infrastructure was that I could only save a single object into a serialized file, which is why I needed to work around that and collate every order together. This was done through the use of dynamic queues which were separate for Dine In, Pickup and Delivery. Once there was a queue of these implemented, it was quite straightforward to save them into a serialised file, and retrieve them when the need arises.

The serialised files, when populated with data, look as such :



```
|¨Ísr
DineInList{
çêZ„ÆÁLheadtLDineIn;Ltempq~xpsrDineInâcªxÒË˚
´IonItableLnextq~xrOrder©nØZ\çXIdnDfinvLcusntLjava/lang/String;Lheadt
LOrder$Dish;Ltempq~xp@i%ôôôôötAbhisr
Order$Dish·GÃÿ7·7IdnDpriceIqtyDtotalpLNameq~Lnextq~Lthis$0tLOrder;xp@^@•êtPBMsq~
@4Í@≤HtB/Npq~q~ppp
```

All of the lists that are to be serialised are stored in separate serialised files as such :



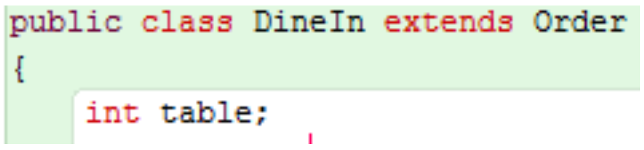
deliver.ser
dinein.ser
pickup.ser
menu.txt

This allows for these lists and their data to be accessed much more effectively throughout the program.

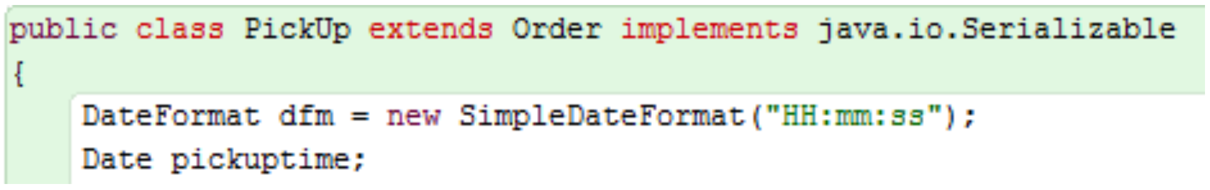
Text Files : I implemented a Scanner class that could read data off of a text file, where I stored the details of every item on the restaurant's menu, such as the name of the dish and the price for a single dish. By using a text file to conduct this task, the user would not be required to enter the price of every item he adds to the order, and instead simply has to enter the name of the dish which is then cross-referenced with the text file database and the total price of that dish is calculated and bound to the Dish object. **(Refer to the "Set Price" function for further details)**

Inheritance : Through the use of inheritance, I simplified the redundancies in coding, as most of the classes inherit their variables and methods from the Order class, after which they add their own special variables, such as the addition of a pickup time for pick-up orders, or an address for delivery orders.

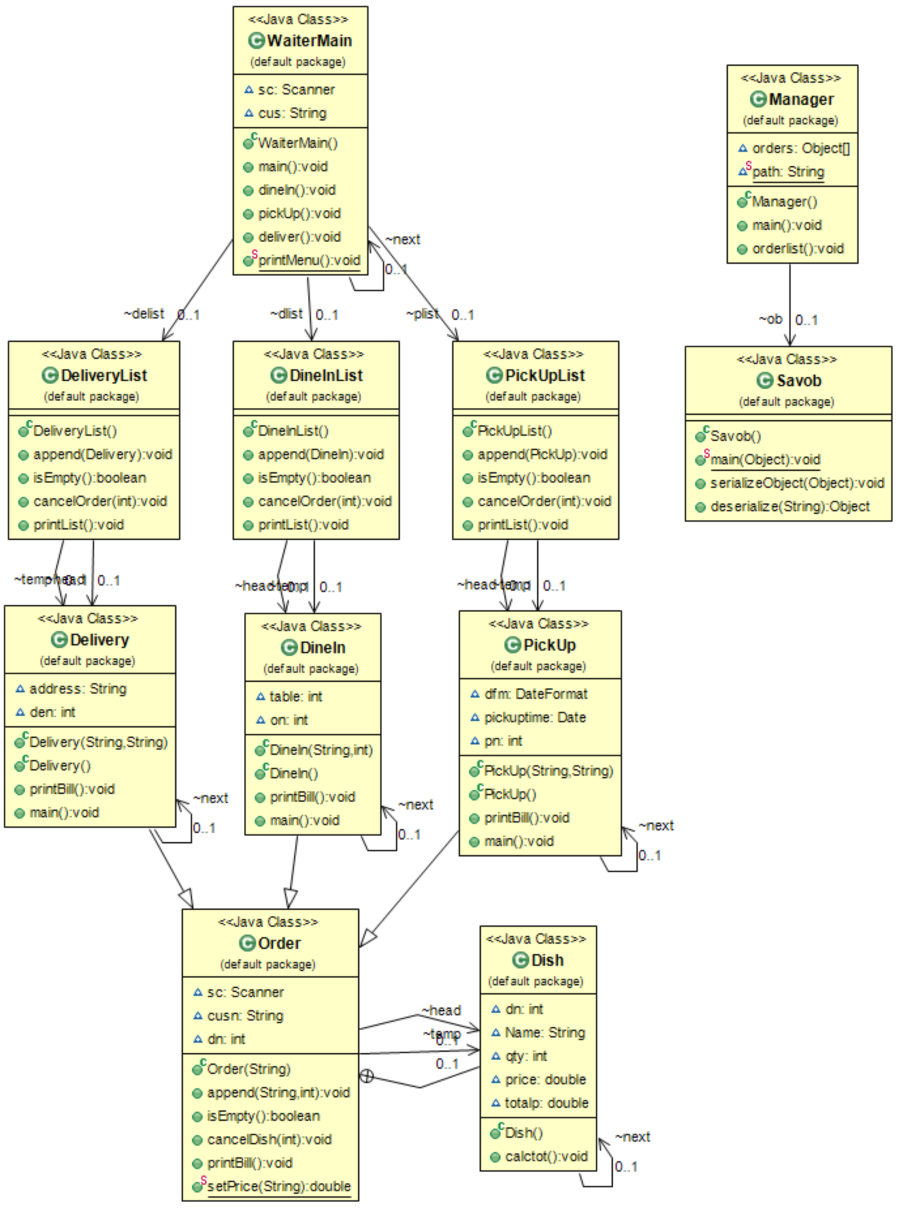
In the DineIn, PickUp, and Delivery class, the constructor from the parent class "Order" is inherited, and extra variable are added to make the orders more categorical. This includes additions such as a "time" variable for the PickUp class, and a table number in the DineIn class. Below are the corresponding snippets of code :



```
public class DineIn extends Order
{
    int table;
```



```
public class PickUp extends Order implements java.io.Serializable
{
    DateFormat dfm = new SimpleDateFormat("HH:mm:ss");
    Date pickuptime;
```



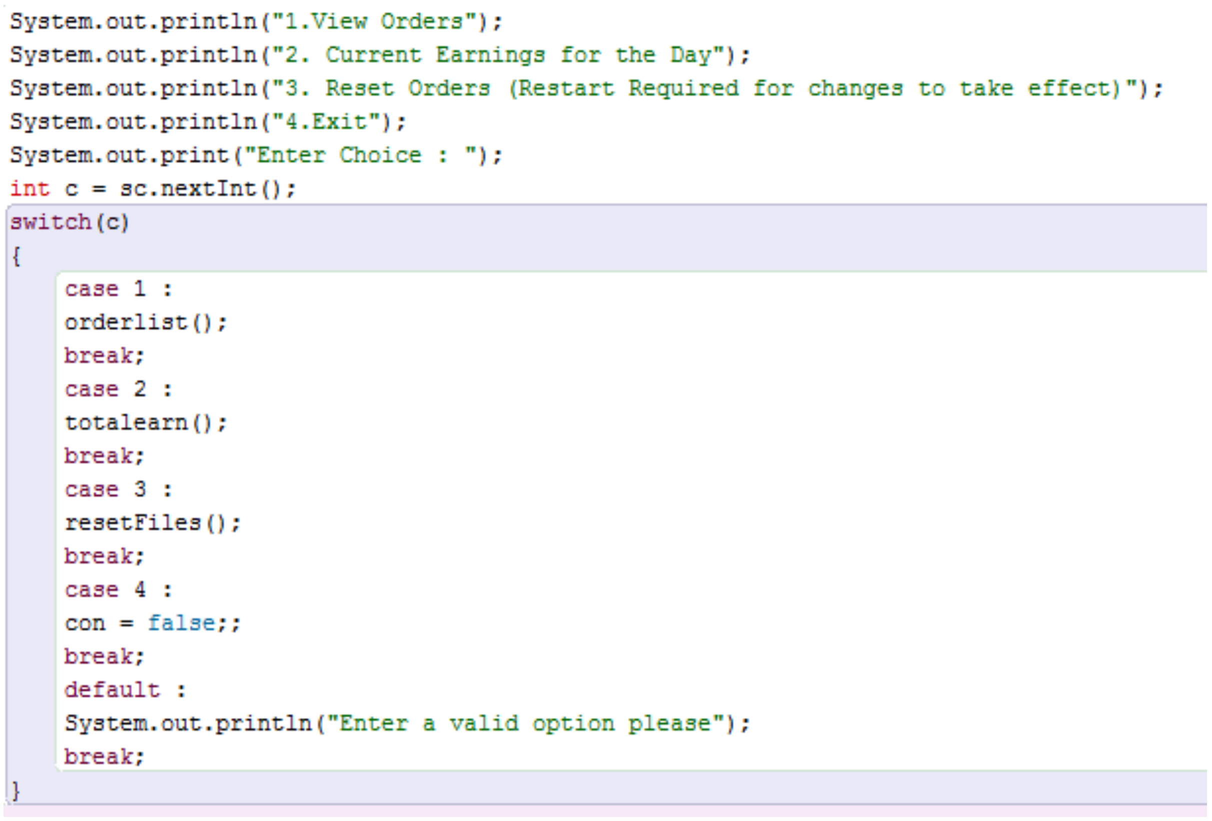
Encapsulation : Through the use and implementation of various access specifiers, I ensured that variables could only be accessed by the appropriate classes.

```
private Dish head = new Dish();
public double finv = 0.0;
private Dish temp = new Dish();
String cusn = "";
private int dn = 0;
```

The “transient” specifier is one that does not serialise the specified variable, as serialising a Scanner object for each instance would simply waste memory.

```
transient Scanner sc = new Scanner(System.in);
```

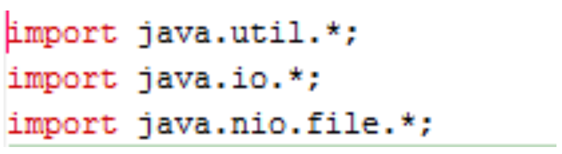
Switch Cases : The usage of switch cases in my program allowed me to compensate for the lack of a Graphical User Interface by allowing a much more clear and comprehensive interaction between my program and the user. The menus I implement are quite standard in terms of format, requiring users to enter a number that corresponds to a certain task, which will take them to the next menu, or perform the appropriate tasks. Below is an example of a switch case that I have implemented for the functioning of a menu in my product.



```
System.out.println("1.View Orders");
System.out.println("2. Current Earnings for the Day");
System.out.println("3. Reset Orders (Restart Required for changes to take effect)");
System.out.println("4.Exit");
System.out.print("Enter Choice : ");
int c = sc.nextInt();
switch(c)
{
    case 1 :
    orderlist();
    break;
    case 2 :
    totalearn();
    break;
    case 3 :
    resetFiles();
    break;
    case 4 :
    con = false;;
    break;
    default :
    System.out.println("Enter a valid option please");
    break;
}
```

Iterations : The main usage of iterative structures in my product lies in the implementation of infinite "while" loops, that encompass most of my menus, so that a constant session is created and this session can only be terminated once the user chooses to exit the application. The use of "boolean" variables that check whether the menu needs to be terminated ensure that the program runs only as much as the user wants it to.

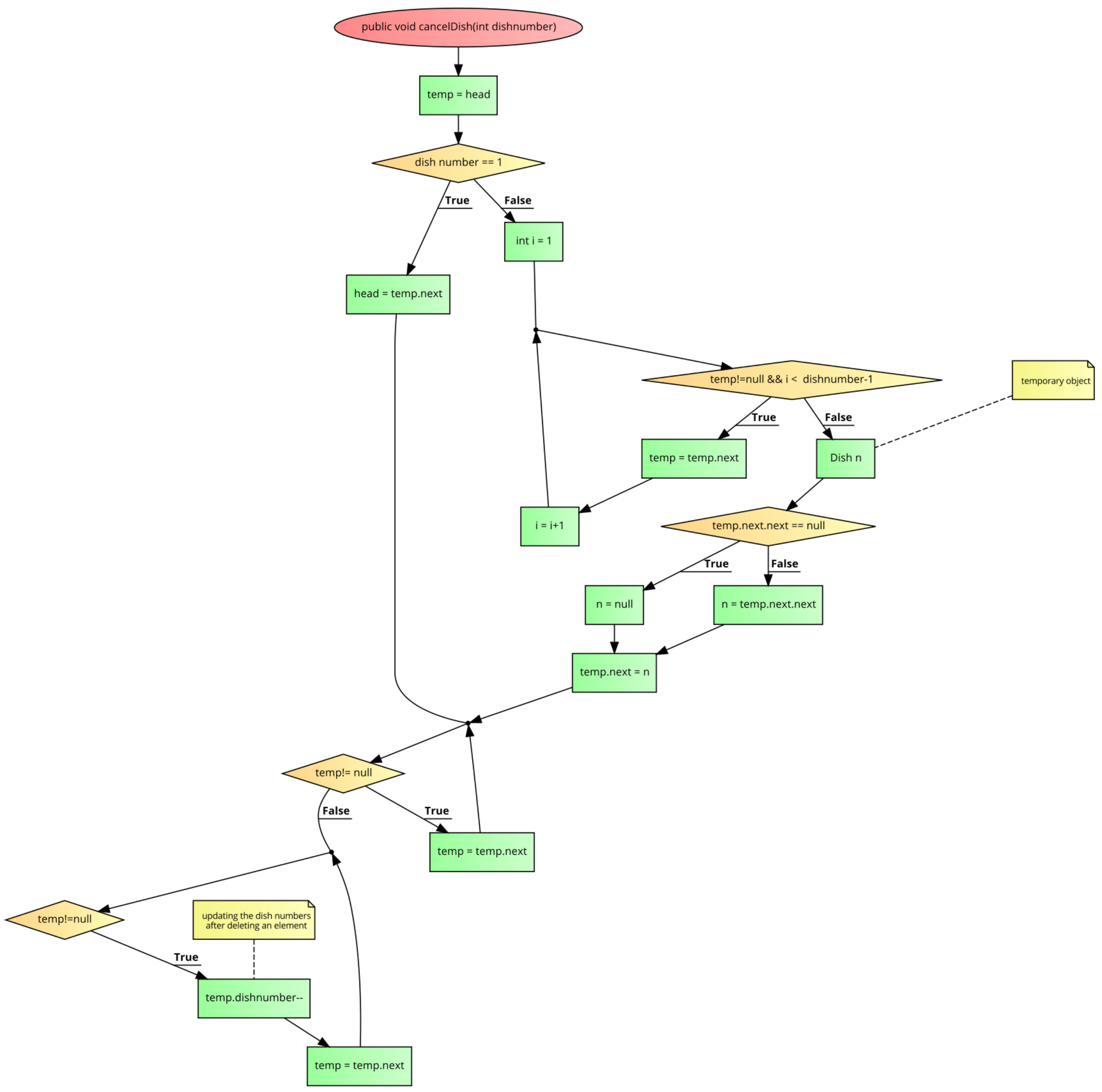
Java Core Libraries : Throughout this product, I have implemented various pure-defined Java methods, such as the Date class in Java's "util" package, or the use of the File class in Java's "io" package. The methods provided in these packages have simplified various processes in my product.



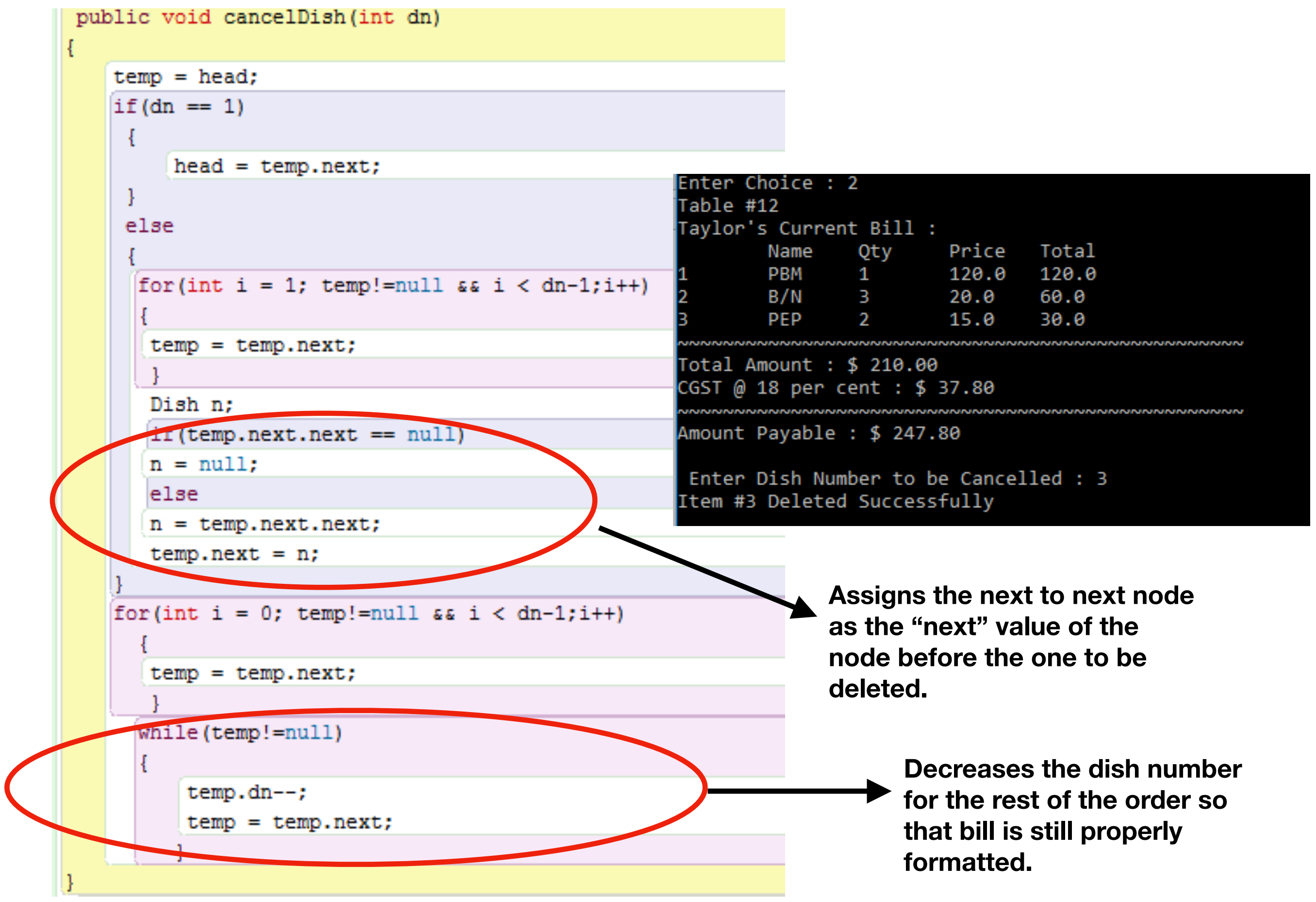
```
import java.util.*;
import java.io.*;
import java.nio.file.*;
```

User-Defined Functions

1. Cancel Dish Method



Above is a flow chart for the cancelDish() method I have implemented in my program. This method accepts an integer index called “dishnumber” that points to the index number of the dish with regards to the current bill. “Dishnumber” starts from 1. Once the number is entered, the program will traverse the list and delete the item from the current bill. The code for this method is as follows :



```
public void cancelDish(int dn)
{
    temp = head;
    if(dn == 1)
     {
         head = temp.next;
     }
     else
     {
      for(int i = 1; temp!=null && i < dn-1;i++)
      {
       temp = temp.next;
       }
       Dish n;
       if(temp.next.next == null)
       n = null;
       else
       n = temp.next.next;
       temp.next = n;
    }
    for(int i = 0; temp!=null && i < dn-1;i++)
      {
       temp = temp.next;
       }
      while(temp!=null)
      {
          temp.dn--;
          temp = temp.next;
        }
}
```

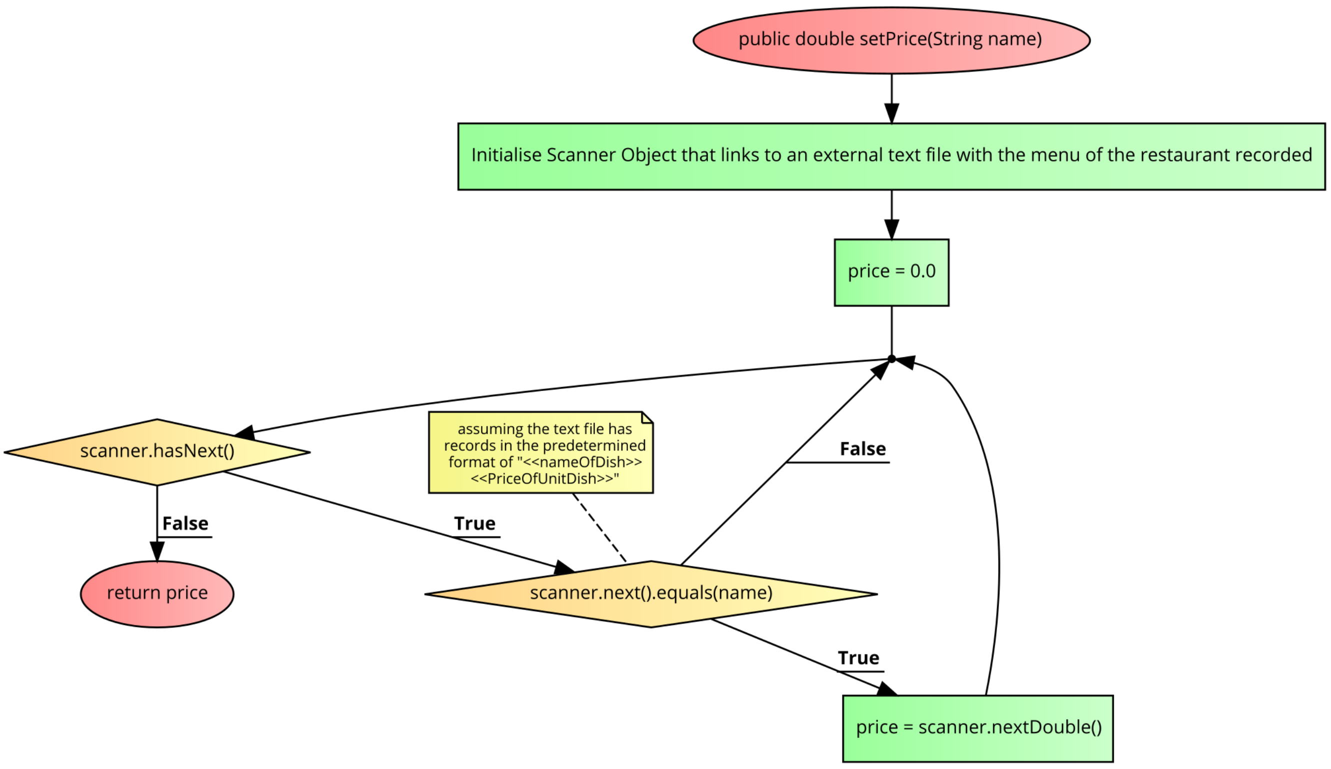
```
Enter Choice : 2
Table #12
Taylor's Current Bill :
        Name    Qty     Price   Total
1       PBM     1       120.0   120.0
2       B/N     3       20.0    60.0
3       PEP     2       15.0    30.0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Total Amount : $ 210.00
CGST @ 18 per cent : $ 37.80
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Amount Payable : $ 247.80

 Enter Dish Number to be Cancelled : 3
Item #3 Deleted Successfully
```

Assigns the next to next node as the “next” value of the node before the one to be deleted.

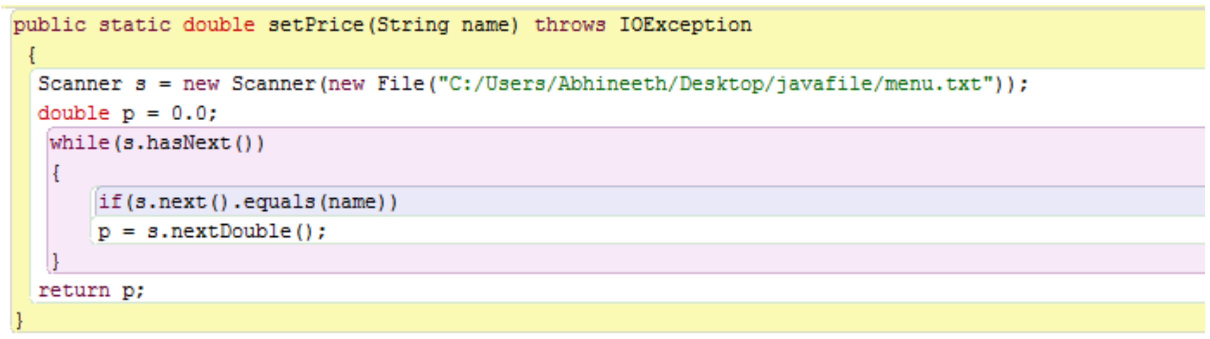
Decreases the dish number for the rest of the order so that bill is still properly formatted.

In the above example, once the code is executed, Item #3 “PEP” would not be on the bill anymore and the customer will not be charged for it.

2. Set Price Method

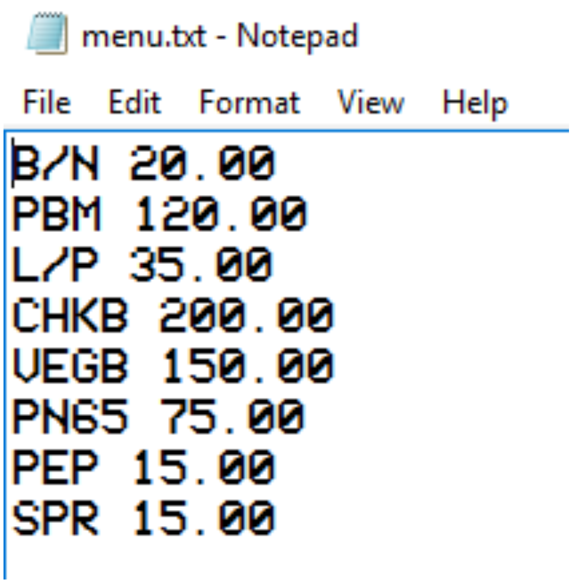

public double setPrice(String name)
Initialise Scanner Object that links to an external text file with the menu of the restaurant recorded
price = 0.0
scanner.hasNext()
assuming the text file has records in the predetermined format of "<<nameOfDish>> <<PriceOfUnitDish>>"
False
False
True
return price
scanner.next().equals(name)
True
price = scanner.nextDouble()

Above is the program flow chart for the setPrice() method in my program. This function enables the automatic setting of the price of a dish for the bill. Once the user enters the code name of the dish(String name), the function searches through the “menu” text file with all of the items and their prices using a Scanner object, and returns the price of the selected dish. This method makes the waiter’s job much easier as they do not need to manually enter the price of every single dish they add to the bill. The code for this function is as follows :



```
public static double setPrice(String name) throws IOException
 {
  Scanner s = new Scanner(new File("C:/Users/Abhineeth/Desktop/javafile/menu.txt"));
  double p = 0.0;
   while(s.hasNext())
   {
       if(s.next().equals(name))
       p = s.nextDouble();
   }
  return p;
}
```

The text file that this function is reading from looks as such :



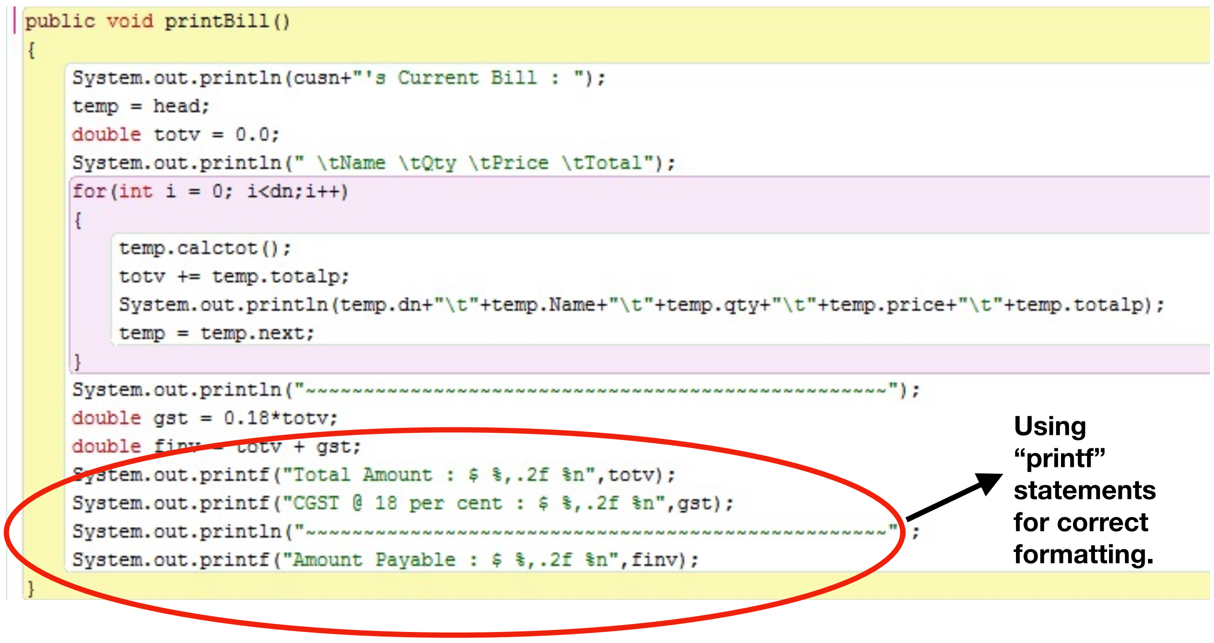
menu.txt - Notepad

```
B/N 20.00
PBM 120.00
L/P 35.00
CHKB 200.00
VEGB 150.00
PN65 75.00
PEP 15.00
SPR 15.00
```

With a single space acting as the delimiter for the scanner class. This menu, however, has to be manually entered by the client.

3. Print Bill method

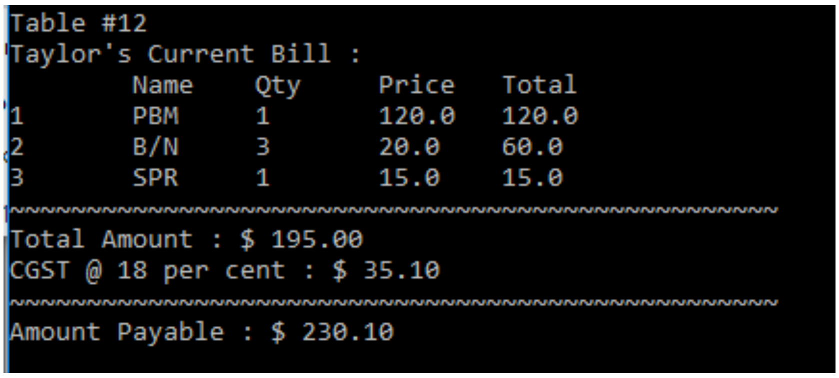
This method relates to the formatting and printing of a bill of the customer's order so far. This bill will also display the tax value and the final price inclusive of tax. The code is as follows :



```
public void printBill()
{
    System.out.println(cusn+"'s Current Bill : ");
    temp = head;
    double totv = 0.0;
    System.out.println(" \tName \tQty \tPrice \tTotal");
    for(int i = 0; i<dn;i++)
    {
        temp.calctot();
        totv += temp.totalp;
        System.out.println(temp.dn+"\t"+temp.Name+"\t"+temp.qty+"\t"+temp.price+"\t"+temp.totalp);
        temp = temp.next;
    }
    System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    double gst = 0.18*totv;
    double finv = totv + gst;
    System.out.printf("Total Amount : $ %,.2f %n",totv);
    System.out.printf("CGST @ 18 per cent : $ %,.2f %n",gst);
    System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    System.out.printf("Amount Payable : $ %,.2f %n",finv);
}
```

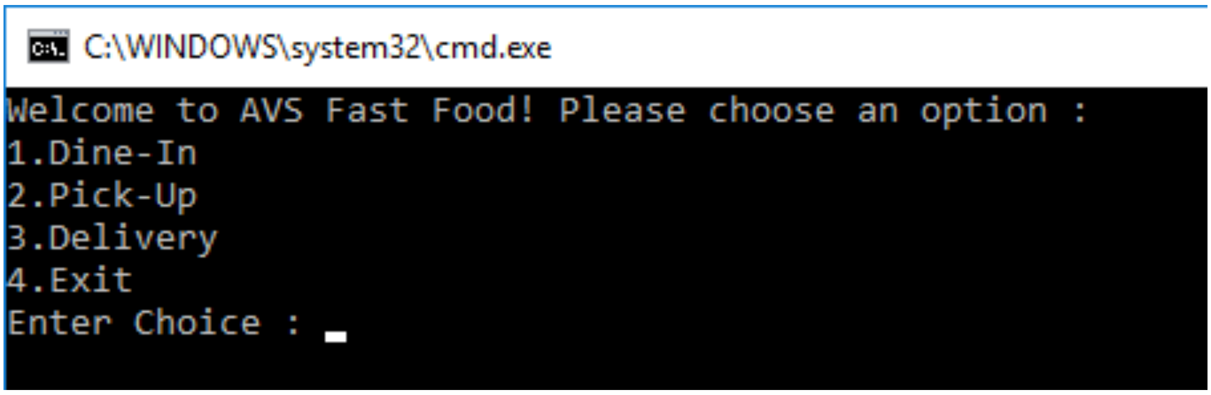
Using "printf" statements for correct formatting.

An example bill is given below :



```
Table #12
Taylor's Current Bill :
        Name    Qty     Price   Total
1       PBM     1       120.0   120.0
2       B/N     3       20.0    60.0
3       SPR     1       15.0    15.0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Total Amount : $ 195.00
CGST @ 18 per cent : $ 35.10
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Amount Payable : $ 230.10
```

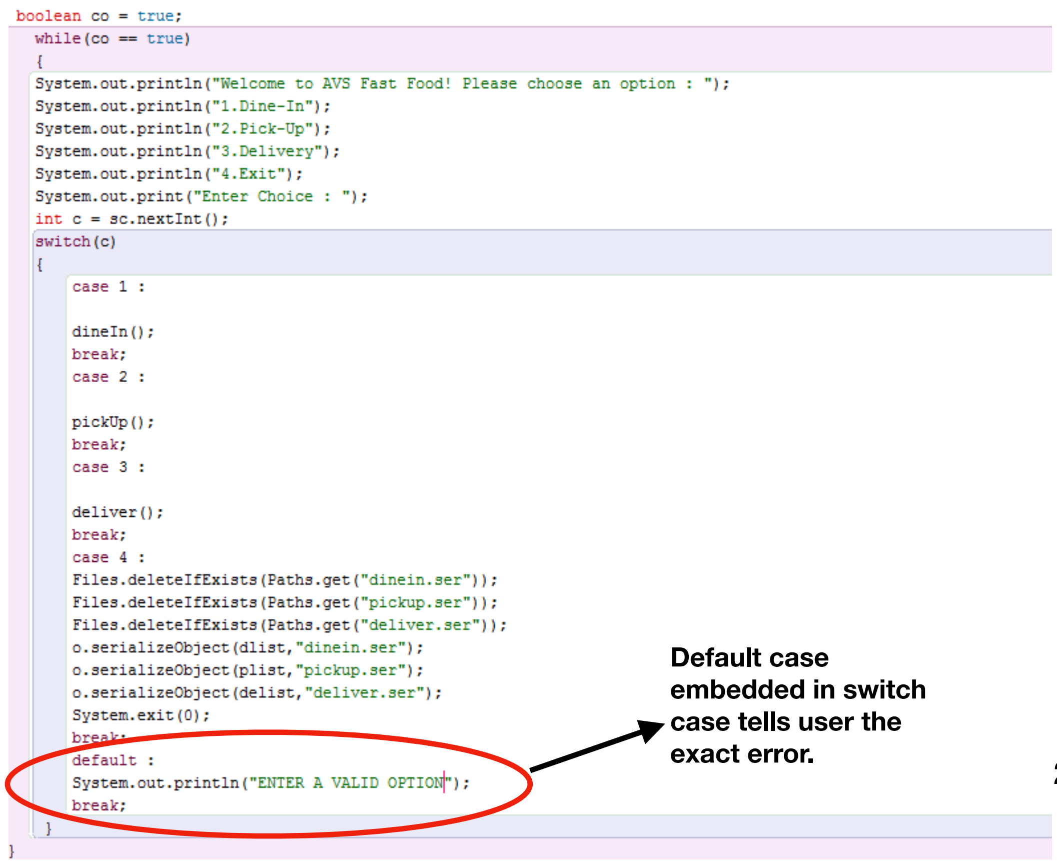
Upon starting up the program, this menu is shown :



C:\WINDOWS\system32\cmd.exe

```
Welcome to AVS Fast Food! Please choose an option :
1.Dine-In
2.Pick-Up
3.Delivery
4.Exit
Enter Choice : _
```

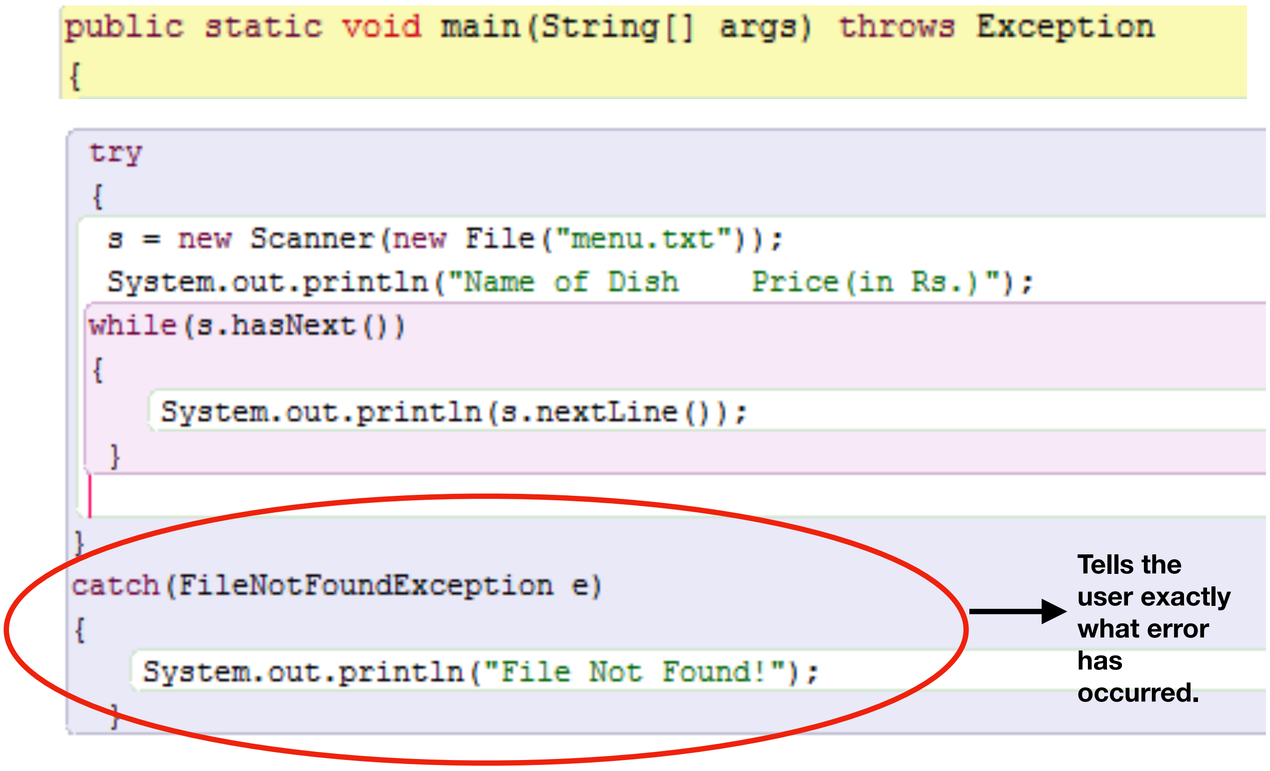
Error Handling & Validation Techniques : In most areas of my program, I have implemented various validation rules for the menus by including a “default” case that executes if an unexpected value is entered, and by enclosing the entire switch case into a while loop that only terminates when the user wants it to.



```java
boolean co = true;
  while(co == true)
  {
  System.out.println("Welcome to AVS Fast Food! Please choose an option : ");
  System.out.println("1.Dine-In");
  System.out.println("2.Pick-Up");
  System.out.println("3.Delivery");
  System.out.println("4.Exit");
  System.out.print("Enter Choice : ");
  int c = sc.nextInt();
  switch(c)
  {
      case 1 :

      dineIn();
      break;
      case 2 :

      pickUp();
      break;
      case 3 :

      deliver();
      break;
      case 4 :
      Files.deleteIfExists(Paths.get("dinein.ser"));
      Files.deleteIfExists(Paths.get("pickup.ser"));
      Files.deleteIfExists(Paths.get("deliver.ser"));
      o.serializeObject(dlist,"dinein.ser");
      o.serializeObject(plist,"pickup.ser");
      o.serializeObject(delist,"deliver.ser");
      System.exit(0);
      break;
      default :
      System.out.println("ENTER A VALID OPTION");
      break;
   }
}
```

Default case embedded in switch case tells user the exact error.

I also use quite a few throw statements and try-catch blocks to ensure that errors are caught and displayed to the user.

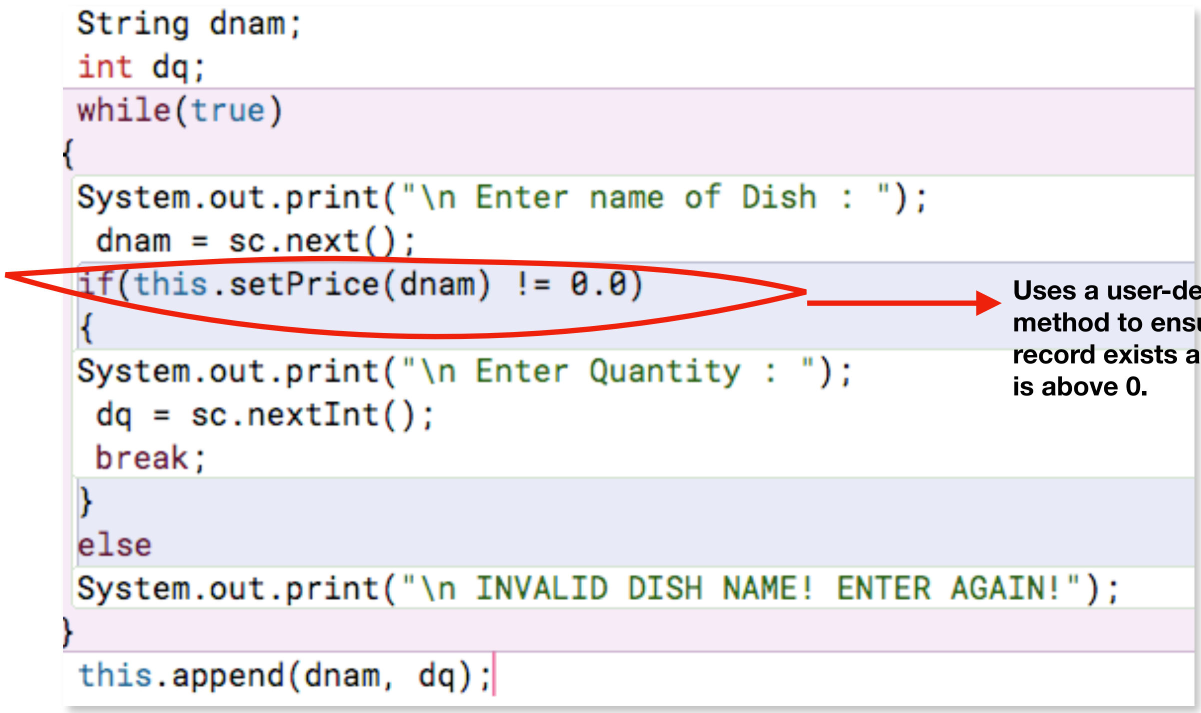


```
public static void main(String[] args) throws Exception
{
```

```
try
{
 s = new Scanner(new File("menu.txt"));
 System.out.println("Name of Dish     Price(in Rs.)");
while(s.hasNext())
{
    System.out.println(s.nextLine());
 }

}
catch(FileNotFoundException e)
{
    System.out.println("File Not Found!");
 }
```

Tells the user exactly what error has occurred.

The screenshot below shows the use of an if-else statement to perform a special validity check in my program

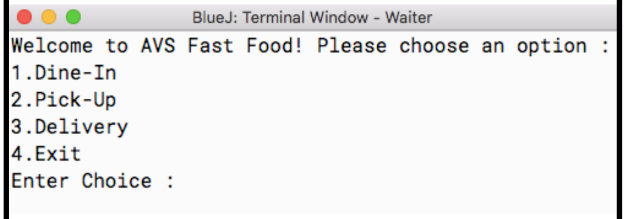


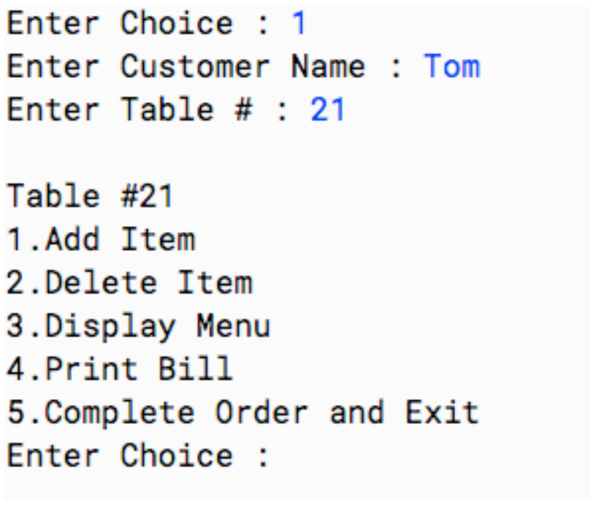
```
String dnam;
int dq;
while(true)
{
System.out.print("\n Enter name of Dish : ");
 dnam = sc.next();
if(this.setPrice(dnam) != 0.0)
{
System.out.print("\n Enter Quantity : ");
 dq = sc.nextInt();
 break;
}
else
System.out.print("\n INVALID DISH NAME! ENTER AGAIN!");
}
this.append(dnam, dq);
```

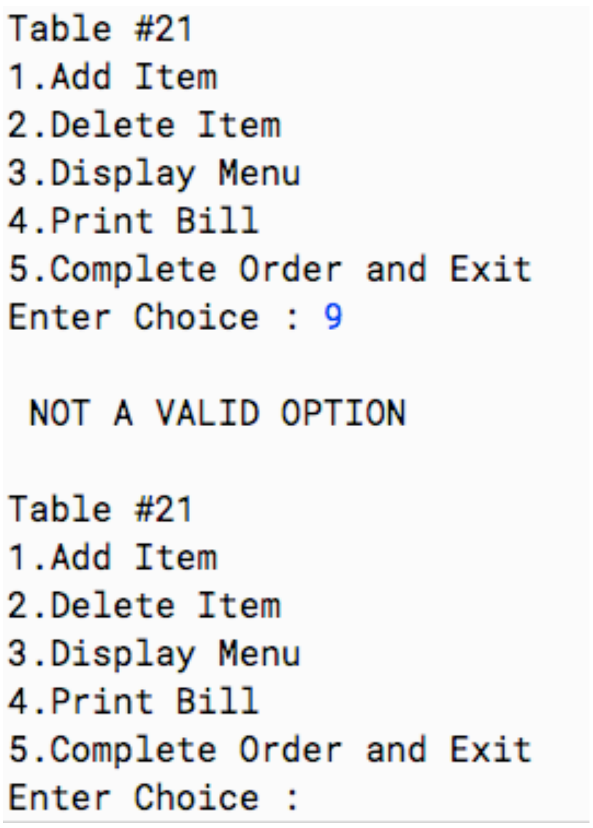
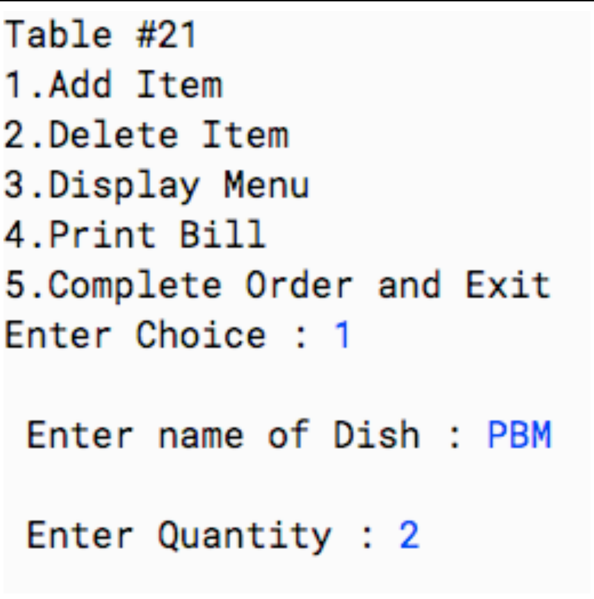
Uses a user-defined method to ensure that the record exists and the price is above 0.

Implementation of Test Plan

Description of the Test	Nature of the Test	Example	Output Screenshot	Result (Pass/ Fail)

Upon Starting up the program, the main menu should be displayed.	To check if the conditions of the menu work	"Hello, Welcome to AVS Restaurant, Choose one of the options below" "1. Dine In" …	 BlueJ: Terminal Window - Waiter
			`Welcome to AVS Fast Food! Please choose an option :`
			`1.Dine-In`
			`2.Pick-Up`
			`3.Delivery`
			`4.Exit`
			`Enter Choice :`
				PASS

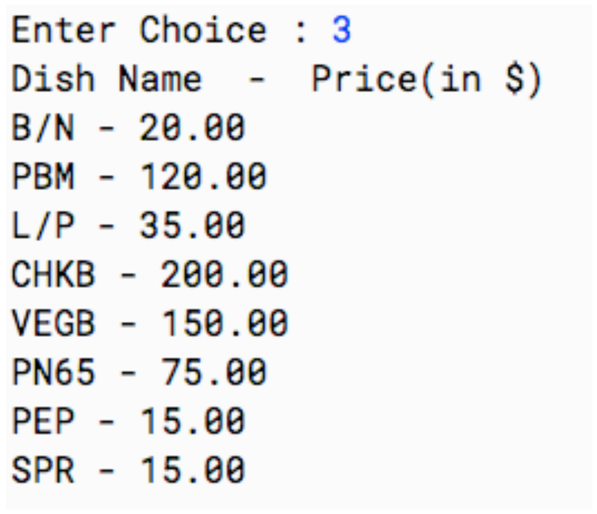
When an option is chosen the menu should navigate to the appropriate sub-menu	Ensuring that the switch case works properly	*User enters 1* "Enter Table Number : " "Choose option" " 1. Add Item" …	 `Enter Choice : 1`
			`Enter Customer Name : Tom`
			`Enter Table # : 21`
			
			`Table #21`
			`1.Add Item`
			`2.Delete Item`
			`3.Display Menu`
			`4.Print Bill`
			`5.Complete Order and Exit`
			`Enter Choice :`
				PASS

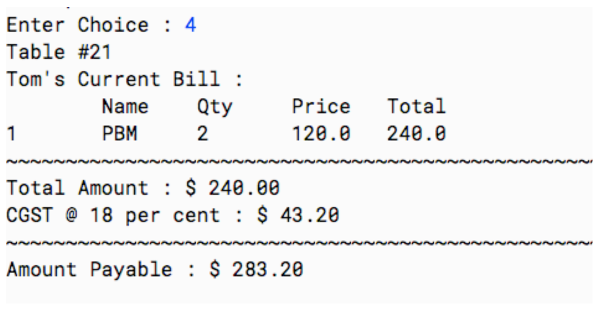
Description of the Test	**Nature of the Test**	**Example**	**Output Screenshot**	**Result (Pass/ Fail)**
The menu should continuously loop until a valid option is entered	Checking if the validity checks work.	“1.Add Item” *User enters 900* “INVALID OPTION ENTER AGAIN” “1. Add Item” …		**PASS**
Check if the user is able to add items to the current order	To check if the linked list is functioning as it should be	*The program should ask for the name of the dish, and after entering, it should add the item to the list.*		***PASS***

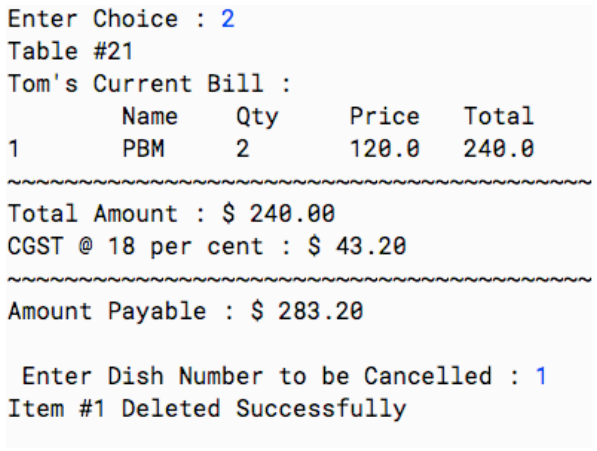
```
Table #21
1.Add Item
2.Delete Item
3.Display Menu
4.Print Bill
5.Complete Order and Exit
Enter Choice : 9

 NOT A VALID OPTION

Table #21
1.Add Item
2.Delete Item
3.Display Menu
4.Print Bill
5.Complete Order and Exit
Enter Choice :
```

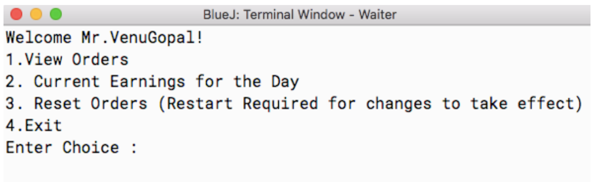
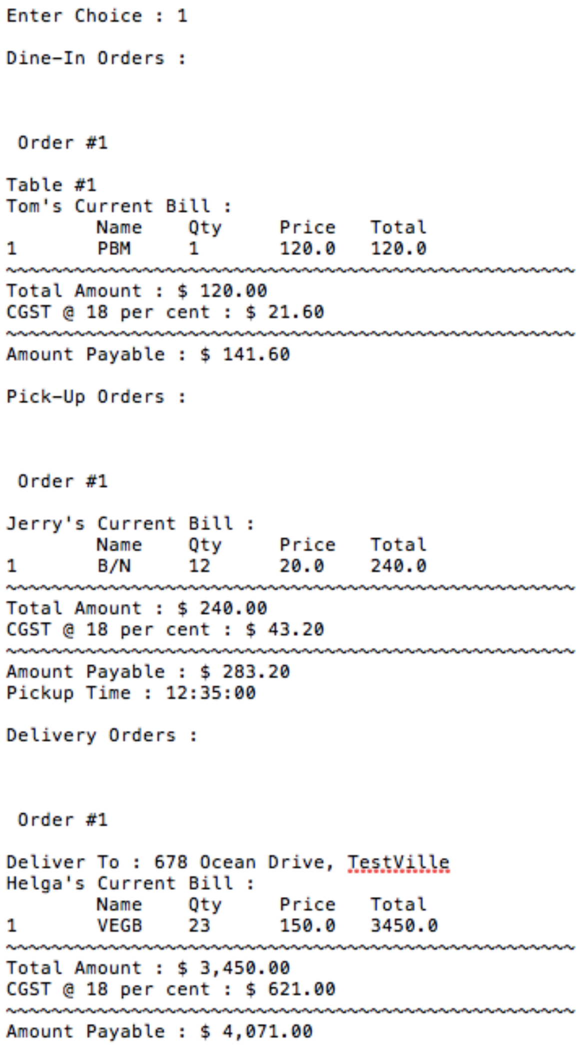
```
Table #21
1.Add Item
2.Delete Item
3.Display Menu
4.Print Bill
5.Complete Order and Exit
Enter Choice : 1

 Enter name of Dish : PBM

 Enter Quantity : 2
```

Description of the Test	**Nature of the Test**	**Example**	**Output Screenshot**	**Result (Pass/ Fail)**

Check if the print menu method works	To ensure that the file access method works	“3. Print Menu” “<<menuitem>> <<price>>”	
			`Enter Choice : 3`
			`Dish Name  -  Price(in $)`
			`B/N - 20.00`
			`PBM - 120.00`
			`L/P - 35.00`
			`CHKB - 200.00`
			`VEGB - 150.00`
			`PN65 - 75.00`
			`PEP - 15.00`
			`SPR - 15.00`
				PASS

Checking whether the print bill option works	Ensuring that the calculations and the formatting of the bill are precise	“4. Print Current Bill” *Bill stating current items in the order and total price so far is printed in a uniform format.*	
			`Enter Choice : 4`
			`Table #21`
			`Tom's Current Bill :`
			`        Name    Qty     Price    Total`
			`1       PBM     2       120.0    240.0`
			`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
			`Total Amount : $ 240.00`
			`CGST @ 18 per cent : $ 43.20`
			`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
			`Amount Payable : $ 283.20`
				PASS

Check if the delete item method works	To check if the code works effectively as it should	*User tells the product to delete item 1 from the order* “Item 1 has been deleted” *Prints the rest of the order after deletion.*	
			`Enter Choice : 2`
			`Table #21`
			`Tom's Current Bill :`
			`        Name    Qty     Price    Total`
			`1       PBM     2       120.0    240.0`
			`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
			`Total Amount : $ 240.00`
			`CGST @ 18 per cent : $ 43.20`
			`~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~`
			`Amount Payable : $ 283.20`
			
			` Enter Dish Number to be Cancelled : 1`
			`Item #1 Deleted Successfully`
				PASS(doesn’t print bill after deletion)

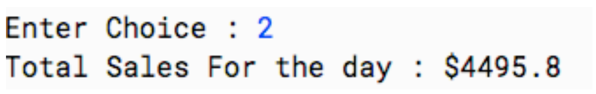
Description of the Test	**Nature of the Test**	**Example**	**Output Screenshot**	**Result (Pass/ Fail)**
Upon opening the Manager module, the main menu should be displayed	To check if the conditions of the menu work	“Hello, Mr. VenuGopal, Choose one of the options below” “1. View Orders” …	 (output A below)	**PASS**
Check if the Display current orders method works	To ensure that the code picks up on all of the permanently stored orders	*Dine In Orders* … *PickUp Orders* … *Delivery Orders* …	 (output B below)	***PASS***

Output A:

```
BlueJ: Terminal Window - Waiter
Welcome Mr.VenuGopal!
1.View Orders
2. Current Earnings for the Day
3. Reset Orders (Restart Required for changes to take effect)
4.Exit
Enter Choice :
```

Output B:

```
Enter Choice : 1

Dine-In Orders :


 Order #1

Table #1
Tom's Current Bill :
        Name    Qty     Price   Total
1       PBM     1       120.0   120.0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Total Amount : $ 120.00
CGST @ 18 per cent : $ 21.60
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Amount Payable : $ 141.60

Pick-Up Orders :


 Order #1

Jerry's Current Bill :
        Name    Qty     Price   Total
1       B/N     12      20.0    240.0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Total Amount : $ 240.00
CGST @ 18 per cent : $ 43.20
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Amount Payable : $ 283.20
Pickup Time : 12:35:00

Delivery Orders :


 Order #1

Deliver To : 678 Ocean Drive, TestVille
Helga's Current Bill :
        Name    Qty     Price   Total
1       VEGB    23      150.0   3450.0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Total Amount : $ 3,450.00
CGST @ 18 per cent : $ 621.00
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Amount Payable : $ 4,071.00
```

Description of the Test	**Nature of the Test**	**Example**	**Output Screenshot**	**Result (Pass/ Fail)**
Check if the calculate total earnings function works	To ensure that the arithmetic calculations are properly programmed	"Total Earnings for the day = $XXXXX"	 `Enter Choice : 2` `Total Sales For the day : $4495.8`	***PASS***

Word Count : 1,284 Words

Works Cited

"Serialization and Deserialization in Java with Example." *GeeksforGeeks*, 10 Feb. 2018, www.geeksforgeeks.org/serialization-in-java/.

Flowcharts created using www.code2flow.com and draw.io

Program created using BlueJ IDE