



K.R. MANGALAM UNIVERSITY

DATA STRUCTURES LAB FILE

Submitted by : Yashraj Rawani

Submitted to : SWATI MAM

Course : B.TECH CSE DATA SCIENCE

Semester : III

Roll No : 2401420037

TASK-1

➤ Store Inventory: To manage items in inventory

{ Add item, remove item, display inventory, search item }

- Code:

```
J InventoryManager.java X
C:\Users\Rakes>OneDrive>Documents>javadsa>J InventoryManager.java > ...
1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 class Product {
5     private String code;
6     private String title;
7     private int stock;
8
9     public Product(String code, String title, int stock) {
10         this.code = code;
11         this.title = title;
12         this.stock = stock;
13     }
14
15     public String getCode() {
16         return code;
17     }
18
19     public String getTitle() {
20         return title;
21     }
22
23     public int getStock() {
24         return stock;
25     }
26
27     @Override
28     public String toString() {
29         return "Product{" +
30             "code=" + code +
31             ", title=" + title +
32             ", stock=" + stock +
33             '}';
34     }
35 }
36
37 Windsurf: Refactor | Explain | Generate Javadoc | X
38 public class InventoryManager {
39
40     private static ArrayList<Product> catalog = new ArrayList<>();
41
42     public static void addItem(Scanner sc) {
43         System.out.print("Enter Item Code: ");
44         String c = sc.nextLine().trim().toUpperCase();
45
46         System.out.print("Enter Item Name: ");
47         String n = sc.nextLine().trim();
48
49         System.out.print("Enter Quantity: ");
50         int q = Integer.parseInt(sc.nextLine());
51
52         Product p = new Product(c, n, q);
53         catalog.add(p);
54
55         System.out.println("Item Added Successfully!\n");
56     }
57
58     public static void removeItem(Scanner sc) {
59         System.out.print("Enter Item Code to Remove: ");
60         String c = sc.nextLine().trim().toUpperCase();
61
62         for (Product p : catalog) {
63             if (p.getCode().equals(c)) {
64                 catalog.remove(p);
65                 System.out.println("Item Removed Successfully!");
66                 break;
67             }
68         }
69     }
70
71     public static void displayInventory() {
72         System.out.println("Inventory Manager");
73         System.out.println("-----");
74         for (Product p : catalog) {
75             System.out.println(p);
76         }
77     }
78
79     public static void searchItem(String code) {
80         for (Product p : catalog) {
81             if (p.getCode().equals(code)) {
82                 System.out.println("Item Found: " + p);
83                 break;
84             }
85         }
86     }
87
88     public static void main(String[] args) {
89         Scanner sc = new Scanner(System.in);
90
91         while (true) {
92             System.out.println("1. Add Item");
93             System.out.println("2. Remove Item");
94             System.out.println("3. Display Inventory");
95             System.out.println("4. Search Item");
96             System.out.println("5. Exit");
97
98             System.out.print("Enter Your Choice: ");
99             int choice = sc.nextInt();
100
101            switch (choice) {
102                case 1:
103                    addItem(sc);
104                    break;
105                case 2:
106                    removeItem(sc);
107                    break;
108                case 3:
109                    displayInventory();
110                    break;
111                case 4:
112                    System.out.print("Enter Item Code to Search: ");
113                    String code = sc.nextLine();
114                    searchItem(code);
115                    break;
116                case 5:
117                    System.out.println("Exiting...");
118                    System.exit(0);
119                default:
120                    System.out.println("Invalid Choice!");
121            }
122        }
123    }
124}
```

```
J InventoryManager.java X
C:\Users\Rakes>OneDrive>Documents>javadsa>J InventoryManager.java > ...
1 package com.javadsa;
2
3 public class InventoryManager {
4
5     private static ArrayList<Product> catalog = new ArrayList<>();
6
7     public static void addItem(Scanner sc) {
8         System.out.print("Enter Item Code: ");
9         String c = sc.nextLine().trim().toUpperCase();
10
11         System.out.print("Enter Item Name: ");
12         String n = sc.nextLine().trim();
13
14         System.out.print("Enter Quantity: ");
15         int q = Integer.parseInt(sc.nextLine());
16
17         Product p = new Product(c, n, q);
18         catalog.add(p);
19
20         System.out.println("Item Added Successfully!\n");
21     }
22
23     public static void removeItem(Scanner sc) {
24         System.out.print("Enter Item Code to Remove: ");
25         String c = sc.nextLine().trim().toUpperCase();
26
27         for (Product p : catalog) {
28             if (p.getCode().equals(c)) {
29                 catalog.remove(p);
30                 System.out.println("Item Removed Successfully!");
31                 break;
32             }
33         }
34     }
35
36     public static void displayInventory() {
37         System.out.println("Inventory Manager");
38         System.out.println("-----");
39         for (Product p : catalog) {
40             System.out.println(p);
41         }
42     }
43
44     public static void searchItem(String code) {
45         for (Product p : catalog) {
46             if (p.getCode().equals(code)) {
47                 System.out.println("Item Found: " + p);
48                 break;
49             }
50         }
51     }
52
53     public static void main(String[] args) {
54         Scanner sc = new Scanner(System.in);
55
56         while (true) {
57             System.out.println("1. Add Item");
58             System.out.println("2. Remove Item");
59             System.out.println("3. Display Inventory");
60             System.out.println("4. Search Item");
61             System.out.println("5. Exit");
62
63             System.out.print("Enter Your Choice: ");
64             int choice = sc.nextInt();
65
66             switch (choice) {
67                 case 1:
68                     addItem(sc);
69                     break;
70                 case 2:
71                     removeItem(sc);
72                     break;
73                 case 3:
74                     displayInventory();
75                     break;
76                 case 4:
77                     System.out.print("Enter Item Code to Search: ");
78                     String code = sc.nextLine();
79                     searchItem(code);
80                     break;
81                 case 5:
82                     System.out.println("Exiting...");
83                     System.exit(0);
84                 default:
85                     System.out.println("Invalid Choice!");
86             }
87         }
88     }
89 }
```

```
InventoryManager.java | X
C > Users > Rakes > OneDrive > Documents > javadsa > J InventoryManager.java > ...
33 public class InventoryManager {
37     public static void addItem(Scanner sc) {
38         System.out.println("Item Added Successfully!\n");
39     }
40
41     public static void removeItem(Scanner sc) {
42         System.out.print("Enter Item Code to Remove: ");
43         String c = sc.nextLine().trim().toUpperCase();
44
45         boolean removed = catalog.removeIf(item -> item.getCode().equals(c));
46
47         if (removed) {
48             System.out.println("Item Removed!\n");
49         } else {
50             System.out.println("Item Not Found!\n");
51         }
52     }
53
54     public static void displayInventory() {
55         if (catalog.isEmpty()) {
56             System.out.println("Inventory is Empty!\n");
57             return;
58         }
59
60         System.out.println("--- Inventory List ---");
61         for (Product p : catalog) {
62             System.out.println(p);
63         }
64         System.out.println();
65     }
66
67     public static void searchItem(Scanner sc) {
68         System.out.print("Enter Item Code to Search: ");
69         String c = sc.nextLine().trim().toUpperCase();
70
71         for (Product p : catalog) {
72             if (p.getCode().equals(c)) {
73                 System.out.println("Item Found:");
74                 System.out.println(p + "\n");
75                 return;
76             }
77         }
78         System.out.println("Item Not Found!\n");
79     }
80
81     public static void main(String[] args) {
82         Scanner sc = new Scanner(System.in);
83         int choice;
84
85         while (true) {
86             System.out.println("==== Store Inventory Menu ====");
87             System.out.println("1. Add Item");
88             System.out.println("2. Remove Item");
89             System.out.println("3. Display Inventory");
90             System.out.println("4. Search Item");
91             System.out.println("5. Exit");
92             System.out.print("Enter Choice: ");
93
94             choice = Integer.parseInt(sc.nextLine());
95
96             switch (choice) {
97                 case 1:
98                     addItem(sc);
99                 break;
100                case 2:
101                    removeItem(sc);
102                break;
103                case 3:
104                    displayInventory();
105                break;
106                case 4:
107                    searchItem(sc);
108                break;
109                case 5:
110                    System.out.println("Exiting...");
111                    System.exit(0);
112                break;
113            }
114        }
115    }
116}
```

Ln 123, Col 1 Spaces: 4 UTF-8 CRLF {} Java Go Live Windsurf: ⚡

```
InventoryManager.java | X
C > Users > Rakes > OneDrive > Documents > javadsa > J InventoryManager.java > ...
33 public class InventoryManager {
37     public static void addItem(Scanner sc) {
38         System.out.println("Item Added Successfully!\n");
39     }
40
41     public static void removeItem(Scanner sc) {
42         System.out.print("Enter Item Code to Remove: ");
43         String c = sc.nextLine().trim().toUpperCase();
44
45         boolean removed = catalog.removeIf(item -> item.getCode().equals(c));
46
47         if (removed) {
48             System.out.println("Item Removed!\n");
49         } else {
50             System.out.println("Item Not Found!\n");
51         }
52     }
53
54     public static void displayInventory() {
55         if (catalog.isEmpty()) {
56             System.out.println("Inventory is Empty!\n");
57             return;
58         }
59
60         System.out.println("--- Inventory List ---");
61         for (Product p : catalog) {
62             System.out.println(p);
63         }
64         System.out.println();
65     }
66
67     public static void searchItem(Scanner sc) {
68         System.out.print("Enter Item Code to Search: ");
69         String c = sc.nextLine().trim().toUpperCase();
70
71         for (Product p : catalog) {
72             if (p.getCode().equals(c)) {
73                 System.out.println("Item Found:");
74                 System.out.println(p + "\n");
75                 return;
76             }
77         }
78         System.out.println("Item Not Found!\n");
79     }
80
81     public static void main(String[] args) {
82         Scanner sc = new Scanner(System.in);
83         int choice;
84
85         while (true) {
86             System.out.println("==== Store Inventory Menu ====");
87             System.out.println("1. Add Item");
88             System.out.println("2. Remove Item");
89             System.out.println("3. Display Inventory");
90             System.out.println("4. Search Item");
91             System.out.println("5. Exit");
92             System.out.print("Enter Choice: ");
93
94             choice = Integer.parseInt(sc.nextLine());
95
96             switch (choice) {
97                 case 1:
98                     addItem(sc);
99                 break;
100                case 2:
101                    removeItem(sc);
102                break;
103                case 3:
104                    displayInventory();
105                break;
106                case 4:
107                    searchItem(sc);
108                break;
109                case 5:
110                    System.out.println("Exiting...");
111                    System.exit(0);
112                break;
113            }
114        }
115    }
116}
```

Ln 123, Col 1 Spaces: 4 UTF-8 CRLF {} Java Go Live Windsurf: ⚡

A screenshot of a Java IDE interface. The main window displays the code for `InventoryManager.java`. The code implements a menu system with five options: Add Item, Remove Item, Display Inventory, Search Item, and Exit. It uses `Scanner` to read user input and `System.out.println` to display output. The IDE includes a toolbar with various icons for file operations, a search bar at the top, and a status bar at the bottom indicating the current file is ready.

```
33  public class InventoryManager {  
34      //...  
35      public static void main(String[] args) {  
36          Scanner sc = new Scanner(System.in);  
37          int choice;  
38  
39          while (true) {  
40              System.out.println("==== Store Inventory Menu ====");  
41              System.out.println("1. Add Item");  
42              System.out.println("2. Remove Item");  
43              System.out.println("3. Display Inventory");  
44              System.out.println("4. Search Item");  
45              System.out.println("5. Exit");  
46              System.out.print("Enter Choice: ");  
47  
48              choice = Integer.parseInt(sc.nextLine());  
49  
50              switch (choice) {  
51                  case 1: addItem(sc); break;  
52                  case 2: removeItem(sc); break;  
53                  case 3: displayInventory(); break;  
54                  case 4: searchItem(sc); break;  
55                  case 5:  
56                      System.out.println("Exiting Program...");  
57                      sc.close();  
58                      return;  
59                  default:  
60                      System.out.println("Invalid Option!\n");  
61              }  
62          }  
63      }  
64  }
```

*Output:

The screenshot shows a terminal window in Visual Studio Code. The terminal tab is selected at the top. The command line shows a Java command being run: `PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa\" ; if ($?) { javac InventoryManager.java } ; if ($?) { java InventoryManager }`. Below the command, the terminal displays the output of a Java application. The application's main menu is shown twice, followed by user input and a success message.

```
PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa\" ; if ($?) { javac InventoryManager.java } ; if ($?) { java InventoryManager }
==== Store Inventory Menu ====
1. Add Item
2. Remove Item
3. Display Inventory
4. Search Item
5. Exit
Enter Choice: 1
Enter Item Code: 54
Enter Item Name: yttffffh
Enter Quantity: 55
Item Added Successfully!
==== Store Inventory Menu ====
1. Add Item
2. Remove Item
3. Display Inventory
4. Search Item
5. Exit
Enter Choice: |
```

TASK-2

- Inventory Stock Manager:Reduce stock according to sales and identify zero stock.

- Code:



```
File Edit Selection View ... < > Search

J InventoryStockManager.java 1 X
C:\Users\Rakes>OneDrive>Documents>javadsa>J InventoryStockManager.java > InventoryStockManager

127
128
129
130 import java.util.HashMap;
131 import java.util.Map;
132 import java.util.Scanner;
Windsurf: Refactor | Explain
133 class StockItem {
134     private String sku;
135     private String name;
136     private int quantity;
137
138     public StockItem(String sku, String name, int quantity) {
139         this.sku = sku;
140         this.name = name;
141         this.quantity = Math.max(a: 0, quantity);
142     }
143
144     public String getSku() { return sku; }
145     public String getName() { return name; }
146     public int getQuantity() { return quantity; }
147
148 Windsurf: Refactor | Explain | Generate Javadoc | X
149     public boolean reduce(int amount) {
150         if (amount < 0) return false;
151         if (quantity >= amount) {
152             quantity -= amount;
153             return true;
154         }
155         return false;
156     }
}
Ln 222, Col 5 Spaces: 4 UTF-8 CRLF { } Java ⌂ Go Live Windsurf: (...)
```

The screenshot shows a Java code editor with the following code:

```
File Edit Selection View ... < > Search
J InventoryStockManager.java 1 X
C:\> Users\Raeks\OneDrive\Documents>javadsa>J InventoryStockManager.java >InventoryStockManager
133 class StockItem {
134     public boolean reduce(int amount) {
135         quantity -= amount;
136     }
137
138     public void increase(int amount) {
139         if (amount > 0) quantity += amount;
140     }
141
142     @Override
143     public String toString() {
144         return String.format("SKU: %s | Name: %s | Qty: %d", sku, name, quantity);
145     }
146 }
147
148 Windsurf: Refactor | Explain | Generate Javadoc | X
149 public class InventoryStockManager {
150
151     private static Map<String, StockItem> inventory = new HashMap<>();
152
153     private static void addProduct(Scanner in) {
154         System.out.print("Enter SKU: ");
155         String sku = in.nextLine().trim().toUpperCase();
156         if (sku.isEmpty()) { System.out.println("SKU cannot be empty."); return; }
157
158         System.out.print("Enter product name: ");
159         String name = in.nextLine().trim();
160         if (name.isEmpty()) { System.out.println("Name cannot be empty."); return; }
161
162         Windsurf: Refactor | Explain | Generate Javadoc | X
163         @Override
164         public String toString() {
165             return String.format("SKU: %s | Name: %s | Qty: %d", sku, name, quantity);
166         }
167     }
168
169 Windsurf: Refactor | Explain
170 public class InventoryStockManager {
171
172     private static Map<String, StockItem> inventory = new HashMap<>();
173
174     private static void addProduct(Scanner in) {
175         System.out.print("Enter SKU: ");
176         String sku = in.nextLine().trim().toUpperCase();
177         if (sku.isEmpty()) { System.out.println("SKU cannot be empty."); return; }
178
179         System.out.print("Enter product name: ");
180         String name = in.nextLine().trim();
181         if (name.isEmpty()) { System.out.println("Name cannot be empty."); return; }
182
183         Windsurf: Refactor | Explain | Generate Javadoc | X
184         @Override
185         public String toString() {
186             return String.format("SKU: %s | Name: %s | Qty: %d", sku, name, quantity);
187         }
188     }
189
190     public static void main(String[] args) {
191         addProduct(new Scanner(System.in));
192     }
193 }
```

The screenshot shows a Java IDE interface with the following details:

- Title Bar:** File Edit Selection View ... < > Search
- Project Explorer:** Shows a tree structure with nodes like J_InventoryStockManager.java, G > Users > Rakes > OneDrive > Documents > javadsa > J_InventoryStockManager.java > J_InventoryStockManager.
- Code Editor:** The main area displays the `InventoryStockManager.java` file with the following content:

```
169 public class InventoryStockManager {  
173     private static void addProduct(Scanner in) {  
180         if (name.isEmpty()) { System.out.println(x: "Name cannot be empty."); return; }  
181  
182         System.out.print(s: "Enter initial quantity: ");  
183         int qty;  
184         try { qty = Integer.parseInt(in.nextLine().trim()); }  
185         catch (NumberFormatException e) { System.out.println(x: "Invalid number."); return; }  
186  
187         if (inventory.containsKey(sku)) {  
188             System.out.println(x: "SKU already exists. Use restock or record sale instead.");  
189             return;  
190         }  
191  
192         inventory.put(sku, new StockItem(sku, name, Math.max(a: 0, qty)));  
193         System.out.println(x: "Product added.");  
194     }  
195  
196     private static void removeProduct(Scanner in) {  
197         System.out.print(s: "Enter SKU to remove: ");  
198         String sku = in.nextLine().trim().toUpperCase();  
199         if (inventory.remove(sku) != null) System.out.println(x: "Product removed.");  
200         else System.out.println(x: "SKU not found.");  
201     }  
202  
203     private static void displayInventory() {  
204         if (inventory.isEmpty()) {  
205             System.out.println(x: "Inventory empty.");  
206             return;  
207         }  
208     }  
209 }
```

- Bottom Status Bar:** Ln 222, Col 5 Spaces: 4 UTRF-CRLF {} Java Go Live Windsurf: (...) Java Ready

```
File Edit Selection View ... ← → Search
J InventoryStockManager.java 1 ×
C > Users > Rakes > OneDrive > Documents > javadsa > J InventoryStockManager.java > InventoryStockManager
169 public class InventoryStockManager {
202
Windsurf: Refactor | Explain | Generate Javadoc | X
203     private static void displayInventory() {
204         if (inventory.isEmpty()) {
205             System.out.println(x: "Inventory empty.");
206             return;
207         }
208         System.out.println(x: "----- Current Inventory -----");
209         for (StockItem item : inventory.values()) {
210             System.out.println(item);
211         }
212     }
213
Windsurf: Refactor | Explain | Generate Javadoc | X
214     private static void searchProduct(Scanner in) {
215         System.out.print(s: "Enter SKU to search: ");
216         String sku = in.nextLine().trim().toUpperCase();
217         StockItem item = inventory.get(sku);
218         if (item == null) System.out.println(x: "Not found.");
219         else System.out.println("Found: " + item);
220     }
221
222
Windsurf: Refactor | Explain | Generate Javadoc | X
223     private static void recordSale(Scanner in) {
224         System.out.print(s: "Enter SKU sold: ");
225         String sku = in.nextLine().trim().toUpperCase();
226         StockItem item = inventory.get(sku);
227         if (item == null) {
228             System.out.println(x: "SKU not found.");
}
Ln 222, Col 5 Spaces:4 UTF-8 CRLF { } Java ⌂ Go Live Windsurf: (...)
```

InventoryStockManager.java

```
169 public class InventoryStockManager {  
223     private static void recordSale(Scanner in) {  
224         if (in.hasNextDouble() && in.nextDouble() > 0) {  
225             System.out.println("Sale recorded. New quantity: " + item.getQuantity());  
226         } else {  
227             System.out.println("Insufficient stock. Current qty: " + item.getQuantity());  
228         }  
229     }  
240 }  
241  
Windsurf: Refactor | Explain | Generate Javadoc | X  
248 private static void restockProduct(Scanner in) {  
249     System.out.print(s: "Enter SKU to restock: ");  
250     String sku = in.nextLine().trim().toUpperCase();  
251     StockItem item = inventory.get(sku);  
252     if (item == null) {  
253         System.out.println(x: "SKU not found.");  
254         return;  
255     }  
256     System.out.print(s: "Enter quantity to add: ");  
257     int add;  
258     try { add = Integer.parseInt(in.nextLine().trim()); }  
259     catch (NumberFormatException e) { System.out.println(x: "Invalid number."); return; }  
260     if (add <= 0) { System.out.println(x: "Must add positive quantity."); return; }  
261     item.increase(add);  
262     System.out.println("Restocked. New quantity: " + item.getQuantity());  
263 }  
264 }
```

Ln 222, Col 5 Spaces: 4 UTF-8 CRLF {} Java Go Live Windsurf: (...) □

InventoryStockManager.java

```
169 public class InventoryStockManager {  
223     private static void restockProduct(Scanner in) {  
224         ...  
263     }  
264 }  
265 Windsurf: Refactor | Explain | Generate Javadoc | X  
266 private static void listZeroStock() {  
267     boolean any = false;  
268     for (StockItem item : inventory.values()) {  
269         if (item.getQuantity() == 0) {  
270             if (!any) {  
271                 System.out.println(x: "Zero-stock items:");  
272                 any = true;  
273             }  
274             System.out.println(item);  
275         }  
276     }  
277     if (!any) System.out.println(x: "No zero-stock items.");  
278 }  
279 Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc | X  
280 public static void main(String[] args) {  
281     Scanner in = new Scanner(System.in);  
282     while (true) {  
283         System.out.println(x: "\n==== Inventory Stock Manager ====");  
284         System.out.println(x: "1. Add product");  
285         System.out.println(x: "2. Remove product");  
286         System.out.println(x: "3. Display inventory");  
287         System.out.println(x: "4. Search product");  
288         System.out.println(x: "5. Record sale (reduce stock)");  
289         System.out.println(x: "6. Restock product (increase stock)");  
290         System.out.println(x: "7. List zero-stock items");  
291     }  
292 }
```

Ln 222, Col 5 Spaces: 4 UTF-8 CRLF {} Java Go Live Windsurf: (...) □

```
InventoryStockManager.java
169 public class InventoryStockManager {
170     public static void main(String[] args) {
171         System.out.println("1. Display inventory");
172         System.out.println("2. Search product");
173         System.out.println("3. Record sale (reduce stock)");
174         System.out.println("4. Restock product (increase stock)");
175         System.out.println("5. List zero-stock items");
176         System.out.println("6. Exit");
177         System.out.print("Choose (1-6): ");
178
179         String choice = in.nextLine().trim();
180         switch (choice) {
181             case "1": addProduct(in); break;
182             case "2": removeProduct(in); break;
183             case "3": displayInventory(); break;
184             case "4": searchProduct(in); break;
185             case "5": recordSale(in); break;
186             case "6": restockProduct(in); break;
187             case "7": listZeroStock(); break;
188             case "8":
189                 System.out.println("Goodbye.");
190                 in.close();
191                 return;
192             default:
193                 System.out.println("Invalid choice. Try again.");
194         }
195     }
196 }
197 }
```

Ln 222, Col 5 Spaces: 4 UTF-8 CRLF {} Java Go Live Windsurf: ...

Output:

```
Choose (1-8): cd "c:\Users\Rakes\OneDrive\Documents\javadsa\" ; if ($?) { javac InventoryStockManager.java } ; if ($?) { java InventoryStockManager }
Invalid choice. Try again.

*** Inventory Stock Manager ***
1. Add product
2. Remove product
3. Display inventory
4. Search product
5. Record sale (reduce stock)
6. Restock product (increase stock)
7. List zero-stock items
8. Exit
Choose (1-8): 1
Enter SKU: 55
Enter product name: yfh
Enter initial quantity: 6
Product added.

*** Inventory Stock Manager ***
1. Add product
2. Remove product
3. Display inventory
4. Search product
5. Record sale (reduce stock)
6. Restock product (increase stock)
7. List zero-stock items
8. Exit
Choose (1-8): 1
Enter SKU: yfh
Enter product name: yfh
Enter initial quantity: 6
Product added.
```

Ln 222, Col 5 Spaces: 4 UTF-8 CRLF {} Java Go Live Windsurf: ...

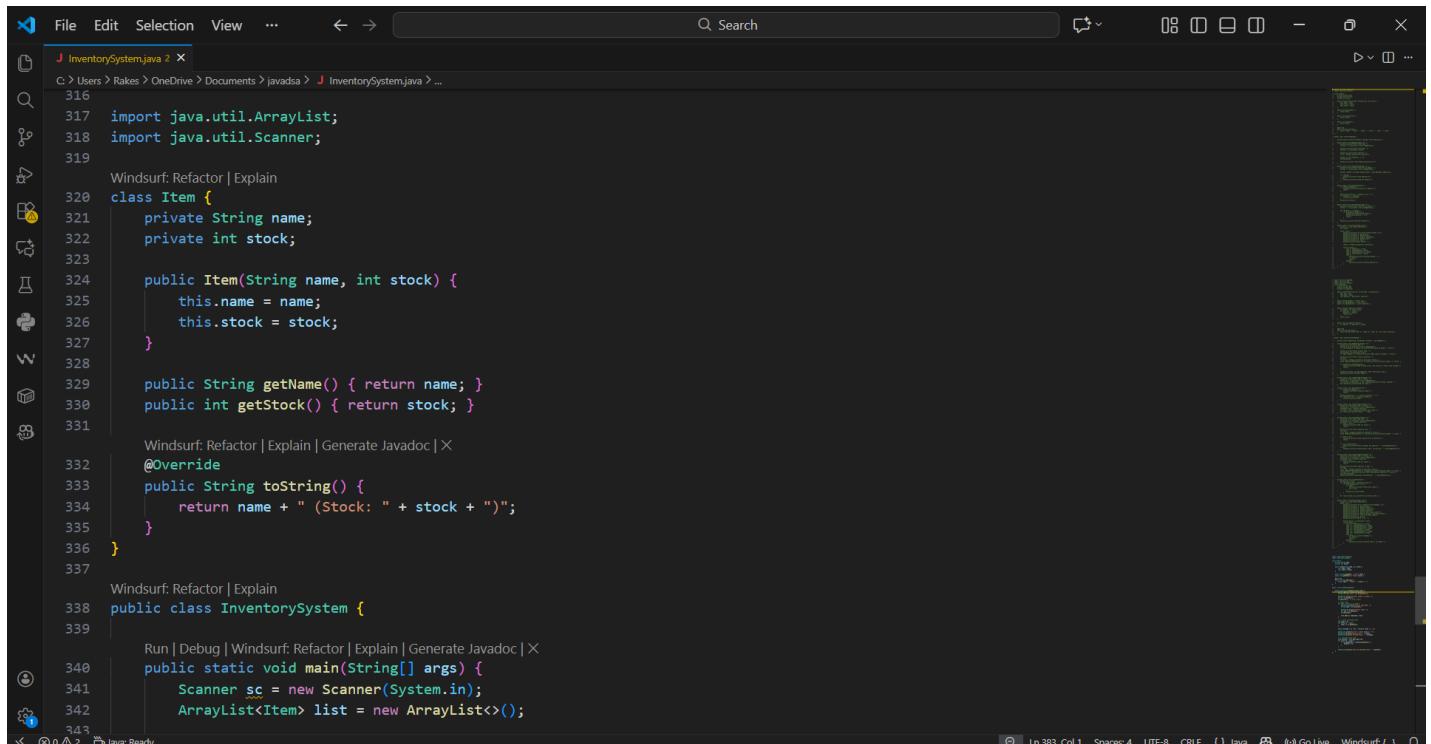
TASK-3

Inventory system with two functionalities:

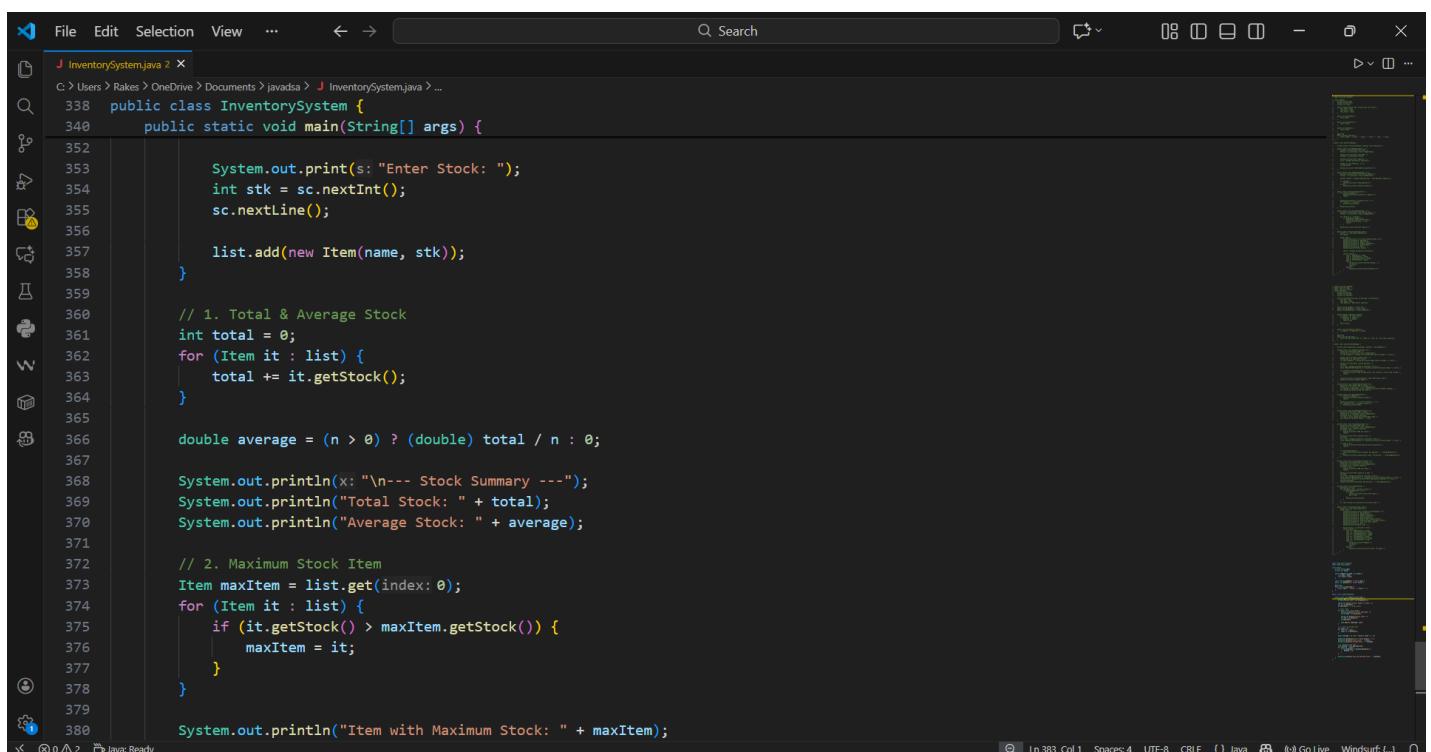
1. Calculate Total & Average Stock

2. Find Maximum Stock Item

*CODE:



```
InventorySystem.java 2 x
C:\Users\Rakes>OneDrive>Documents>javadsa>J InventorySystem.java > ...
316
317 import java.util.ArrayList;
318 import java.util.Scanner;
319
320 Windsurf: Refactor | Explain
321 class Item {
322     private String name;
323     private int stock;
324
325     public Item(String name, int stock) {
326         this.name = name;
327         this.stock = stock;
328     }
329
330     public String getName() { return name; }
331     public int getStock() { return stock; }
332
333 Windsurf: Refactor | Explain | Generate Javadoc | X
334 @Override
335     public String toString() {
336         return name + " (Stock: " + stock + ")";
337     }
338
339 Windsurf: Refactor | Explain
340 public class InventorySystem {
341
342     public static void main(String[] args) {
343         Scanner sc = new Scanner(System.in);
344         ArrayList<Item> list = new ArrayList<>();
345     }
346
347 Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc | X
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
```



```
InventorySystem.java 2 x
C:\Users\Rakes>OneDrive>Documents>javadsa>J InventorySystem.java > ...
338 public class InventorySystem {
339     public static void main(String[] args) {
340
341         Scanner sc = new Scanner(System.in);
342         ArrayList<Item> list = new ArrayList<>();
343
344         System.out.print("Enter Stock: ");
345         int stk = sc.nextInt();
346         sc.nextLine();
347
348         list.add(new Item(name, stk));
349     }
350
351     // 1. Total & Average Stock
352     int total = 0;
353     for (Item it : list) {
354         total += it.getStock();
355     }
356
357     double average = (n > 0) ? (double) total / n : 0;
358
359     System.out.println(x: \n--- Stock Summary ---");
360     System.out.println("Total Stock: " + total);
361     System.out.println("Average Stock: " + average);
362
363     // 2. Maximum Stock Item
364     Item maxItem = list.get(index: 0);
365     for (Item it : list) {
366         if (it.getStock() > maxItem.getStock()) {
367             maxItem = it;
368         }
369     }
370
371     System.out.println("Item with Maximum Stock: " + maxItem);
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
497
498
499
499
500
501
502
503
504
505
506
507
507
508
509
509
510
511
512
513
514
515
515
516
517
517
518
518
519
519
520
521
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
```

OUTPUT:

A screenshot of a terminal window within a code editor interface. The terminal tab is active, showing the following command-line session:

```
PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa\" ; if ($?) { javac InventorySystem.java } ; if ($?) { java InventorySystem }
```

The user has entered the following input:

```
Enter number of items: 6
Enter Item Name: gg
Enter Stock: 5
Enter Item Name: mgm
Enter Stock: 
```

The terminal window includes standard navigation and search controls at the top, and a status bar at the bottom indicating the current file is Java Ready.

TASK 4: STACK OPERATIONS: {push, pop, peek, display}

CODE:

The screenshot shows a Java code editor interface with the following details:

- File Path:** C:\Users\Rakes>OneDrive>Documents>javadsa>J StackDemo.java
- Code Content:** The code defines a class StackDemo with methods pop(), peek(), and display().
 - pop():** Returns the top element of the stack.
 - peek():** Prints the top element or indicates if the stack is empty.
 - display():** Prints all elements in the stack.
- Toolbars and Icons:** Standard Java development toolbar icons for file operations, search, and code navigation.
- Code Navigation:** A vertical sidebar on the right side of the editor displays a tree view of the code structure, showing nodes for StackDemo.java, StackDemo.class, and various method implementations.
- Status Bar:** Shows the current line (Ln 495), column (Col 1), and spaces (Spaces: 4). It also indicates the file is Java Ready.

A screenshot of a Java code editor interface. The main window displays the code for `StackDemo.java`. The code implements a menu system with five options: push, pop, peek, display, and exit. The editor features syntax highlighting for Java keywords and punctuation. A code completion dropdown menu is open on the right side of the screen, listing various Java methods and classes, such as `System.out.println`, `int`, `sc.nextInt`, `st.push`, `st.pop`, `st.peek`, `st.display`, and `sc.close`. The status bar at the bottom shows the current file path, line number (Ln 495), column number (Col 1), and encoding (UTF-8). The bottom right corner includes links for Java documentation, Go Live, and Windsurf.

```
StackDemo.java
C:\Users\Rakes>OneDrive>Documents>javadsa>J StackDemo.java>...
390 public class StackDemo {
446     public static void main(String[] args) {
464         switch (choice) {
465             case 1:
466                 System.out.print(s: "Enter value to push: ");
467                 int v = sc.nextInt();
468                 st.push(v);
469                 break;
470
471             case 2:
472                 st.pop();
473                 break;
474
475             case 3:
476                 st.peek();
477                 break;
478
479             case 4:
480                 st.display();
481                 break;
482
483             case 5:
484                 System.out.println(x: "Exiting...");
485                 sc.close();
486                 return;
487
488             default:
489                 System.out.println(x: "Invalid Choice!");
490
491         }
492     }
}
```

Output:

The screenshot shows a terminal window within a code editor interface. The terminal output is as follows:

```
PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa\" ; if ($?) { javac StackDemo.java } ; if ($?) { java StackDemo }

Enter stack size: 5
--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter choice: 1
Enter value to push: 65
Pushed: 65

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter choice: 4
Stack elements:
65

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter choice: 
```

The terminal shows the execution of a Java application named `StackDemo`. It first asks for the stack size (5). Then it displays a menu with options 1 through 5. Option 1 is selected, pushing the value 65 onto the stack. The stack is then displayed, showing the single element 65. Finally, the menu is shown again, but no choice is made.

TASK-5: Implementation of stack browser:

Code:

The screenshot shows a Java code editor interface with a sidebar on the left containing various icons for file operations like Open, Save, Find, and Run. The main area displays a Java program named `BrowserStack.java`. The code implements a simple browser-like application using stacks for back and forward navigation and a scanner for user input. The code is as follows:

```
497  
498  
499 import java.util.Stack;  
500 import java.util.Scanner;  
501  
502 Windsurf: Refactor | Explain  
503 public class BrowserStack {  
504     Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc | X  
505     public static void main(String[] args) {  
506         Stack<String> backStack = new Stack<>();  
507         Stack<String> forwardStack = new Stack<>();  
508         Scanner sc = new Scanner(System.in);  
509  
510         String currentPage = "Home"; // Starting page  
511         int choice;  
512  
513         while (true) {  
514  
515             System.out.println("\nCurrent Page: " + currentPage);  
516             System.out.println("==== Browser Menu ===");  
517             System.out.println("1. Visit New Page");  
518             System.out.println("2. Go Back");  
519             System.out.println("3. Go Forward");  
520             System.out.println("4. Display History");  
521             System.out.println("5. Exit");  
522             System.out.print("Enter choice: ");  
523             choice = sc.nextInt();  
524             sc.nextLine(); // clear buffer  
525         }  
526     }  
527 }
```

```
File Edit Selection View < > Search
J BrowserStack.java x
G:\Users\Rakes>OneDrive>Documents>javadsa>J BrowserStack.java ...
502 public class BrowserStack {
504     public static void main(String[] args) {
532         backStack.push(currentPage); // push old page into back history
533         currentPage = nextPage;
534         forwardStack.clear(); // clear forward history
535         break;
536     }
537
538     case 2:
539         if (backStack.isEmpty()) {
540             System.out.println(x: "No page in BACK history!");
541         } else {
542             forwardStack.push(currentPage);
543             currentPage = backStack.pop();
544         }
545         break;
546
547     case 3:
548         if (forwardStack.isEmpty()) {
549             System.out.println(x: "No page in FORWARD history!");
550         } else {
551             backStack.push(currentPage);
552             currentPage = forwardStack.pop();
553         }
554         break;
555
556     case 4:
557         System.out.println(x: "\n--- BACK History ---");
558         System.out.println(backStack);
559
560         System.out.println(x: "--- FORWARD History ---");
561         System.out.println(forwardStack);
562     }
563 }
```

The screenshot shows a Java code editor with the following code:

```
File Edit Selection View ... < > Search J BrowserStack.java
```

```
C:\> Users > Rakes > OneDrive > Documents > javadsa > J BrowserStack.java > ...
```

```
502 public class BrowserStack {  
504     public static void main(String[] args) {  
505         backStack.push(currentPage);  
506         currentPage = forwardStack.pop();  
507         break;  
508     }  
509     case 4:  
510         System.out.println("---- BACK History ----");  
511         System.out.println(backStack);  
512         System.out.println("---- FORWARD History ----");  
513         System.out.println(forwardStack);  
514         System.out.println("Current Page: " + currentPage);  
515         break;  
516     case 5:  
517         System.out.println("Exiting Browser...");  
518         sc.close();  
519         return;  
520     default:  
521         System.out.println("Invalid Choice!");  
522     }  
523 }  
524 }  
525 }  
526 }  
527 }
```

The code implements a browser history stack using two stacks: `backStack` and `forwardStack`. It handles five cases: 1) navigating back, 2) navigating forward, 3) viewing the current page, 4) exiting the browser, and 5) handling invalid user input.

Output:

The screenshot shows a terminal window within a code editor interface. The terminal tab is active, displaying the following text:

```
PS C:\Users\Rakes> cd "c:/Users/Rakes/OneDrive/Documents\javadsa" ; if ($?) { javac BrowserStack.java } ; if ($?) { java BrowserStack }
```

Below the command, a simulated browser menu is shown:

```
Current Page: Home  
--- Browser Menu ---  
1. Visit New Page  
2. Go Back  
3. Go Forward  
4. Display History  
5. Exit  
Enter choice: 1  
Enter new page URL: 55g
```

Another set of menu options follows:

```
Current Page: 55g  
--- Browser Menu ---  
1. Visit New Page  
2. Go Back  
3. Go Forward  
4. Display History  
5. Exit  
Enter choice: 4
```

Then, a history simulation is displayed:

```
--- BACK History ---  
[Home]  
--- FORWARD History ---  
[]  
Current Page: 55g
```

Finally, another browser menu is shown:

```
Current Page: 55g  
--- Browser Menu ---  
1. Visit New Page  
2. Go Back  
3. Go Forward  
4. Display History  
5. Exit  
Enter choice: |
```

The bottom status bar of the code editor indicates:

Ln 576, Col 1 Spaces: 4 UTF-8 CRLF {} Java ⚡ Go Live Windsurf: {}

TASK-6: Insertion in circular linkedlist

CODE:

The screenshot shows a Java code editor with the file `CircularLinkedList.java` open. The code implements a circular linked list with methods for inserting at the start and end, and after a specific value. The code is annotated with 'Windsurf: Refactor | Explain' comments. The interface includes standard file operations like File, Edit, Selection, View, and a search bar.

```
File Edit Selection View ... ← → Search
C > Users > Rakes > OneDrive > Documents > javadsa > CircularLinkedList.java ...
579
  Windsurf: Refactor | Explain
580  class Node {
581    int data;
582    Node next;
583
584    Node(int data) {
585      this.data = data;
586    }
587  }
588
  Windsurf: Refactor | Explain
589  public class CircularLinkedList {
590
591    Node head = null;
592
593    // Insert at the beginning
594    Windsurf: Refactor | Explain | X
595    public void insertAtStart(int value) {
596      Node newNode = new Node(value);
597
598      if (head == null) {
599        head = newNode;
600        newNode.next = head; // Point to itself
601      } else {
602        Node temp = head;
603
604        // Go to last node
605        while (temp.next != head) {
606          temp = temp.next;
607        }
608
609        temp.next = newNode;
610        newNode.next = head;
611      }
612
613    // Insert after specific value
614    Windsurf: Refactor | Explain | X
615    public void insertAfter(int key, int value) {
616      if (head == null) {
617        System.out.println("List is empty");
618        return;
619      }
620
621      Node temp = head;
622
623      do {
624        if (temp.data == key) {
625          Node newNode = new Node(value);
626          newNode.next = temp.next;
627        }
628
629        temp = temp.next;
630      }
631
632    }
633
634    // Insert at the end
635    Windsurf: Refactor | Explain | X
636    public void insertAtEnd(int value) {
637      Node newNode = new Node(value);
638
639      if (head == null) {
640        head = newNode;
641        newNode.next = head;
642      } else {
643        Node temp = head;
644
645        do {
646          if (temp.next == head) {
647            temp.next = newNode;
648            newNode.next = head;
649          }
650          temp = temp.next;
651        }
652
653        temp.next = newNode;
654        newNode.next = head;
655      }
656
657    }
658
659    // Print the list
660    public void printList() {
661      Node temp = head;
662
663      do {
664        System.out.print(temp.data + " ");
665        temp = temp.next;
666      }
667
668      System.out.println();
669    }
670
671    // Delete a node by value
672    public void delete(int value) {
673      Node temp = head;
674
675      if (head == null) {
676        System.out.println("List is empty");
677        return;
678      }
679
680      do {
681        if (temp.data == value) {
682          if (temp.next == head) {
683            head = null;
684            temp.next = null;
685          } else {
686            Node nextNode = temp.next;
687            temp.next = nextNode.next;
688            nextNode.next = head;
689          }
690        }
691        temp = temp.next;
692      }
693
694      if (head == null) {
695        System.out.println("List is empty");
696      }
697    }
698
699    // Find a node by value
700    public Node find(int value) {
701      Node temp = head;
702
703      do {
704        if (temp.data == value) {
705          return temp;
706        }
707        temp = temp.next;
708      }
709
710      return null;
711    }
712
713    // Get the size of the list
714    public int getSize() {
715      Node temp = head;
716
717      int count = 0;
718
719      do {
720        count++;
721        temp = temp.next;
722      }
723
724      return count;
725    }
726
727    // Reverse the list
728    public void reverse() {
729      Node temp = head;
730
731      if (head == null) {
732        System.out.println("List is empty");
733        return;
734      }
735
736      do {
737        Node nextNode = temp.next;
738        temp.next = head;
739        head = temp;
740        temp = nextNode;
741      }
742
743      temp.next = head;
744    }
745
746    // Sort the list
747    public void sort() {
748      Node temp = head;
749
750      if (head == null) {
751        System.out.println("List is empty");
752        return;
753      }
754
755      do {
756        Node nextNode = temp.next;
757
758        if (temp.data > nextNode.data) {
759          int tempData = temp.data;
760          temp.data = nextNode.data;
761          nextNode.data = tempData;
762        }
763        temp = nextNode;
764      }
765
766      temp.next = head;
767    }
768
769    // Get the middle element
770    public Node getMiddle() {
771      Node slow = head;
772      Node fast = head;
773
774      if (head == null) {
775        System.out.println("List is empty");
776        return null;
777      }
778
779      do {
780        if (fast.next == head) {
781          return slow;
782        }
783        slow = slow.next;
784        fast = fast.next.next;
785      }
786
787      return null;
788    }
789
790    // Get the k-th element from the end
791    public Node getKthFromEnd(int k) {
792      Node slow = head;
793      Node fast = head;
794
795      if (head == null) {
796        System.out.println("List is empty");
797        return null;
798      }
799
800      do {
801        if (fast.next == head) {
802          return slow;
803        }
804        if (k == 0) {
805          slow = slow.next;
806        }
807        fast = fast.next.next;
808        k--;
809      }
810
811      return null;
812    }
813
814    // Get the intersection point of two lists
815    public Node getIntersectionPoint(CircularLinkedList otherList) {
816      Node head1 = this.head;
817      Node head2 = otherList.head;
818
819      if (head1 == null || head2 == null) {
820        System.out.println("One or both lists are empty");
821        return null;
822      }
823
824      Node slow1 = head1;
825      Node slow2 = head2;
826
827      do {
828        if (slow1.next == head1) {
829          slow1 = head1;
830        }
831        if (slow2.next == head2) {
832          slow2 = head2;
833        }
834        if (slow1 == slow2) {
835          return slow1;
836        }
837        slow1 = slow1.next;
838        slow2 = slow2.next;
839      }
840
841      return null;
842    }
843
844    // Get the length of the longest common subsequence
845    public int getLCSLength(CircularLinkedList otherList) {
846      Node head1 = this.head;
847      Node head2 = otherList.head;
848
849      if (head1 == null || head2 == null) {
850        System.out.println("One or both lists are empty");
851        return 0;
852      }
853
854      Node slow1 = head1;
855      Node slow2 = head2;
856
857      int lcsLength = 0;
858
859      do {
860        if (slow1.next == head1) {
861          slow1 = head1;
862        }
863        if (slow2.next == head2) {
864          slow2 = head2;
865        }
866        if (slow1 == slow2) {
867          lcsLength++;
868        }
869        slow1 = slow1.next;
870        slow2 = slow2.next;
871      }
872
873      return lcsLength;
874    }
875
876    // Get the length of the longest common prefix
877    public int getLCPLength(CircularLinkedList otherList) {
878      Node head1 = this.head;
879      Node head2 = otherList.head;
880
881      if (head1 == null || head2 == null) {
882        System.out.println("One or both lists are empty");
883        return 0;
884      }
885
886      Node slow1 = head1;
887      Node slow2 = head2;
888
889      int lcpLength = 0;
890
891      do {
892        if (slow1.next == head1) {
893          slow1 = head1;
894        }
895        if (slow2.next == head2) {
896          slow2 = head2;
897        }
898        if (slow1 == slow2) {
899          lcpLength++;
900        }
901        slow1 = slow1.next;
902        slow2 = slow2.next;
903      }
904
905      return lcpLength;
906    }
907
908    // Get the length of the longest common suffix
909    public int getLCSLength(CircularLinkedList otherList) {
910      Node head1 = this.head;
911      Node head2 = otherList.head;
912
913      if (head1 == null || head2 == null) {
914        System.out.println("One or both lists are empty");
915        return 0;
916      }
917
918      Node slow1 = head1;
919      Node slow2 = head2;
920
921      int lcsLength = 0;
922
923      do {
924        if (slow1.next == head1) {
925          slow1 = head1;
926        }
927        if (slow2.next == head2) {
928          slow2 = head2;
929        }
930        if (slow1 == slow2) {
931          lcsLength++;
932        }
933        slow1 = slow1.next;
934        slow2 = slow2.next;
935      }
936
937      return lcsLength;
938    }
939
940    // Get the length of the longest common prefix
941    public int getLCPLength(CircularLinkedList otherList) {
942      Node head1 = this.head;
943      Node head2 = otherList.head;
944
945      if (head1 == null || head2 == null) {
946        System.out.println("One or both lists are empty");
947        return 0;
948      }
949
950      Node slow1 = head1;
951      Node slow2 = head2;
952
953      int lcpLength = 0;
954
955      do {
956        if (slow1.next == head1) {
957          slow1 = head1;
958        }
959        if (slow2.next == head2) {
960          slow2 = head2;
961        }
962        if (slow1 == slow2) {
963          lcpLength++;
964        }
965        slow1 = slow1.next;
966        slow2 = slow2.next;
967      }
968
969      return lcpLength;
970    }
971
972    // Get the length of the longest common suffix
973    public int getLCSLength(CircularLinkedList otherList) {
974      Node head1 = this.head;
975      Node head2 = otherList.head;
976
977      if (head1 == null || head2 == null) {
978        System.out.println("One or both lists are empty");
979        return 0;
980      }
981
982      Node slow1 = head1;
983      Node slow2 = head2;
984
985      int lcsLength = 0;
986
987      do {
988        if (slow1.next == head1) {
989          slow1 = head1;
990        }
991        if (slow2.next == head2) {
992          slow2 = head2;
993        }
994        if (slow1 == slow2) {
995          lcsLength++;
996        }
997        slow1 = slow1.next;
998        slow2 = slow2.next;
999      }
1000
1001      return lcsLength;
1002    }
1003
1004    // Get the length of the longest common prefix
1005    public int getLCPLength(CircularLinkedList otherList) {
1006      Node head1 = this.head;
1007      Node head2 = otherList.head;
1008
1009      if (head1 == null || head2 == null) {
1010        System.out.println("One or both lists are empty");
1011        return 0;
1012      }
1013
1014      Node slow1 = head1;
1015      Node slow2 = head2;
1016
1017      int lcpLength = 0;
1018
1019      do {
1020        if (slow1.next == head1) {
1021          slow1 = head1;
1022        }
1023        if (slow2.next == head2) {
1024          slow2 = head2;
1025        }
1026        if (slow1 == slow2) {
1027          lcpLength++;
1028        }
1029        slow1 = slow1.next;
1030        slow2 = slow2.next;
1031      }
1032
1033      return lcpLength;
1034    }
1035
1036    // Get the length of the longest common subsequence
1037    public int getLCSLength(CircularLinkedList otherList) {
1038      Node head1 = this.head;
1039      Node head2 = otherList.head;
1040
1041      if (head1 == null || head2 == null) {
1042        System.out.println("One or both lists are empty");
1043        return 0;
1044      }
1045
1046      Node slow1 = head1;
1047      Node slow2 = head2;
1048
1049      int lcsLength = 0;
1050
1051      do {
1052        if (slow1.next == head1) {
1053          slow1 = head1;
1054        }
1055        if (slow2.next == head2) {
1056          slow2 = head2;
1057        }
1058        if (slow1 == slow2) {
1059          lcsLength++;
1060        }
1061        slow1 = slow1.next;
1062        slow2 = slow2.next;
1063      }
1064
1065      return lcsLength;
1066    }
1067
1068    // Get the length of the longest common prefix
1069    public int getLCPLength(CircularLinkedList otherList) {
1070      Node head1 = this.head;
1071      Node head2 = otherList.head;
1072
1073      if (head1 == null || head2 == null) {
1074        System.out.println("One or both lists are empty");
1075        return 0;
1076      }
1077
1078      Node slow1 = head1;
1079      Node slow2 = head2;
1080
1081      int lcpLength = 0;
1082
1083      do {
1084        if (slow1.next == head1) {
1085          slow1 = head1;
1086        }
1087        if (slow2.next == head2) {
1088          slow2 = head2;
1089        }
1090        if (slow1 == slow2) {
1091          lcpLength++;
1092        }
1093        slow1 = slow1.next;
1094        slow2 = slow2.next;
1095      }
1096
1097      return lcpLength;
1098    }
1099
1100    // Get the length of the longest common suffix
1101    public int getLCSLength(CircularLinkedList otherList) {
1102      Node head1 = this.head;
1103      Node head2 = otherList.head;
1104
1105      if (head1 == null || head2 == null) {
1106        System.out.println("One or both lists are empty");
1107        return 0;
1108      }
1109
1110      Node slow1 = head1;
1111      Node slow2 = head2;
1112
1113      int lcsLength = 0;
1114
1115      do {
1116        if (slow1.next == head1) {
1117          slow1 = head1;
1118        }
1119        if (slow2.next == head2) {
1120          slow2 = head2;
1121        }
1122        if (slow1 == slow2) {
1123          lcsLength++;
1124        }
1125        slow1 = slow1.next;
1126        slow2 = slow2.next;
1127      }
1128
1129      return lcsLength;
1130    }
1131
1132    // Get the length of the longest common prefix
1133    public int getLCPLength(CircularLinkedList otherList) {
1134      Node head1 = this.head;
1135      Node head2 = otherList.head;
1136
1137      if (head1 == null || head2 == null) {
1138        System.out.println("One or both lists are empty");
1139        return 0;
1140      }
1141
1142      Node slow1 = head1;
1143      Node slow2 = head2;
1144
1145      int lcpLength = 0;
1146
1147      do {
1148        if (slow1.next == head1) {
1149          slow1 = head1;
1150        }
1151        if (slow2.next == head2) {
1152          slow2 = head2;
1153        }
1154        if (slow1 == slow2) {
1155          lcpLength++;
1156        }
1157        slow1 = slow1.next;
1158        slow2 = slow2.next;
1159      }
1160
1161      return lcpLength;
1162    }
1163
1164    // Get the length of the longest common subsequence
1165    public int getLCSLength(CircularLinkedList otherList) {
1166      Node head1 = this.head;
1167      Node head2 = otherList.head;
1168
1169      if (head1 == null || head2 == null) {
1170        System.out.println("One or both lists are empty");
1171        return 0;
1172      }
1173
1174      Node slow1 = head1;
1175      Node slow2 = head2;
1176
1177      int lcsLength = 0;
1178
1179      do {
1180        if (slow1.next == head1) {
1181          slow1 = head1;
1182        }
1183        if (slow2.next == head2) {
1184          slow2 = head2;
1185        }
1186        if (slow1 == slow2) {
1187          lcsLength++;
1188        }
1189        slow1 = slow1.next;
1190        slow2 = slow2.next;
1191      }
1192
1193      return lcsLength;
1194    }
1195
1196    // Get the length of the longest common prefix
1197    public int getLCPLength(CircularLinkedList otherList) {
1198      Node head1 = this.head;
1199      Node head2 = otherList.head;
1200
1201      if (head1 == null || head2 == null) {
1202        System.out.println("One or both lists are empty");
1203        return 0;
1204      }
1205
1206      Node slow1 = head1;
1207      Node slow2 = head2;
1208
1209      int lcpLength = 0;
1210
1211      do {
1212        if (slow1.next == head1) {
1213          slow1 = head1;
1214        }
1215        if (slow2.next == head2) {
1216          slow2 = head2;
1217        }
1218        if (slow1 == slow2) {
1219          lcpLength++;
1220        }
1221        slow1 = slow1.next;
1222        slow2 = slow2.next;
1223      }
1224
1225      return lcpLength;
1226    }
1227
1228    // Get the length of the longest common suffix
1229    public int getLCSLength(CircularLinkedList otherList) {
1230      Node head1 = this.head;
1231      Node head2 = otherList.head;
1232
1233      if (head1 == null || head2 == null) {
1234        System.out.println("One or both lists are empty");
1235        return 0;
1236      }
1237
1238      Node slow1 = head1;
1239      Node slow2 = head2;
1240
1241      int lcsLength = 0;
1242
1243      do {
1244        if (slow1.next == head1) {
1245          slow1 = head1;
1246        }
1247        if (slow2.next == head2) {
1248          slow2 = head2;
1249        }
1250        if (slow1 == slow2) {
1251          lcsLength++;
1252        }
1253        slow1 = slow1.next;
1254        slow2 = slow2.next;
1255      }
1256
1257      return lcsLength;
1258    }
1259
1260    // Get the length of the longest common prefix
1261    public int getLCPLength(CircularLinkedList otherList) {
1262      Node head1 = this.head;
1263      Node head2 = otherList.head;
1264
1265      if (head1 == null || head2 == null) {
1266        System.out.println("One or both lists are empty");
1267        return 0;
1268      }
1269
1270      Node slow1 = head1;
1271      Node slow2 = head2;
1272
1273      int lcpLength = 0;
1274
1275      do {
1276        if (slow1.next == head1) {
1277          slow1 = head1;
1278        }
1279        if (slow2.next == head2) {
1280          slow2 = head2;
1281        }
1282        if (slow1 == slow2) {
1283          lcpLength++;
1284        }
1285        slow1 = slow1.next;
1286        slow2 = slow2.next;
1287      }
1288
1289      return lcpLength;
1290    }
1291
1292    // Get the length of the longest common subsequence
1293    public int getLCSLength(CircularLinkedList otherList) {
1294      Node head1 = this.head;
1295      Node head2 = otherList.head;
1296
1297      if (head1 == null || head2 == null) {
1298        System.out.println("One or both lists are empty");
1299        return 0;
1300      }
1301
1302      Node slow1 = head1;
1303      Node slow2 = head2;
1304
1305      int lcsLength = 0;
1306
1307      do {
1308        if (slow1.next == head1) {
1309          slow1 = head1;
1310        }
1311        if (slow2.next == head2) {
1312          slow2 = head2;
1313        }
1314        if (slow1 == slow2) {
1315          lcsLength++;
1316        }
1317        slow1 = slow1.next;
1318        slow2 = slow2.next;
1319      }
1320
1321      return lcsLength;
1322    }
1323
1324    // Get the length of the longest common prefix
1325    public int getLCPLength(CircularLinkedList otherList) {
1326      Node head1 = this.head;
1327      Node head2 = otherList.head;
1328
1329      if (head1 == null || head2 == null) {
1330        System.out.println("One or both lists are empty");
1331        return 0;
1332      }
1333
1334      Node slow1 = head1;
1335      Node slow2 = head2;
1336
1337      int lcpLength = 0;
1338
1339      do {
1340        if (slow1.next == head1) {
1341          slow1 = head1;
1342        }
1343        if (slow2.next == head2) {
1344          slow2 = head2;
1345        }
1346        if (slow1 == slow2) {
1347          lcpLength++;
1348        }
1349        slow1 = slow1.next;
1350        slow2 = slow2.next;
1351      }
1352
1353      return lcpLength;
1354    }
1355
1356    // Get the length of the longest common suffix
1357    public int getLCSLength(CircularLinkedList otherList) {
1358      Node head1 = this.head;
1359      Node head2 = otherList.head;
1360
1361      if (head1 == null || head2 == null) {
1362        System.out.println("One or both lists are empty");
1363        return 0;
1364      }
1365
1366      Node slow1 = head1;
1367      Node slow2 = head2;
1368
1369      int lcsLength = 0;
1370
1371      do {
1372        if (slow1.next == head1) {
1373          slow1 = head1;
1374        }
1375        if (slow2.next == head2) {
1376          slow2 = head2;
1377        }
1378        if (slow1 == slow2) {
1379          lcsLength++;
1380        }
1381        slow1 = slow1.next;
1382        slow2 = slow2.next;
1383      }
1384
1385      return lcsLength;
1386    }
1387
1388    // Get the length of the longest common prefix
1389    public int getLCPLength(CircularLinkedList otherList) {
1390      Node head1 = this.head;
1391      Node head2 = otherList.head;
1392
1393      if (head1 == null || head2 == null) {
1394        System.out.println("One or both lists are empty");
1395        return 0;
1396      }
1397
1398      Node slow1 = head1;
1399      Node slow2 = head2;
1400
1401      int lcpLength = 0;
1402
1403      do {
1404        if (slow1.next == head1) {
1405          slow1 = head1;
1406        }
1407        if (slow2.next == head2) {
1408          slow2 = head2;
1409        }
1410        if (slow1 == slow2) {
1411          lcpLength++;
1412        }
1413        slow1 = slow1.next;
1414        slow2 = slow2.next;
1415      }
1416
1417      return lcpLength;
1418    }
1419
1420    // Get the length of the longest common subsequence
1421    public int getLCSLength(CircularLinkedList otherList) {
1422      Node head1 = this.head;
1423      Node head2 = otherList.head;
1424
1425      if (head1 == null || head2 == null) {
1426        System.out.println("One or both lists are empty");
1427        return 0;
1428      }
1429
1430      Node slow1 = head1;
1431      Node slow2 = head2;
1432
1433      int lcsLength = 0;
1434
1435      do {
1436        if (slow1.next == head1) {
1437          slow1 = head1;
1438        }
1439        if (slow2.next == head2) {
1440          slow2 = head2;
1441        }
1442        if (slow1 == slow2) {
1443          lcsLength++;
1444        }
1445        slow1 = slow1.next;
1446        slow2 = slow2.next;
1447      }
1448
1449      return lcsLength;
1450    }
1451
1452    // Get the length of the longest common prefix
1453    public int getLCPLength(CircularLinkedList otherList) {
1454      Node head1 = this.head;
1455      Node head2 = otherList.head;
1456
1457      if (head1 == null || head2 == null) {
1458        System.out.println("One or both lists are empty");
1459        return 0;
1460      }
1461
1462      Node slow1 = head1;
1463      Node slow2 = head2;
1464
1465      int lcpLength = 0;
1466
1467      do {
1468        if (slow1.next == head1) {
1469          slow1 = head1;
1470        }
1471        if (slow2.next == head2) {
1472          slow2 = head2;
1473        }
1474        if (slow1 == slow2) {
1475          lcpLength++;
1476        }
1477        slow1 = slow1.next;
1478        slow2 = slow2.next;
1479      }
1480
1481      return lcpLength;
1482    }
1483
1484    // Get the length of the longest common suffix
1485    public int getLCSLength(CircularLinkedList otherList) {
1486      Node head1 = this.head;
1487      Node head2 = otherList.head;
1488
1489      if (head1 == null || head2 == null) {
1490        System.out.println("One or both lists are empty");
1491        return 0;
1492      }
1493
1494      Node slow1 = head1;
1495      Node slow2 = head2;
1496
1497      int lcsLength = 0;
1498
1499      do {
1500        if (slow1.next == head1) {
1501          slow1 = head1;
1502        }
1503        if (slow2.next == head2) {
1504          slow2 = head2;
1505        }
1506        if (slow1 == slow2) {
1507          lcsLength++;
1508        }
1509        slow1 = slow1.next;
1510        slow2 = slow2.next;
1511      }
1512
1513      return lcsLength;
1514    }
1515
1516    // Get the length of the longest common prefix
1517    public int getLCPLength(CircularLinkedList otherList) {
1518      Node head1 = this.head;
1519      Node head2 = otherList.head;
1520
1521      if (head1 == null || head2 == null) {
1522        System.out.println("One or both lists are empty");
1523        return 0;
1524      }
1525
1526      Node slow1 = head1;
1527      Node slow2 = head2;
1528
1529      int lcpLength = 0;
1530
1531      do {
1532        if (slow1.next == head1) {
1533          slow1 = head1;
1534        }
1535        if (slow2.next == head2) {
1536          slow2 = head2;
1537        }
1538        if (slow1 == slow2) {
1539          lcpLength++;
1540        }
1541        slow1 = slow1.next;
1542        slow2 = slow2.next;
1543      }
1544
1545      return lcpLength;
1546    }
1547
1548    // Get the length of the longest common subsequence
1549    public int getLCSLength(CircularLinkedList otherList) {
1550      Node head1 = this.head;
1551      Node head2 = otherList.head;
1552
1553      if (head1 == null || head2 == null) {
1554        System.out.println("One or both lists are empty");
1555        return 0;
1556      }
1557
1558      Node slow1 = head1;
1559      Node slow2 = head2;
1560
1561      int lcsLength = 0;
1562
1563      do {
1564        if (slow1.next == head1) {
1565          slow1 = head1;
1566        }
1567        if (slow2.next == head2) {
1568          slow2 = head2;
1569        }
1570        if (slow1 == slow2) {
1571          lcsLength++;
1572        }
1573        slow1 = slow1.next;
1574        slow2 = slow2.next;
1575      }
1576
1577      return lcsLength;
1578    }
1579
1580    // Get the length of the longest common prefix
1581    public int getLCPLength(CircularLinkedList otherList) {
1582      Node head1 = this.head;
1583      Node head2 = otherList.head;
1584
1585      if (head1 == null || head2 == null) {
1586        System.out.println("One or both lists are empty");
1587        return 0;
1588      }
1589
1590      Node slow1 = head1;
1591      Node slow2 = head2;
1592
1593      int lcpLength = 0;
1594
1595      do {
1596        if (slow1.next == head1) {
1597          slow1 = head1;
1598        }
1599        if (slow2.next == head2) {
1600          slow2 = head2;
1601        }
1602        if (slow1 == slow2) {
1603          lcpLength++;
1604        }
1605        slow1 = slow1.next;
1606        slow2 = slow2.next;
1607      }
1608
1609      return lcpLength;
1610    }
1611
1612    // Get the length of the longest common suffix
1613    public int getLCSLength(CircularLinkedList otherList) {
1614      Node head1 = this.head;
1615      Node head2 = otherList.head;
1616
1617      if (head1 == null || head2 == null) {
1618        System.out.println("One or both lists are empty");
1619        return 0;
1620      }
1621
1622      Node slow1 = head1;
1623      Node slow2 = head2;
1624
1625      int lcsLength = 0;
1626
1627      do {
1628        if (slow1.next == head1) {
1629          slow1 = head1;
1630        }
1631        if (slow2.next == head2) {
1632          slow2 = head2;
1633        }
1634        if (slow1 == slow2) {
1635          lcsLength++;
1636        }
1637        slow1 = slow1.next;
1638        slow2 = slow2.next;
1639      }
1640
1641      return lcsLength;
1642    }
1643
1644    // Get the length of the longest common prefix
1645    public int getLCPLength(CircularLinkedList otherList) {
1646      Node head1 = this.head;
1647      Node head2 = otherList.head;
1648
1649      if (head1 == null || head2 == null) {
1650        System.out.println("One or both lists are empty");
1651        return 0;
1652      }
1653
1654      Node slow1 = head1;
1655      Node slow2 = head2;
1656
1657      int lcpLength = 0;
1658
1659      do {
1660        if (slow1.next == head1) {
1661          slow1 = head1;
1662        }
1663        if (slow2.next == head2) {
1664          slow2 = head2;
1665        }
1666        if (slow1 == slow2) {
1667          lcpLength++;
1668        }
1669        slow1 = slow1.next;
1670        slow2 = slow2.next;
1671      }
1672
1673      return lcpLength;
1674    }
1675
1676    // Get the length of the longest common subsequence
1677    public int getLCSLength(CircularLinkedList otherList) {
1678      Node head1 = this.head;
1679      Node head2 = otherList.head;
1680
1681      if (head1 == null || head2 == null) {
1682        System.out.println("One or both lists are empty");
1683        return 0;
1684      }
1685
1686      Node slow1 = head1;
1687      Node slow2 = head2;
1688
1689      int lcsLength = 0;
1690
1691      do {
1692        if (slow1.next == head1) {
1693          slow1 = head1;
1694        }
1695        if (slow2.next == head2) {
1696          slow2 = head2;
1697        }
1698        if (slow1 == slow2) {
1699          lcsLength++;
1700        }
1701        slow1 = slow1.next;
1702        slow2 = slow2.next;
1703      }
1704
1705      return lcsLength;
1706    }
1707
1708    // Get the length of the longest common prefix
1709    public int getLCPLength(CircularLinkedList otherList) {
1710      Node head1 = this.head;
1711      Node head2 = otherList.head;
1712
1713      if (head1 == null || head2 == null) {
1714        System.out.println("One or both lists are empty");
1715        return 0;
1716      }
1717
1718      Node slow1 = head1;
1719      Node slow2 = head2;
1720
1721      int lcpLength = 0;
1722
1723      do {
1724        if (slow1.next == head1) {
1725          slow1 = head1;
1726        }
1727        if (slow2.next == head2) {
1728          slow2 = head2;
1729        }
1730        if (slow1 == slow2) {
1731          lcpLength++;
1732        }
1733        slow1 = slow1.next;
1734        slow2 = slow2.next;
1735      }
1736
1737      return lcpLength;
1738    }
1739
1740    // Get the length of the longest common suffix
1741    public int getLCSLength(CircularLinkedList otherList) {
1742      Node head1 = this.head;
1743      Node
```

The screenshot shows a Java code editor interface with a dark theme. The main window displays the `CircularLinkedList.java` file. The code implements a circular linked list with methods for insertion after a key and displaying the list. A vertical scrollbar is visible on the right side of the code area. The status bar at the bottom shows "Ln 685, Col 1" and "Java: Ready".

```
J CircularLinkedList.java 1 ×
C > Users > Rakes > OneDrive > Documents > javadsa > J CircularLinkedList.java > ...
589 public class CircularLinkedList {
590     public void insertAfter(int key, int value) {
591         Node temp = head;
592         if (temp == null) {
593             System.out.println("List is empty");
594             return;
595         }
596         do {
597             if (temp.data == key) {
598                 temp.next = newNode;
599                 return;
600             }
601             temp = temp.next;
602         } while (temp != head);
603         System.out.println("Key not found!");
604     }
605     // Display list
606     public void display() {
607         if (head == null) {
608             System.out.println("List is empty");
609             return;
610         }
611         Node temp = head;
612         System.out.print("Circular Linked List: ");
613         do {
614             System.out.print(temp.data + " ");
615             temp = temp.next;
616         } while (temp != head);
617         System.out.println();
618     }
619 }
Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc | ×
Ln 685, Col 1 Spaces: 4 UTF-8 CRLF () Java ⌂ Go Live Windsurf: ⌂
```

The screenshot shows a Java code editor interface with a dark theme. The main window displays the `CircularLinkedList.java` file, which now includes a `main` method that creates an instance of `CircularLinkedList`, performs some insertions, and then calls the `display` method. A vertical scrollbar is visible on the right side of the code area. The status bar at the bottom shows "Ln 685, Col 1" and "Java: Ready".

```
J CircularLinkedList.java 1 ×
C > Users > Rakes > OneDrive > Documents > javadsa > J CircularLinkedList.java > ...
589 public class CircularLinkedList {
590     public void display() {
591         Node temp = head;
592         do {
593             System.out.print(temp.data + " ");
594             temp = temp.next;
595         } while (temp != head);
596         System.out.println();
597     }
598 }
599 Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc | ×
600 public static void main(String[] args) {
601     CircularLinkedList cll = new CircularLinkedList();
602     cll.insertAtStart(value: 30);
603     cll.insertAtStart(value: 20);
604     cll.insertAtEnd(value: 40);
605     cll.insertAfter(key: 20, value: 25);
606     cll.display();
607 }
608
609 }
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
Ln 685, Col 1 Spaces: 4 UTF-8 CRLF () Java ⌂ Go Live Windsurf: ⌂
```

OUTPUT:

A screenshot of a terminal window within a code editor interface. The terminal is titled "TERMINAL" and shows the following command-line session:

```
PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa\" ; if ($?) { javac CircularLinkedList.java } ; if ($?) { java CircularLinkedList }
Circular Linked List: 20 25 30 40
PS C:\Users\Rakes\OneDrive\Documents\javadsa>
```

The terminal window has a dark background with light-colored text. The status bar at the bottom indicates "Java: Ready".

TASK-7:

CODE:

```
1 - class CNodeDel:
2 -     def __init__(self, value):
3 -         self.data = value
4 -         self.next = None
5 - class CircularListDelete:
6 -     def __init__(self):
7 -         self.head = None
8 -     def append(self, value):
9 -         new_node = CNodeDel(value)
10 -        if self.head is None:
11 -            self.head = new_node
12 -            new_node.next = self.head
13 -            return
14 -        temp = self.head
15 -        while temp.next != self.head:
16 -            temp = temp.next
17 -        temp.next = new_node
18 -        new_node.next = self.head
19 -    def display(self):
20 -        if self.head is None:
21 -            print("List is empty")
22 -            return
23 -        print("Circular Linked List:", end=" ")
24 -        temp = self.head

25 -        while True:
26 -            print(temp.data, end=" -> ")
27 -            temp = temp.next
28 -            if temp == self.head:
29 -                break
30 -        print("(back to head)")
31 -    def delete_begin(self):
32 -        if self.head is None:
33 -            print("List is empty, nothing to delete.")
34 -            return
35 -        if self.head.next == self.head:
36 -            self.head = None
37 -            print("Deleted the only node in the list.")
38 -            return
39 -        last = self.head
40 -        while last.next != self.head:
41 -            last = last.next
42 -        self.head = self.head.next
43 -        last.next = self.head
44 -        print("Node deleted from beginning.")
45 -    def delete_end(self):
46 -        if self.head is None:
47 -            print("List is empty, nothing to delete.")
48 -            return
```

```

49     if self.head.next == self.head:
50         self.head = None
51         print("Deleted the only node in the list.")
52         return
53     prev = None
54     temp = self.head
55     while temp.next != self.head:
56         prev = temp
57         temp = temp.next
58     prev.next = self.head
59     print("Node deleted from end.")
60 def delete_at_pos(self, pos):
61     if self.head is None:
62         print("List is empty, nothing to delete.")
63         return
64     if pos == 1:
65         self.delete_begin()
66         return
67     prev = None
68     temp = self.head
69     count = 1
70     while count < pos and temp.next != self.head:
71         prev = temp
72         temp = temp.next

```

```

73             count += 1
74     if count != pos:
75         print("Position out of range.")
76         return
77     prev.next = temp.next
78     print(f"Node deleted from position {pos}.")
79 cl_del = CircularListDelete()
80 cl_del.append(10)
81 cl_del.append(20)
82 cl_del.append(30)
83 cl_del.append(40)
84 print("Initial List:")
85 cl_del.display()
86 cl_del.delete_begin()
87 cl_del.display()
88 cl_del.delete_end()
89 cl_del.display()
90 cl_del.delete_at_pos(2)
91 cl_del.display()

```

OUTPUT:

```
Initial List:  
Circular Linked List: 10 -> 20 -> 30 -> 40 -> (back to head)  
Node deleted from beginning.  
Circular Linked List: 20 -> 30 -> 40 -> (back to head)  
Node deleted from end.  
Circular Linked List: 20 -> 30 -> (back to head)  
Node deleted from position 2.  
Circular Linked List: 20 -> (back to head)  
  
==== Code Execution Successful ===
```

TASK 8: Circular queue insertion and deletion

CODE:

```
File Edit Selection View ... < > Search
J CircularQueue.java 1 ×
C > Users > Rakes > OneDrive > Documents > javadsa > J CircularQueue.java > CircularQueue.enqueue(int)
690 public class CircularQueue {
691     // ...
692     // Windsurf: Refactor | Explain | X
693     public void enqueue() {
694         if (front == -1) {
695             System.out.println("Queue Underflow! Nothing to delete.");
696             return;
697         }
698
699         System.out.println("Deleted: " + queue[front]);
700
701         if (front == rear) {
702             // Only one element was present
703             front = rear = -1;
704         } else {
705             front = (front + 1) % size;
706         }
707     }
708
709     // Display queue
710     // Windsurf: Refactor | Explain | X
711     public void display() {
712         if (front == -1) {
713             System.out.println("Queue is empty.");
714             return;
715         }
716
717         System.out.print("Circular Queue: ");
718         int i = front;
719         while (true) {
720             System.out.print(queue[i] + " ");
721             if (i == rear) break;
722         }
723     }
724
725     // ...
726 }
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
```

A screenshot of a Java code editor showing the `main` method of the `CircularQueue` class. The code is as follows:

```
690 public class CircularQueue {  
752     // MAIN  
753     Run | Debug | Windsurf Refactor | Explain | X  
754     public static void main(String[] args) {  
755         Scanner sc = new Scanner(System.in);  
756         System.out.print(s: "Enter queue size: ");  
757         int n = sc.nextInt();  
758         CircularQueue cq = new CircularQueue(n);  
759  
760         while (true) {  
761             System.out.println(x: "\n--- Circular Queue Menu ---");  
762             System.out.println(x: "1. Enqueue");  
763             System.out.println(x: "2. Dequeue");  
764             System.out.println(x: "3. Display");  
765             System.out.println(x: "4. Exit");  
766             System.out.print(s: "Enter choice: ");  
767  
768             int choice = sc.nextInt();  
769  
770             switch (choice) {  
771                 case 1:  
772                     System.out.print(s: "Enter value to insert: ");  
773                     int val = sc.nextInt();  
774                     cq.enqueue(val);  
775                     break;  
776  
777                 case 2:  
778                     cq.dequeue();  
779                     break;  
780  
781                 case 3:  
782                     cq.display();  
783                     break;  
784  
785                 case 4:  
786                     System.out.println(x: "Exiting...");  
787                     sc.close();  
788                     return;  
789  
790                 default:  
791                     System.out.println(x: "Invalid choice!");  
792             }  
793         }  
794     }  
795 }  
796 }  
797 }  
798 }
```

The code editor interface includes a search bar, toolbars, and a status bar at the bottom indicating the current line (Ln 708), column (Col 27), and file type (Java). A vertical sidebar on the right shows a tree view of the project structure.

A screenshot of a Java code editor showing the `main` method of the `CircularQueue` class. The code has been modified to include a `default` case in the switch statement:

```
690 public class CircularQueue {  
753     public static void main(String[] args) {  
754         //  
755         switch (choice) {  
756             case 1:  
757                 System.out.print(s: "Enter value to insert: ");  
758                 int val = sc.nextInt();  
759                 cq.enqueue(val);  
760                 break;  
761  
762             case 2:  
763                 cq.dequeue();  
764                 break;  
765  
766             case 3:  
767                 cq.display();  
768                 break;  
769  
770             case 4:  
771                 System.out.println(x: "Exiting...");  
772                 sc.close();  
773                 return;  
774  
775             default:  
776                 System.out.println(x: "Invalid choice!");  
777         }  
778     }  
779 }  
780 }  
781 }
```

The code editor interface is identical to the first screenshot, including the search bar, toolbars, and project sidebar.

OUTPUT:

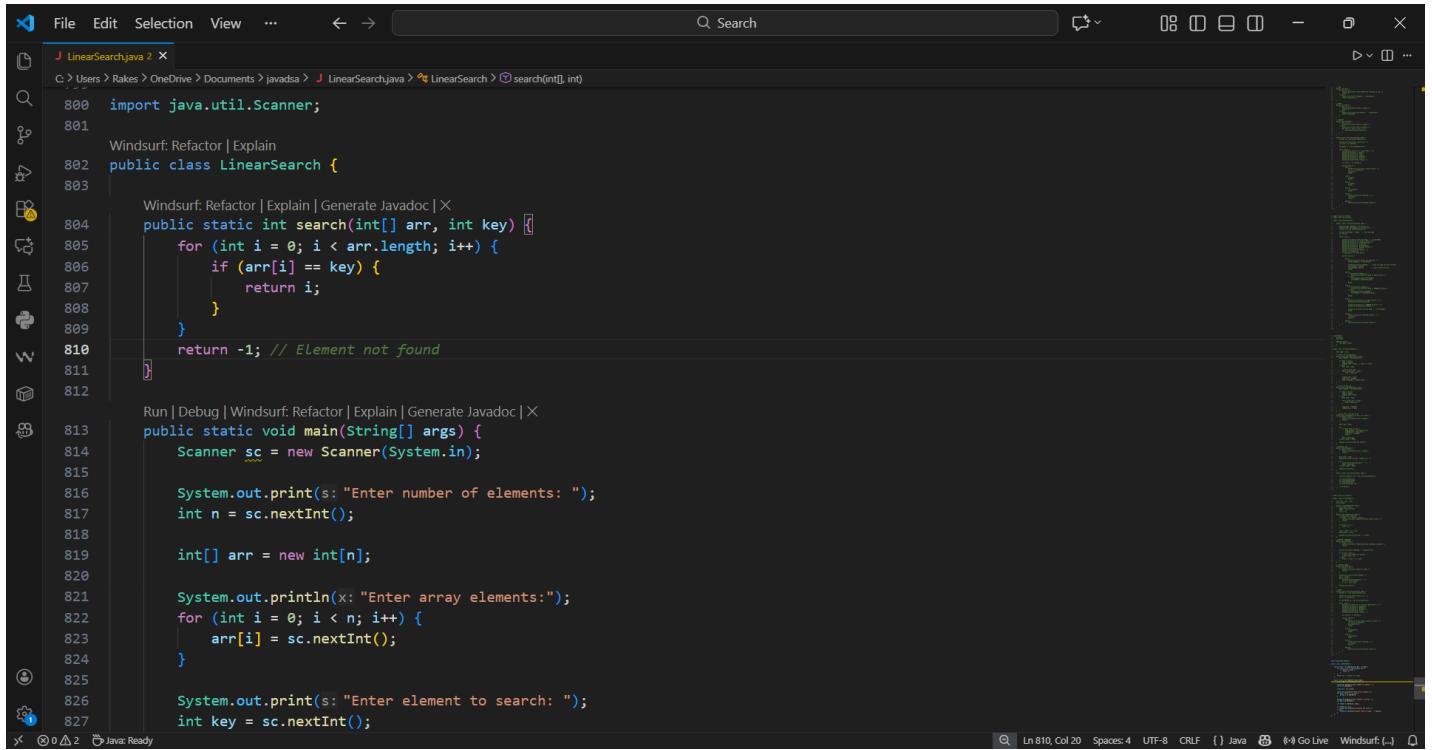
```
PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa\" ; if ($?) { javac CircularQueue.java } ; if ($?) { java CircularQueue }
Enter queue size: 2
--- Circular Queue Menu ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter choice: 1
Enter value to insert: 33
Inserted: 33
--- Circular Queue Menu ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter choice: 2
Deleted: 33
--- Circular Queue Menu ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter choice: 3
Queue is empty.
--- Circular Queue Menu ---
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter choice: 4
Exiting...
PS C:\Users\Rakes\OneDrive\Documents\javadsa>
```

The screenshot shows a terminal window within a code editor interface. The terminal output displays the execution of a Java program for a Circular Queue. The user enters a queue size of 2. They then choose to enqueue the value 33, which is successfully inserted. Subsequent choices to dequeue and display the queue result in an empty queue message. Finally, the user exits the program. The terminal window has a dark theme and includes standard Windows-style icons for file operations like copy, paste, and search.

TASK-9:

LINEAR SEARCH

CODE:



The screenshot shows a Java code editor window with the following code:

```
File Edit Selection View ... ← → Q Search X
LinearSearch.java 2 ×
C:\Users\Rakesh\OneDrive\Documents\javadsa\J\LinearSearch> search(int[], int)
800 import java.util.Scanner;
801
802 Windsurf: Refactor | Explain
803 public class LinearSearch {
804     Windsurf: Refactor | Explain | Generate Javadoc ×
805     public static int search(int[] arr, int key) {
806         for (int i = 0; i < arr.length; i++) {
807             if (arr[i] == key) {
808                 return i;
809             }
810         }
811         return -1; // Element not found
812     }
813     Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc ×
814     public static void main(String[] args) {
815         Scanner sc = new Scanner(System.in);
816
817         System.out.print("Enter number of elements: ");
818         int n = sc.nextInt();
819
820         int[] arr = new int[n];
821
822         System.out.println("Enter array elements:");
823         for (int i = 0; i < n; i++) {
824             arr[i] = sc.nextInt();
825         }
826
827         System.out.print("Enter element to search: ");
828         int key = sc.nextInt();
}
Ln 810, Col 20 Spaces: 4 CRLF {} Java Go Live Windsurf: ...
```

The code implements a linear search algorithm. It defines a class named `LinearSearch` with a static method `search` that takes an integer array and a key as parameters and returns the index of the key if found, or -1 if not found. The main method prompts the user for the number of elements, creates an array, and then reads each element from the user. Finally, it prompts the user for a key to search for.

OUTPUT:

TASK-10:

BINARY SEARCH:

CODE:

```
1 import random
2
3 values = sorted(random.randint(1, 100) for _ in range(10))
4 print("Array:", values)
5
6 key = int(input("Enter element to search (binary): "))
7
8 low = 0
9 high = len(values) - 1
10 found_flag = False
11
12 while low <= high:
13     mid = (low + high) // 2
14     if values[mid] == key:
15         print("Element found at index", mid)
16         found_flag = True
17         break
18     elif values[mid] < key:
19         low = mid + 1
20     else:
21         high = mid - 1
22
23 if not found_flag:
24     print("Element not found")
```

OUTPUT:

The screenshot shows a terminal window within a code editor interface. The terminal bar at the top has tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The TERMINAL tab is active. The output pane displays the following text:

```
PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa" ; if ($?) { javac LinearSearch.java } ; if ($?) { java LinearSearch }
Enter number of elements:
1 2 3 4 5
Enter element to search: 4
Element found at index: 3
PS C:\Users\Rakes\OneDrive\Documents\javadsa>
```

The terminal window includes standard Windows-style icons for file operations (New, Open, Save, Print, Find, Copy, Paste, Cut, Delete) and a status bar at the bottom showing file paths, line numbers, and character counts.

TASK 11:BUBBLE SORT

CODE:

The screenshot shows a Java code editor with the file `BubbleSort.java` open. The code implements the Bubble Sort algorithm. It includes a `bubbleSort` method that takes an integer array as input and sorts it in ascending order. The main method prompts the user to enter the number of elements and the array elements, then calls the `bubbleSort` method and prints the sorted array. The code is annotated with line numbers from 841 to 868.

```
File Edit Selection View ... ← → Search
J BubbleSort.java ×
C > Users > Rakes > OneDrive > Documents > javadsa > J BubbleSort.java > bubbleSort(int[])
841
842 import java.util.Scanner;
843
844 public class BubbleSort {
845
846     public static void bubbleSort(int[] arr) {
847         int n = arr.length;
848
849         for (int i = 0; i < n - 1; i++) {
850             for (int j = 0; j < n - i - 1; j++) {
851                 if (arr[j] > arr[j + 1]) {
852                     int temp = arr[j];
853                     arr[j] = arr[j + 1];
854                     arr[j + 1] = temp;
855                 }
856             }
857         }
858     }
859 }
860
Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc | X
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of elements: ");
    int n = sc.nextInt();

    int[] arr = new int[n];
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881 }
```

The screenshot shows a Java code editor with the file `BubbleSort.java` open. This version of the code includes additional logic to print the elements of the array before and after sorting. It uses a `for` loop to read array elements from the user and another `for` loop to print the sorted array. The code is annotated with line numbers from 844 to 884.

```
File Edit Selection View ... ← → Search
J BubbleSort.java ×
C > Users > Rakes > OneDrive > Documents > javadsa > J BubbleSort.java > bubbleSort(int[])
844 public class BubbleSort {
845     public static void bubbleSort(int[] arr) {
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
Run | Debug | Windsurf: Refactor | Explain | Generate Javadoc | X
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of elements: ");
    int n = sc.nextInt();

    int[] arr = new int[n];
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884 }
```

OUTPUT:

The screenshot shows a terminal window within a code editor interface. The terminal tab is selected, displaying the following command-line session:

```
PS C:\Users\Rakes> cd "c:\Users\Rakes\OneDrive\Documents\javadsa" ; if ($?) { javac BubbleSort.java } ; if ($?) { java BubbleSort }
Enter number of elements:
1 2 3 4 5
Sorted Array (Bubble Sort):
1 2 3 4 5
PS C:\Users\Rakes\OneDrive\Documents\javadsa>
```

The terminal output shows the execution of a Java program named `BubbleSort`. It first asks for the number of elements, receives input `1 2 3 4 5`, and then displays the sorted array `1 2 3 4 5`.

At the bottom of the terminal window, there is a status bar with the following information:

Ln 852, Col 21 Spaces: 4 UTF-8 CRLF {} Java Go Live Windsurf: (...) □