

```

1  import java.io.Serializable; //needed for Serializable interface
2  /**this class creates the CustomerFile objects which will store the customer information for each
3   * Respective customer. There are "get" and "set" methods to initialise values to each respective
4   * Instance variable and the constructor will use the methods to initialise the variables
5   *
6   * @author Luke Geeson
7   * @version 1.00
8   * @date 20/11/12
9   * @school Norton Knatchbull School
10  */
11  public class CustomerFile implements Serializable //can be serialized and stored in a Serializable file
12  {
13      /**
14       * This is the constructor, it will initialise values to the instance variables that create the
15       * Customer record using the get and set methods of this class. It does not return anything
16       * So it is void but it will output a String ensuring the user that a record has been created
17       *
18       * @param fName      this is the forename string which will be assigned to forename
19       * @param sName      this is the surname string which will be assigned to surname
20       * @param pNum       this is the phone number which will be assigned to phoneNumber
21       * @param eAddress   this is the email address which will be assigned to eAddress
22       * @param hAddress   this is the home address which will be assigned to homeAddress
23       * @param pCode     this is the post code which will be assigned to postCode
24       */
25      public CustomerFile(String fName, String sName, String pNum, String eAddress, String hAddress, String pCode)
26      {
27          setForename(fName);
28          setSurname(sName);
29          setPhoneNumber(pNum);
30          setEmailAddress(eAddress);
31          setHomeAddress(hAddress);
32          setPostCode(pCode);
33          System.out.println("Details successfully input");
34      }
35      /**
36       * This is the default constructor
37       */
38      public CustomerFile(){}

```

```
39  //-----
40  /**
41   * This is the method that will initialise the variable "forename" - method is void, it doesn't return
42   * anything
43   *
44   * @param a          this is the string that will be assigned to String forename
45   */
46  public void setForename(String inputForename)
47  {
48      this.forename = convertAndPresent(inputForename);
49  }
50  //-----
51  /**
52   * This is the method that will return the variable "forename"
53   *
54   * @return forename    this is the variable that stores the forename
55   */
56  public String getForename()
57  {
58      return this.forename;
59  }
60  //-----
61  /**
62   * This method initialises the variable "surname" with the input string - method is void, it doesn't
63   * return anything
64   *
65   * @param a          this String will be used to initialise the "surname" variable
66   */
67  public void setSurname(String inputSurname)
68  {
69      this.surname = convertAndPresent(inputSurname);
70  }
71  //-----
72  /**
73   * This method will return the String variable "surname"
74   *
75   * @return surname    this is the variable that stores the surname
76   */
```

```
77     public String getSurname()
78     {
79         return this.surname;
80     }
81     //-----
82     /**
83      * This method initialises the String variable "phoneNumber" - method is void
84      *
85      * @param a          this string will be used to initialise the "phoneNumber" variable
86      */
87     public void setPhoneNumber(String inputPNum)
88     {
89         this.phoneNumber = inputPNum.trim();
90     }
91     //-----
92     /**
93      * This method returns the String variable "phoneNumber"
94      *
95      * @return phoneNumber the variable that stores the phone number
96      */
97     public String getPhoneNumber()
98     {
99         return this.phoneNumber;
100    }
101    //-----
102    /**
103     * This method initialises the variable "emailAddress" - method is void
104     *
105     * @param a          the String used to initialise the variable "emailAddress"
106     */
107    public void setEmailAddress(String inputEAddress)
108    {
109        inputEAddress = inputEAddress.trim();
110        this.emailAddress = inputEAddress;
111    }
112    //-----
113    /**
114     * This method returns the variable "emailAddress"
```

```
115     *
116     * @return emailAddress the variable that stores the email address
117     */
118     public String getEmailAddress()
119     {
120         return this.emailAddress;
121     }
122     //-----
123     /**
124     * This method initialises the variable "homeAddress" - method is void
125     *
126     * @param a          the string used to initialise the variable "homeAddress"
127     */
128     public void setHomeAddress(String inputHAddress)
129     {
130         inputHAddress = inputHAddress.trim();
131         this.homeAddress = inputHAddress;
132     }
133     //-----
134     /**
135     * This method returns the variable "homeAddress"
136     *
137     * @return homeAddress the String used to store the home address
138     */
139     public String getHomeAddress()
140     {
141         return this.homeAddress;
142     }
143     //-----
144     /**
145     * This method initialises the variable "postCode" - method is void
146     *
147     * @param a          the String used to initialise the variable "postCode"
148     */
149     public void setPostCode(String inputPCode)
150     {
151         inputPCode = inputPCode.trim();
152         inputPCode = inputPCode.toUpperCase();
```

```

153         this.postCode = inputPCode;
154     }
155     //-----
156     /**
157     * This method returns the variable "postCode"
158     *
159     * @return postCode    the string used to store the post code
160     */
161     public String getPostCode()
162     {
163         return this.postCode;
164     }
165     //-----
166     /**
167     * This method converts Strings input so that the first letter is changed to uppercase to be used with proper nouns
168     * such as names and days of the year - in this method it changes the String input (the forename and Surname for each
169     file)
170     *
171     * @param toConvert    the string which will be converted to a proper noun e.g input: "luke" output: "Luke"
172     *
173     * @return converted    returns the converted String with an appropriate capital letter
174     */
175     public static String convertAndPresent(String toConvert)
176     {
177         toConvert = toConvert.trim();
178         if (toConvert.equals(""))
179         {
180             toConvert = "empty field";
181         }
182         char firstLetter = toConvert.charAt(0);           //takes the first letter
183         String converted = String.valueOf(firstLetter).toUpperCase() + toConvert.substring(1,toConvert.length()); //returns
184         the letter (in upper case)
185         return converted;                                   //returns this string
186     }
187     //-----
188     /**
189     * Method used to display all the details of a CustomerFile - method is void as it is printing and
190     * takes no parameters

```

```

191     */
192     public void displayAllDetails()
193     {
194         System.out.println("Customer name:\t " + getForename() + " " + getSurname());           //prints customer name
195         System.out.println("Phone number:\t " + getPhoneNumber());                             //prints phone number
196         System.out.println("Email address\t " + getEmailAddress());                         //prints email address
197         System.out.println("Home address:\t " + getHomeAddress());                         //prints home address
198         System.out.println("Post code:\t " + getPostCode());                             //prints post code
199         System.out.print("\n");                                                         //prints gap between text on screen
200     }
201     //-----
202     //state variables used in this class - private for data protection
203     private String forename;                 //declares the forename of this customer as a string
204     private String surname;                 //declares the surname of the customer as a string
205     private String phoneNumber;             //declares the phone number of the customer as a string
206     private String emailAddress;            //declares the email address of the customer as a string
207     private String homeAddress;             //declares the home address of the customer as a string
208     private String postCode;               //declares the post code of the customer as a string
209 }

210                                     //-----new class-----\\

211 import java.io.Serializable;           //needed for the Serializable interface
212 /**
213  * This class creates the nodes used in the singly linked list of customer records. This method contains
214  * the "get" and "set" methods for the node data and nextNode variables
215  *
216  * @author Luke Geeson
217  * @version 1.00
218  * @date 25/11/12
219  * @school Norton Knatchbull School
220  */
221 public class Node implements Serializable //used so it can be serialized
222 {
223     /**
224      * This is the constructor; it will initialise the variables of the Node object
225      *
226      * @param data      this is the CustomerFile object that will make up the data of the node

```

```
227     * @param next      this is the next Object in the list
228     */
229     public Node(CustomerFile data, Node next)
230     {
231         nextNode = next;
232         nodeData = data;
233     }
234     public Node(){}
235     //-----
236     /**
237      * This method initialises the variable "nodeData" with a CustomerFile object as the data - method is void
238      *
239      * @param input      this is the Object object which will be assigned to the Object object "nodeData"
240      */
241     public void setData(CustomerFile input)
242     {
243         this.nodeData = input;
244     }
245     //-----
246     /**
247      * This method returns the CustomerFile object data assigned to "nodeData"
248      *
249      * @return nodeData this is the CustomerFile object which contains the data for a customer file
250      */
251     public CustomerFile getData()
252     {
253         return this.nodeData;
254     }
255     //-----
256     /**
257      * This method initialises the Node object "nextNode" with the Node input - method is void
258      *
259      * @param inputNext the Node which will initialise the variable "nextNode"
260      */
261     public void setNext(Node inputNext)
262     {
263         this.nextNode = inputNext;
264     }
```

```

265 //-----
266 /**
267  * This method returns the Node stored in the variable "nextNode" which is the next node in the linked list
268  *
269  * @return nextNode the Node which contains the next item in the linked list
270  */
271 public Node getNext()
272 {
273     return this.nextNode;
274 }
275 //-----
276 //state variables used in this class - private for data protection
277 private CustomerFile nodeData;      //contains the customer record of this node
278 private Node nextNode;              //contains a reference to the next node
279 }

```

//-----new class-----\\

```

281 import java.io.Serializable;          //needed for the Serializable interface
282 /**
283  * This is the class that is used to make a singly linked list with Node objects as the elements. There
284  * are methods to add, remove, modify, search, find the size, determine if it is empty, sort, merge sort
285  * set/get the head node (add first), set/get the last node (add last), print items in ascending/descending
286  * order and a method to convert the linked list into a node array
287  *
288  * @author Luke Geeson
289  * @version 1.00
290  * @date 29/11/12
291  * @school Norton Knatchbull School
292  */
293 public class SinglyLinkedList implements Serializable
294 {
295     /**
296     * this constructor is used when a new linked list is created and you want to pass the first node
297     * of the list to it. If there is only one Node passed then pass null as the nextNode parameter
298     *
299     * @param firstNode    the first node in the linked list, set as the head of the list
300     * @param nextNode     the next node after the first, pass as null if there are no other nodes

```



```
301     */
302     public SinglyLinkedList (Node firstNode, Node nextNode)
303     {
304         setHead(firstNode);
305         firstNode.setNext(nextNode);
306     }
307     /**
308     * This is the default constructor
309     */
310     public SinglyLinkedList()
311     {
312         this.head = null;
313     }
314     //-----
315     /**
316     * returns a Boolean value dependent on whether the Linked List is empty - no parameters
317     *
318     * @return empty          will determine whether it is empty - 'true' = the list is empty
319     */
320     public boolean isEmpty()
321     {
322         boolean empty;
323         if (this.head == null)
324         {
325             empty = true;           //if there is not a head node i.e. the list is empty
326         }
327         else
328         {
329             empty = false;        //if there is a head node i.e. the list is full
330         }
331         return empty;
332     }
333     //-----
334     /**
335     * returns the size of the list
336     *
337     * @param input          calculates the size of the list with this variable - pass head node
338     *
```

```
339     * @return int           the size of the list (null if empty)
340     */
341     public static int length(Node input)
342     {
343         if (input == null)
344         {
345             return 0;                //if the node is empty, if the head node and the list is empty
346         }
347         else
348         {
349             return 1 + length(input.getNext());    //recursive length check
350         }
351     }
352     //-----
353     /**
354     * Inserts a Node in the list, in order according to the Surname is void as it does not return anything
355     *
356     * @param newData          the new Node which contains the new data to be inserted
357     * @param trailNode        the node which the new data will compare with
358     */
359     public void addRecord(Node newData, Node trailNode)
360     {
361         if(trailNode == null)        //if the trailNode is null
362         {
363             if (this.head == null)    //if the linked list has no head i.e. it is empty
364             {
365                 this.setHead(newData);    //sets the new data as the head of the list
366             }
367             else
368             {
369                 //if the list is at its end
370                 trailNode = newData;    //sets the newData as the last item
371                 newData.setNext(null);    //sets the next item as null if the node is the first or last item
372             }
373         }
374         else if (trailNode == getHead() && newData.getData().getSurname().compareTo(trailNode.getData().getSurname())<0)
375         {
376             //if the trail node = head node and the new data comes before it
377             newData.setNext(getHead());    //sets the newdata.next to the current head
378             setHead(newData);    //sets the newdata as the head of the list
379         }
380     }
```

```

377     }
378     else if (trailNode.getNext() == null)           //if the trail.next is empty
379     {
380         trailNode.setNext(newData);                 //set the next item in the list
381         newData.setNext(null);                      //set the next item in the list as null
382     }
383     else if (newData.getData().getSurname().equalsIgnoreCase(trailNode.getData().getSurname()))
384     {                                                //if the surnames are identical - store new one after old
385         newData.setNext(trailNode.getNext());       //set new data.next to trail node.next
386         trailNode.setNext(newData);                 //set trailnode.next as new data
387     }
388     else if (newData.getData().getSurname().compareTo(trailNode.getNext().getData().getSurname())<0)
389     {                                                //if the new item comes before the trail.next
390         if (newData.getData().getSurname().compareTo(trailNode.getData().getSurname())>0)
391         {                                            //if the new item comes after the trail node
392             newData.setNext(trailNode.getNext());  //sets the new data as the node before the item after trail node
393             trailNode.setNext(newData);             //sets the new data as the item after the trail node
394         }
395     }
396     else
397     {                                                //if the new data is not within the confines of the 2 nodes
398         addRecord(newData, trailNode.getNext());    //recursive traversal through the list to the correct point
399     }
400 }
401 //-----
402 /**
403  * This method searches for the first occurrence of the Node with an identical key String to "search"
404  *
405  * @param compNode    the node which will be compared with the search string
406  * @param search      the string which that is sought after
407  *
408  * @return compNode or null this will return the first occurrence of the Node containing the search String null if no such
409  node is found
410  */
411 public Node searchList(Node compNode, String search)
412 {
413     if (compNode == null)
414     {

```

```

415         return null;                                //if the list is empty
416     }
417     else if (compNode.getData().getSurname().equalsIgnoreCase(search))
418     {
419         return compNode;                                //if the node has the search string
420     }
421     else
422     {
423         return searchList(compNode.getNext(),search);    //recursive traversal through the list
424     }
425 }
426 //-----
427 /**
428  * This method will take a Node from the parameter and remove the first occurrence of it in the singly
429  * Linked List. It is void and takes 2 parameters:
430  *
431  * @param toRemove      the node which needs to be removed
432  * @param compNode      the node with which the 'toRemove' node will be compared - allows recursive traversal
433  */
434 public void removeNode(Node toRemove, Node compNode)
435 {
436     if (isEmpty() == true)
437     {
438         System.out.println("The list is empty - cannot remove this item as it does not exist");
439     }
440     else if (toRemove == getHead())
441     {
442         //if the head of the list is the node to remove
443         if (getHead() != null && getHead().getNext() == null)
444         {
445             //if the head is not null but the next node is
446             //removes the node from the list
447             this.head = null;
448         }
449         else if (getHead().getNext() != null)
450         {
451             //if the next item is not null and the head is to be removed
452             //marks the head to be removed
453             toRemove = getHead();
454             this.head = getHead().getNext();
455             //the head is set as the next node
456             toRemove = null;
457             //the old head is deleted
458         }
459     }
460 }

```

```

453     else if (compNode.getNext() == toRemove && compNode.getNext() == getLast(getHead()))
454     {                                                                 //if the node to remove is at the end of the list
455         compNode.setNext(null);                                     //removes the last item in the list
456     }
457     else if (toRemove == compNode.getNext() )
458     {                                                                 //if the next item in the list is the item to remove
459         toRemove = compNode.getNext();                             //marks the next item to be removed
460         Node remPrev = compNode;                                   //gets the previous node for reallocation
461         Node remAfter = toRemove.getNext();                       //gets the next node for reallocation
462         remPrev.setNext(remAfter);                                 //reallocates other nodes
463         toRemove = null;                                           //removes old node
464     }
465     else
466     {                                                                 //recursive traversal through list until statement is satisfied
467         removeNode(toRemove, compNode.getNext());
468     }
469 }
470 //-----
471 /**
472  * This method will take a node and change the specific details of it based on the data supplied
473  * NOTE: this method alters DATA of the node - use get/set data methods to alter the node itself
474  *
475  * @param nodeToChange      the node which will be altered
476  * @param changeDecision    the decision of which variable is to be altered
477  * @param changeInput       the new information that will replace the old
478  *
479  * @return nodeToChange     the old node is return with the changes
480  */
481 public static Node changeNode(Node nodeToChange, String changeDecision, String changeInput)
482 {
483     if (changeDecision.equalsIgnoreCase("FORENAME") || changeDecision.equalsIgnoreCase("FIRSTNAME") ||
484     changeDecision.equalsIgnoreCase("FIRST NAME") || changeDecision.equalsIgnoreCase("FIRST") ||
485     changeDecision.equalsIgnoreCase("F"))
486     {
487         nodeToChange.getData().setForename(changeInput); //if the forename is to be changed
488     } //change the forename of the node
489     else if (changeDecision.equalsIgnoreCase("SURNAME") || changeDecision.equalsIgnoreCase("SECONDNAME") ||
490     changeDecision.equalsIgnoreCase("SECOND NAME") || changeDecision.equalsIgnoreCase("S"))

```

```

491         {                                     //if the surname is to be changed
492             nodeToChange.getData().setSurname(changeInput); //change the surname
493         }
494         else if (changeDecision.equalsIgnoreCase("PHONENUMBER") || changeDecision.equalsIgnoreCase("PHONE NUMBER") ||
495 changeDecision.equalsIgnoreCase("PHONE"))
496         {                                     //if the phone number is to be changed
497             nodeToChange.getData().setPhoneNumber(changeInput); //change it
498         }
499         else if (changeDecision.equalsIgnoreCase("EMAILADDRESS") || changeDecision.equalsIgnoreCase("EMAIL ADDRESS") ||
500 changeDecision.equalsIgnoreCase("EMAIL") || changeDecision.equalsIgnoreCase("E"))
501         {                                     //if the email address is to be changed
502             nodeToChange.getData().setEmailAddress(changeInput); //change it
503         }
504         else if (changeDecision.equalsIgnoreCase("HOMEADDRESS") || changeDecision.equalsIgnoreCase("HOME ADDRESS") ||
505 changeDecision.equalsIgnoreCase("HOME") || changeDecision.equalsIgnoreCase("H"))
506         {                                     //if the home address is to be changed
507             nodeToChange.getData().setHomeAddress(changeInput); //change it
508         }
509         else if (changeDecision.equalsIgnoreCase("POSTCODE") || changeDecision.equalsIgnoreCase ("POST CODE"))
510         {                                     //if the postcode is to be changed
511             nodeToChange.getData().setPostCode(changeInput); //change it
512         }
513         return nodeToChange;                //return the modified node
514     }
515     //-----
516     /**
517     * This method performs a bubble sort on the linked list and sorts the data in ascending order
518     * performs a sort on the list associated and returns nothing
519     */
520     public void sortList()
521     {
522         if (isEmpty())                        //if the list is empty
523         {
524             System.out.println("cannot sort list - list is empty\n");
525         }
526         else if (getHead().getNext() == null) //if the list has one item only
527         {
528             System.out.println("List sorted\n");

```

```

529     }
530     else
531     {
532         Node current = getHead();                //starts sort from the start of the list
533         boolean swapDone = true;                 //used to iterate the loop
534         while (swapDone == true)
535         {
536             swapDone = false;                     //used to escape the loop
537             while (current != null)               //while current is not equal to the last item
538             {
539                 if (current.getNext() != null &&
540 current.getData().getSurname().compareTo(current.getNext().getData().getSurname()) > 0)
541                 {                                //if the next item comes before the current
542                     CustomerFile tempDat = current.getData();
543                     current.setData(current.getNext().getData());
544                     current.getNext().setData(tempDat); //swap the data
545                     swapDone = true;                 //used to continue the loop
546                 }
547                 current = current.getNext();        //traversal through the loop
548             }
549             current = getHead();                    //continue at start of the loop
550         }
551         System.out.println("List sorted\n");
552     }
553 }
554 //-----
555 /**
556  * This method will take two singly linked list objects as parameters and merge them - it has 4
557  * possible ways of working: 1.if both are empty, 2.one is full and the other is not, 3. vice versa
558  * or 4.both are full.
559  *
560  * @param lst1          the first list
561  * @param lst2          the second list
562  *
563  * @return newList      returns a new list with the 2 lists merged
564  */
565 public static SinglyLinkedList mergeLists(SinglyLinkedList lst1, SinglyLinkedList lst2)
566 {

```

```

567     SinglyLinkedList newList = new SinglyLinkedList(); //new list for merging
568     if (lst1 == null && lst2 == null) //if both lists are empty
569     {
570         newList = null; //cannot merge what is not there
571         System.out.println("Both lists are empty - cannot merge");
572     }
573     else if (lst1 == null && lst2 != null) //if list 1 is empty but the other is not
574     {
575         newList = lst2; //set the new list as list 2
576     }
577     else if (lst1 != null && lst2 == null) //if list 2 is empty but the other is not
578     {
579         newList = lst1; //set the new list as list 1
580     }
581     else //if both lists have data
582     {
583         newList = lst1; //set the new list as list 1 and add all records from the second
584         Node current = lst2.getHead(); //start point for adding to the new list from list 2
585         for (int j = 0; j < lst2.length(lst2.getHead()); j++)
586         {
587             newList.addRecord(current, newList.getHead()); //add record method inserts in order
588             current = current.getNext(); //iterative approach to merging the lists - allows traversal
589         }
590     }
591     return newList; //returns the new list
592 }
593 //-----
594 /**
595  * prints the linked list in descending order to the node which is input
596  *
597  * @param input prints all items from this point onward in descending order
598  */
599 public void printReverseList(Node input)
600 {
601     if (input != null)
602     {
603         printReverseList(input.getNext()); //recursive progression to the end of the list
604         input.getData().displayAllDetails(); //prints the information of the node

```



```
605     }
606 }
607 //-----
608 /**
609  * Prints the entire linked list in ascending order from the node input - will print from any node
610  * in the list to the end of the list but to print the whole list - pass the head node
611  *
612  * @param input          the node from which you print
613  */
614 public void printList(Node input)
615 {
616     if(input != null)
617     {
618         input.getData().displayAllDetails();           //prints all data from the node passed
619         printList(input.getNext());                     //recursive print
620     }
621 }
622 //-----
623 /**
624  * assigns the value input to the "head" variable and sets as the head of the list
625  *
626  * @param input          a Node which will be set as the head
627  */
628 public void setHead(Node input)
629 {
630     input.setNext(this.head);                           //sets the head as the node after the input node
631     head = input;                                         //sets the node input as the head node
632 }
633 //-----
634 /**
635  * Returns the value assigned to variable "head" effectively setting the head
636  * and adding a node to the list
637  *
638  * @return head          the head of the list
639  */
640 public Node getHead()
641 {
642     return this.head;
```

```
643 }
644 //-----
645 /**
646  * adds a node to the end of the list
647  *
648  * @param input      a node which will be set as the last item in the list
649  * @param elementOfList  a node which will be used to cycle through the list and append the input node to the end
650  */
651 public void appendLast(Node elementOfList, Node input)
652 {
653     if (elementOfList == null)                //if the node is empty
654     {
655         if (isEmpty() == true)                //if the list is empty
656         {
657             setHead(input);                    //sets the item as the new head
658             input.setNext(null);                //sets the next as null - it is the only item in the list
659         }
660         else                                    //if the list is not empty
661         {
662             elementOfList = input;                //sets the last item in the list as the input node
663             elementOfList.setNext(null);            //creates an end point to the list
664         }
665     }
666     else if (elementOfList.getNext() == null) //if the next item in the list is null
667     {
668         elementOfList.setNext(input);            //set the last item as the new node
669         input.setNext(null);                    //the new node is now the last item
670     }
671     else                                        //recursive traversal through the list if these are not satisfied
672     {
673         appendLast(elementOfList.getNext(), input);
674     }
675 }
676 //-----
677 /**
678  * Returns the value assigned to the last item in the list
679  *
680  * @param elementOfList  used to cycle through the list and reach the end of the list
```

```

681     * @return elementOfList    returned as the last item in the list
682     */
683     public Node getLast(Node elementOfList)
684     {
685         if (elementOfList == null)
686         {
687             return elementOfList;           //if the linked list is empty - returns null
688         }
689         else if (elementOfList.getNext() == null)
690         {
691             return elementOfList;           //returns the last item in the list
692         }
693         else
694         {
695             return getLast(elementOfList.getNext()); //recursive traversal through the list
696         }
697     }
698     //-----
699     /**
700     * converts the Linked List to a data array so that the data contained in an array will itself be an array
701     * - no parameters
702     *
703     * @return dataArray    an object array that will contain the data within each node of the linked list
704     */
705     public Node[] toArray()
706     {
707         int k = SinglyLinkedList.length(this.head); //gets the size of the list
708         Node dataArray[] = new Node[k];             //initialises a Node array of the size of the list
709         Node listNode = getHead();                  //gets the head of the list
710         for (int j = 0; j < k-1; j++)
711         {
712             dataArray[j] = listNode;                 //initialises each item of the list to an element in the array
713             listNode = listNode.getNext();           //gets the next item in the list
714         }
715         return dataArray;                           //returns the array once complete
716     }
717     //-----
718     //state variables - private for data protection

```

```

719     private Node head;                                //the head of the list
720 }

721 //-----new class-----\\

722 import java.io.*;                                       //used for file i/o and Scanner
723 import java.util.*;                                     //used for the Scanner object
724 import java.io.IOException.*;                           //used to catch exceptions
725 import java.io.ObjectOutputStream;                     //used to write data to secondary memory using a data stream
726 import java.io.ObjectInputStream;                       //used to read data from secondary memory using a data stream
727 /**
728  * This is the main class that manages with file I/O and getting commands and information from the user
729  * that can be set into fields using the get/set methods. There is a main method that will be used to
730  * acquire this.
731  *
732  * @author Luke Geeson
733  * @version 1.20
734  * @date 12/12/2012
735  * @school Norton Knatchbull School
736  */
737 public class mainClass
738 {
739     /**
740      * This is the default constructor
741      */
742     public mainClass(){}
743     //-----
744     /**
745      * this method acts allows the user to input commands and information to the various methods. This
746      * method will interpret the commands and pass control to the other methods of this class which need it
747      * void - does not return anything - does not return anything
748      */
749     public static void main ()
750     {
751         boolean correctPas = false;                     //Boolean for the password loop
752         int attemptNo = 0;                                //used to allow 3 attempts
753         String passwordAttempt = "";                     //string for password attempts
754         Scanner kbReader = new Scanner (System.in);      //scanner used throughout method

```

```

755     while (correctPas != true)                                //password input loop
756     {
757         if ( attemptNo == 3)                                    //if the user has attempted to login 3 times
758         {
759             passwordAttempt = null;                            //same variable is used to input system code
760             while (passwordAttempt == null)                    //will loop until system code is input
761             {
762                 System.out.println("Login failed too many times, please input the SYSTEM PASSCODE");
763                 String systemCode = kbReader.nextLine();//system code input
764                 passwordAttempt = getPassword(systemCode);//using the system code to get the password
765             }
766         }
767         else
768         {
769             System.out.println("Input login password - it is case sensitive ");
770             passwordAttempt = kbReader.nextLine();
771         }
772         if (getPasswordWithoutCode().equals("") || getPasswordWithoutCode().equals(" ") || getPasswordWithoutCode() ==
773         null)
774         {
775             //if the default password is nothing or null
776             setPassword("password");
777         }
778         if (passwordAttempt.equals(getPasswordWithoutCode()))
779         {
780             //if the password is correct
781             //used to leave the loop
782             correctPas = true;
783             Date date = new Date();                             //get the date of login
784             System.out.println("Login successful - Welcome, the date is " + date.toString() + "\n");
785         }
786         else
787         {
788             //if the password is incorrect
789             System.out.println("Incorrect password - please try again " + (2-attemptNo) + " attempts remaining\n");
790             //increments for login attempts
791             if (attemptNo < 3)
792             {
793                 attemptNo++;
794                 //increment the attempts by 1
795             }
796             correctPas = false;
797             //continue the loop
798         }
799     }
800 }

```

```

793     SinglyLinkedList a = loadListFromFile();           //load the list from the file
794     boolean done = false;                             //Boolean for the main loop
795     while (done != true)                               //loop until the user is finished
796     {
797         saveListToFile(a); //saves the list after each successive - keeps everything up to date
798         System.out.println("What do you want to do? add/search/Print all/get size/sort/clear list/change password or EXIT
799 to quit");
800         String choice = kbReader.nextLine();           //takes user request
801         choice.trim();                                 //trims request for useless space
802         if(choice.equalsIgnoreCase("ADD") || choice.equalsIgnoreCase("A"))
803         {                                              //interprets user 'add' request
804             System.out.println("Input Forename");     //add forename
805             String fName = new String (kbReader.nextLine());
806             boolean hasSurname = false;               //loop for surname
807             String sName = "";
808             while (hasSurname != true)                //while an empty surname is NOT input
809             {
810                 System.out.println("Input Surname"); //acquire user input
811                 sName = new String (kbReader.nextLine());
812                 if (sName.equalsIgnoreCase("") || sName == null )
813                 {                                     //if the surname input is empty
814                     System.out.println("Invalid input - please insert a surname");
815                 }
816                 else
817                 {                                     //if the surname input is not empty
818                     hasSurname = true;
819                 }
820             }
821             System.out.println("Input Phone Number"); //add phone number
822             String pNum = new String (kbReader.nextLine());
823             System.out.println("Input Email Address"); //add email address
824             String eAdd = new String (kbReader.nextLine());
825             System.out.println("Input the first line of the Home Address"); //add home address
826             String hAdd = new String (kbReader.nextLine());
827             System.out.println("Input Post Code");     //add post code
828             String pCode = new String (kbReader.nextLine());
829             CustomerFile custDat = new CustomerFile(fName,sName,pNum,eAdd,hAdd,pCode); //create customer file for data
830             Node custNode = new Node(custDat, null);   //create node to store data

```

```
831         if (a == null || a.getHead() == null)           //if the list is empty
832         {
833             a = new SinglyLinkedList();                 //create new list
834         }
835         a.addRecord(custNode, a.getHead());              //adds the record to the list
836         System.out.println("File added successfully\n");
837     }
838     else if(choice.equalsIgnoreCase("SEARCH") || choice.equalsIgnoreCase("S"))
839     {                                                     //interprets the users 'search' request
840         Node result = null, current = null;              //nodes used for comparison and search results
841         boolean foundFile = false;                       //used for while loop
842         String search = "";                               //used for surname input for search
843         while (foundFile != true)                        //while the record is not found
844         {
845             if (result == null)                          //if this is the first search
846             {
847                 System.out.println("Input Surname of customer required");
848                 search = kbReader.nextLine();
849                 search = search.trim();
850             }
851             if (a == null)                                //if the list is empty
852             {
853                 System.out.println("Search failed - the list is empty\n");
854                 foundFile = true;
855                 continue;
856             }
857             else if (current != null)                    //is this is a second search
858             {
859                 if (current.getNext() == null)
860                 { //if the previous search yielded a wrong record and it is the last item (or only item) in the list
861                     System.out.println("search failed");
862                     foundFile = true;
863                     continue;
864                 }
865                 else
866                 {
867                     //will continue search from the previous point in the list
868                     result = a.searchList(current.getNext(), search);
869                 }
870             }
871         }
872     }
```

```

869     }
870     else                                     //search from the beginning
871     {
872         result = a.searchList(a.getHead(), search);
873     }
874     if (result == null || search == "")      //if the search fails OR the search input is empty
875     {
876         System.out.println("Search failed, please try again");
877         foundFile = true;
878         break;
879     }
880     else                                     //if a record is found
881     {
882         System.out.println("search successful - printing details of file\n");
883         result.getData().displayAllDetails();
884     }
885     System.out.println("Is this the record you need? yes/no");
886     String validation = kbReader.nextLine(); //request confirmation for found record
887     if (validation.equalsIgnoreCase("YES") || validation.equalsIgnoreCase("Y"))
888     {                                       //interpret user confirmation for the correct record
889         current = result;
890         foundFile = true;
891         boolean doneEdittingFile = false; //used for modification or remove loop
892         while (doneEdittingFile != true)
893         {
894             System.out.println("What would you like to do with this record? remove/modify or EXIT");
895             String remOrMod = kbReader.nextLine(); //requests action to be performed on record
896             if (remOrMod.equalsIgnoreCase("REMOVE") || remOrMod.equalsIgnoreCase("R"))
897             {
898                 boolean finRemove = false; //used for remove loop
899                 while (finRemove != true)
900                 {
901                     System.out.println("Are you sure that you want to remove this file? yes/no");
902                     String confirm = kbReader.nextLine(); //confirm that the user wants to remove the file
903                     if (confirm.equalsIgnoreCase("YES") || confirm.equalsIgnoreCase("Y"))
904                     {
905                         //if they do
906                         a.removeNode(current, a.getHead());
907                         System.out.println("Record successfully removed\n");

```



```

907         finRemove = true;
908     }
909     else if(confirm.equalsIgnoreCase("NO") || confirm.equalsIgnoreCase("N"))
910     {
911         //if they do not
912         System.out.println("File not removed\n");
913         result = null;
914         current = null;
915         finRemove = true;
916     }
917     else
918     {
919         //if the user inputs invalid data
920         System.out.println("Invalid input - please try again");
921         finRemove = false;
922     }
923 }
924 doneEdittingFile = true; //used to escape remove or modify loop
925 }
926 else if (remOrMod.equalsIgnoreCase("MODIFY") || remOrMod.equalsIgnoreCase("M") ||
927 remOrMod.equalsIgnoreCase("CHANGE") || remOrMod.equalsIgnoreCase("C"))
928 {
929     //interprets the modify action for the record
930     boolean finChange = false; //used for modification loop
931     while (finChange != true)
932     {
933         System.out.println("What data within this record would you like to change?
934 \nforename/surname/email address/home address/post code/phone number or EXIT to leave");
935         String changeDecision = kbReader.nextLine();//to specify what is to be changed
936         if (changeDecision.equalsIgnoreCase("FORENAME") ||
937 changeDecision.equalsIgnoreCase("FIRSTNAME") || changeDecision.equalsIgnoreCase("FIRST NAME") ||
938 changeDecision.equalsIgnoreCase("FIRST") || changeDecision.equalsIgnoreCase("F"))
939         {
940             //to change the forename
941             System.out.println("Input new forename");
942             String newForename = kbReader.nextLine();
943             current = a.changeNode(current, changeDecision, newForename);
944             System.out.println("Forename changed\n");
945             finChange = true; //used to escape modification loop
946         }
947     }
948 }

```

```

943         else if (changeDecision.equalsIgnoreCase("SURNAME") ||
944 changeDecision.equalsIgnoreCase("SECOND NAME") || changeDecision.equalsIgnoreCase("SECONDNAME") ||
945 changeDecision.equalsIgnoreCase("S"))
946         {
947             //to change the surname
948             boolean hasSurname = false; //used for surname loop
949             String newSurname = "";
950             while (hasSurname != true) //test to insure surname input is not empty
951             {
952                 System.out.println("Input New Surname");
953                 newSurname = new String (kbReader.nextLine());
954                 if (newSurname.equalsIgnoreCase("") || newSurname == null )
955                 {
956                     //if the surname input is empty
957                     System.out.println("Invalid input - please insert a surname");
958                 }
959                 else
960                 {
961                     //else escapes the surname loop
962                     hasSurname = true;
963                 }
964             }
965             current = a.changeNode(current, changeDecision, newSurname);
966             System.out.println("Surname changed");
967             finChange = true;
968             a.sortList(); //sorts the list as the surname is the key value
969         }
970         else if ((changeDecision.equalsIgnoreCase("PHONENUMBER") ||
971 changeDecision.equalsIgnoreCase("PHONE NUMBER") || changeDecision.equalsIgnoreCase("PHONE") ||
972 changeDecision.equalsIgnoreCase("P"))))
973         {
974             //to change the phone number of the record
975             System.out.println("Input new phone number");
976             String newNumber = kbReader.nextLine();
977             current = a.changeNode(current, changeDecision, newNumber);
978             System.out.println("Phone number changed\n");
979             finChange = true;
980         }
981         else if ((changeDecision.equalsIgnoreCase("EMAILADDRESS") ||
982 changeDecision.equalsIgnoreCase("EMAIL ADDRESS") || changeDecision.equalsIgnoreCase("EMAIL") ||
983 changeDecision.equalsIgnoreCase("E"))))
984         {
985             //to change the email address of the record

```

```

981         System.out.println("Input new Email Address");
982         String newEmail = kbReader.nextLine();
983         current = a.changeNode(current, changeDecision, newEmail);
984         System.out.println("Email address changed\n");
985         finChange = true;
986     }
987     else if ((changeDecision.equalsIgnoreCase("HOMEADDRESS") ||
988 changeDecision.equalsIgnoreCase("HOME ADDRESS") || changeDecision.equalsIgnoreCase("HOME") ||
989 changeDecision.equalsIgnoreCase("H")))
990     {
991         //to change the home address of the record
992         System.out.println("Input the new first line of house address");
993         String newHAddress = kbReader.nextLine();
994         current = a.changeNode(current, changeDecision, newHAddress);
995         System.out.println("Home address changed\n");
996         finChange = true;
997     }
998     else if (changeDecision.equalsIgnoreCase("POSTCODE") || changeDecision.equalsIgnoreCase
999 ("POST CODE"))
1000     {
1001         //change the post code of the record
1002         System.out.println("Input new post code");
1003         String newPCode = kbReader.nextLine();
1004         current = a.changeNode(current, changeDecision, newPCode);
1005         System.out.println("Post code changed\n");
1006         finChange = true;
1007     }
1008     else if (changeDecision.equalsIgnoreCase("EXIT"))
1009     {
1010         //if the user does not want to edit the record
1011         finChange = true; //exits the loop
1012         continue; //returns to the main menu
1013     }
1014     else
1015     {
1016         //else if the user inputs invalid data at this stage
1017         finChange = false; //will continue in the modify loop until correct input is seen
1018         System.out.println("Invalid input - please try again");
1019     }
1020 }
doneEdittingFile = true; //used to escape the loop
}

```

```
1019         else if(remOrMod.equalsIgnoreCase("EXIT") || remOrMod.equalsIgnoreCase("E"))
1020         { //if the user decides to quit instead of removing or modifying the record
1021             doneEdittingFile = true; //exit the loop - return to main menu
1022             continue;
1023         }
1024         else
1025         { //if invalid commands are input at the remove or modify stage
1026             doneEdittingFile = false; //continue the loop
1027             System.out.println("Invalid input - please try again\n");
1028             remOrMod = null; //resets input for repeat of request when loop continues
1029         }
1030     }
1031 }
1032 else if (validation.equalsIgnoreCase("NO") || validation.equalsIgnoreCase("N"))
1033 { //if the record found is not the one required
1034     current = result; //set it as the new current item so that the search continues from this point
1035 }
1036 else
1037 { //if an invalid command is detected
1038     System.out.println("Invalid request, please try again"); //invalid input and continue loop
1039 }
1040 }
1041 }
1042 else if(choice.equalsIgnoreCase("PRINT ALL") || choice.equalsIgnoreCase("PRINT") || choice.equalsIgnoreCase("P"))
1043 { //if the user wants to print the list
1044     if (a == null || a.isEmpty() == true)
1045     { //if the list is empty
1046         System.out.println("The list is empty - cannot print list\n");
1047     }
1048     else
1049     { //else print the list
1050         a.printList(a.getHead()); //calls print list method of the class
1051     }
1052 }
1053 else if(choice.equalsIgnoreCase("GET SIZE") || choice.equalsIgnoreCase("SIZE") || choice.equalsIgnoreCase("GS"))
1054 { //if the user wants the size of the list
1055     if (a == null || a.isEmpty() == true)
1056     { //if the list is empty
```

```
1057         System.out.println("The list is empty, the size is 0\n");
1058         continue;
1059     }
1060     else
1061     {
1062         //else return and print the size of the list
1063         System.out.println("The size of the list is " + a.length(a.getHead()) + " items\n");
1064     }
1065     else if(choice.equalsIgnoreCase("SORT"))
1066     {
1067         //if the user wants to sort the list
1068         a.sortList(); //calls the sort method
1069     }
1070     else if(choice.equalsIgnoreCase("CLEAR LIST") || choice.equalsIgnoreCase("CLEAR") || choice.equalsIgnoreCase("C"))
1071     {
1072         //if the user wants to clear the list
1073         //used for clear list confirmation loop
1074         boolean doClear = false;
1075         while (doClear != true)
1076         {
1077             System.out.println("Are you sure you want to clear the whole list? yes/no");
1078             String toClear = kbReader.nextLine(); //confirmation
1079             if (toClear.equalsIgnoreCase("YES") || toClear.equalsIgnoreCase("Y"))
1080             {
1081                 //the user wants to clear the list
1082                 a = null;
1083                 System.out.println("List cleared\n");
1084                 doClear = true; //exit clear list loop
1085                 toClear = null; //clears input
1086             }
1087             else if(toClear.equalsIgnoreCase("NO") || toClear.equalsIgnoreCase("N"))
1088             {
1089                 //the user does not want to clear the list
1090                 System.out.println("List not cleared");
1091                 doClear = true; //exit clear list loop
1092                 toClear = null; //clears input
1093             }
1094             else
1095             {
1096                 //else the user inputs invalid command
1097                 System.out.println("Invalid request - please try again");
1098                 doClear = false; //exit the clear list loop
1099             }
1100         }
1101     }
1102 }
```

```

1095     }
1096     else if(choice.equalsIgnoreCase("EXIT") || choice.equalsIgnoreCase("E"))
1097     {
1098         System.out.println("System closing");
1099         done = true;
1100     }
1101     else if (choice.equalsIgnoreCase("CHANGEPASSWORD") || choice.equalsIgnoreCase("CHANGE PASSWORD") ||
1102 choice.equalsIgnoreCase("PASSWORD"))
1103     {
1104         boolean isPassword = false;
1105         String oldPassword = "";
1106         while (isPassword != true)
1107         {
1108             System.out.println("Input old password, it is case sensitive");
1109             oldPassword = kbReader.nextLine();
1110             if(oldPassword.equals(getPasswordWithoutCode()))
1111             {
1112                 System.out.println("Input new password");
1113                 String newPass = kbReader.nextLine();
1114                 setPassword(newPass);
1115                 System.out.println("Password Changed\n");
1116                 isPassword = true;
1117             }
1118             else
1119             {
1120                 boolean confirm = false;
1121                 while (confirm != true)
1122                 {
1123                     //ask user whether they want to retry input or return to main menu and not change password
1124                     System.out.println("Incorrect password, do you want to try again? yes/no");
1125                     String retry = kbReader.nextLine();
1126                     if (retry.equalsIgnoreCase("YES") || retry.equalsIgnoreCase("Y"))
1127                     {
1128                         //if they do want to try again
1129                         isPassword = false;
1130                         //continue loop
1131                         confirm = true;
1132                         //exit this inner loop
1133                     }
1134                     else if(retry.equalsIgnoreCase("NO") || retry.equalsIgnoreCase("N"))
1135                     {
1136                         //if they do not want to try again
1137                         isPassword = true;
1138                         //exits loop
1139                     }
1140                 }
1141             }
1142         }
1143     }

```

```

1133         confirm = true;           //exits this inner loop
1134         System.out.println("Password not changed\n");
1135     }
1136     else
1137     {
1138         //if the user inputs invalid command
1139         System.out.println("Invalid input, please try again");
1140         confirm = false;           //continues this loop until valid input is found
1141     }
1142 }
1143 }
1144 }
1145 else
1146 {
1147     //if the user inputs wrong information at this stage
1148     System.out.println("Invalid request, please try again\n");
1149     done = false;                 //continue loop
1150     continue;
1151 }
1152 saveListToFile(a);               //saves the list with any final changes
1153 kbReader.close();                //closes the reader
1154 }
1155 //-----
1156 /**
1157  * This method will serialise and save the singly linked list object to the ".ser" file which is stored
1158  * in the 'doc' folder of this project - void, no returns
1159  *
1160  * @param lst         the Singly linked list which is to be saved
1161  */
1162 private static void saveListToFile(SinglyLinkedList lst)
1163 {
1164     try
1165     {
1166         ObjectOutputStream os = new ObjectOutputStream(new FileOutputStream(fileName));
1167         os.writeObject(lst);           //writes the list to the file
1168         os.flush();                     //flush the stream
1169         os.close();                     //closes output stream to finalise changes to the list
1170     }

```

```
1171         catch (FileNotFoundException e)
1172         {
1173             e.printStackTrace();
1174         }
1175         catch (IOException e)
1176         {
1177             e.printStackTrace();
1178         }
1179     }
1180 //-----
1181 /**
1182  * This method will 'deserialize' and load the SinglyLinkedList object from the '.ser' file, which
1183  * is saved in the "doc" folder of this project
1184  *
1185  * @return SinglyLinkedList the list which is loaded from the secondary storage
1186  */
1187 private static SinglyLinkedList loadListFromFile()
1188 {
1189     SinglyLinkedList lst = null;
1190     try
1191     {
1192         ObjectInputStream is = new ObjectInputStream(new FileInputStream(fileName));
1193         lst = (SinglyLinkedList) is.readObject();    //loads the list from the file
1194         is.close();                                //closes reader
1195     }
1196     catch (FileNotFoundException e)
1197     {
1198         e.printStackTrace();
1199     }
1200     catch (IOException e)
1201     {
1202         e.printStackTrace();
1203     }
1204     catch (ClassNotFoundException e)
1205     {
1206         e.printStackTrace();
1207     }
1208     return lst;
```



```
1209 }
1210 //-----
1211 /**
1212  * This method is used to set the password that the system uses as a level of security - private so
1213  * it cannot be changed from outside the class. It is simply a set method and it saves the password data to a new file
1214  *
1215  * @param newPassword the new password that the user wants to change to
1216  */
1217 private static void setPassword(String newPassword)
1218 {
1219     try
1220     {
1221         ObjectOutputStream osPassword = new ObjectOutputStream (new FileOutputStream (passwordFileName));
1222         osPassword.writeObject(newPassword);           //writes the new password to file
1223         osPassword.flush();                           //flush the stream
1224         osPassword.close();                           //close the stream
1225     }
1226     catch (FileNotFoundException e)
1227     {
1228         e.printStackTrace();
1229     }
1230     catch (IOException e)
1231     {
1232         e.printStackTrace();
1233     }
1234 }
1235 //-----
1236 /**
1237  * This method is used to return the user password if they forget it, it requires them to pass
1238  * a integer which will be compared with a unique and unchanging system code. Private so it is not
1239  * accessible by malicious users.
1240  *
1241  * @param systCode to be compared with the system code
1242  * @return userPassword the user password returned
1243  */
1244 private static String getPassword(String systCode)
1245 {
1246     if (systCode.equals(PASSCODE))
```

```
1247         {                                     //if the passcode input is equals the system code
1248             String userPassword = "";
1249             try
1250             {
1251                 ObjectInputStream isPassword = new ObjectInputStream (new FileInputStream (passwordFileName));
1252                 userPassword = (String) isPassword.readObject();//retrieve the password from the file
1253                 isPassword.close();                     //close the stream
1254                 System.out.println("Correct system code, your password is: " + userPassword);
1255             }
1256             catch(FileNotFoundException e)
1257             {
1258                 e.printStackTrace();
1259             }
1260             catch(IOException e)
1261             {
1262                 e.printStackTrace();
1263             }
1264             catch(ClassNotFoundException e)
1265             {
1266                 e.printStackTrace();
1267             }
1268             return userPassword;                     //return the retrieved password
1269         }
1270         else
1271         {                                     //else if the password input is not equal to the passcode
1272             System.out.println("That is the incorrect system code, please try again");
1273             return null;
1274         }
1275     }
1276     //-----
1277     /**
1278     * This method also returns the user password however, this method is only used to compare the
1279     * passwords input with those on file - it is not accessible by the user - takes no parameters
1280     *
1281     * @return userPassword the password stored on file
1282     */
1283     private static String getPasswordWithoutCode()
1284     {
```

```

1285     String userPassword = "";
1286     try
1287     {
1288         ObjectInputStream isPassword = new ObjectInputStream (new FileInputStream (passwordFileName));
1289         userPassword = (String) isPassword.readObject();//retrieve the password from the file
1290         isPassword.close();                          //close the stream
1291     }
1292     catch(FileNotFoundException e)
1293     {
1294         e.printStackTrace();
1295     }
1296     catch(IOException e)
1297     {
1298         e.printStackTrace();
1299     }
1300     catch(ClassNotFoundException e)
1301     {
1302         e.printStackTrace();
1303     }
1304     return userPassword;
1305 }
1306 //-----
1307 /**
1308  * This is a "set" method which sets the file name of the ".ser" file which stores the data in
1309  * Serialized form - void as it simply sets data and private because it should not be accessible
1310  * from outside this class.
1311  *
1312  * @param newFileName      sets the variable fileName as this value
1313  */
1314 private static void setFileName(String newFileName)
1315 {
1316     fileName = newFileName + ".ser";                //saves the file path of the customer database file
1317 }
1318 //-----
1319 /**
1320  * This method simply returns the value assigned to fileName, returns the name of the file - simply
1321  * a get method so there are no parameters and private so outside users cannot see the name of the file
1322  *

```

```
1323     * @return String         the file name
1324     */
1325     private static String getFileName()
1326     {
1327         return fileName;                                //returns the file name of the database location
1328     }
1329     //-----
1330     //state variables - private for data protection
1331     private static String fileName = "ListData.ser";    //the name of the file being accessed
1332     private static final String PASSCODE = "14GF5602C2"; //a set passcode used to retrieve user password if lost
1333     private static String passwordFileName = "passDat.ser"; //the string in which the user password is stored
1334 }
```