Development

Techniques used in developing the program that will be discussed include:

- External Libraries
- General Design
    - Code Structure
    - Graphical User Interface and Layout
    - Inheritance
- Storing and Loading Item Data
    - Data Storage
    - Item Representation
    - Complex Data Structures
    - Obtaining Data From Files
- Item Manipulation
    - Viewing an Item
    - Saving Item Data
    - Editing and Adding an Item
    - Deleting an Item
- Error Handling
    - Exceptions and Error Messages
    - Precautions

External Libraries

```
import java.awt.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
```

Several libraries (or parts of libraries) were imported to supplement the code. The `java.awt` and `javax.swing` packages were imported to help the `MainMenu`, `EditItemWindow`, `AddItemWindow`, `ItemWindow`, and `DeleteVerification` classes extend the `JFrame` class to create, hold, and assign functionality to GUI components. The `java.io` package provides for system input and output through `Files`, `PrintWriters`, and `BufferedReaders`. The `java.util` package allows access to Collections like `ArrayList<E>` and `Scanner`.

General Design

Code Structure

An object-oriented approach was used to program the code. Every class is fully encapsulated with private states and public accessors when necessary. Encapsulation allows ease of extensibility in the future.

Graphical User Interface and Layout

The graphical user interface(GUI) layout of my program is adapted from common applications to make my program's interface as intuitive and familiar as possible. Each window of the program is its own class: `MainMenu`, `EditItemWindow`, `AddItemWindow`, `ItemWindow`, and `DeleteVerification`. Each GUI class has an individual `initializeGUI()` method, called in its constructor, to declare and initialize all necessary GUI components including tables, labels, buttons, and text fields. Only the `MainMenu` class is initialized in the `Driver` class because only the `MainMenu` window is displayed initially. As the user executes different actions, the `MainMenu` will be disposed and appropriate alternate windows are initialized and displayed like the add button shown below.

```
addBtn.addMouseListener(new MouseAdapter() {
    @Override
    public void mouseClicked(MouseEvent arg0)
    {
        AddItemWindow add = new AddItemWindow();
        dispose();
    }
});
```

Inheritance

`AddItemWindow` extends `EditItemWindow` because they share many of the same methods. Methods written specifically for `EditItemWindow` are kept `private`. Three methods—`initializeGUI()`, `save()`, and `getRBtnType()`-- are overridden to account for differences between the two classes.

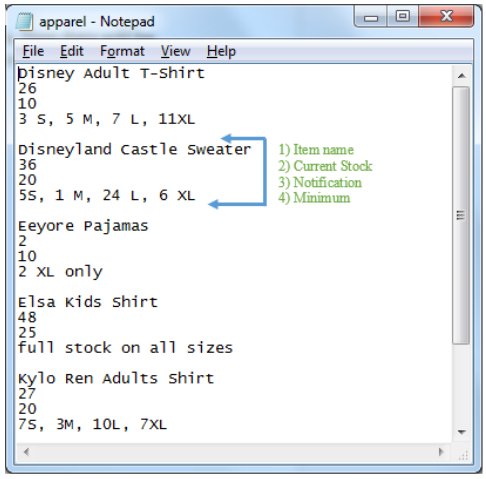
```
//constructor
public AddItemWindow()
{
    super();
    myItem = null;
    initializeGUI();
}

//override parent's initializeGUI()
//post-condition: all GUI elements have been initialized, necessary action events of GUI elements
public void initializeGUI() {

//override parent's save()
//post-condition: if user input is invalid, errors will display
//                if user input is valid, new item is added to appropriate file
public void save() {

//Overrides with child's privates, not parent's
//post-condition: returns text of selected radio button or null if none is selected
public String getRBtnType() {
```

Storing and Loading Item Data



Data Storage

In order to retain the data that the user has input into the program, item data will be stored as a group of lines of `Strings` in a text file. There is a file for each item type. The program utilizes abstraction, so users will not have direct access to these text files through the program. Instead, the program will obtain and manipulate data from the files through methods in the GUI classes based on user mouse actions. If the text files do not initially exist because there are no items in the database, text files will be created automatically.

Item Representation

According to user specifications, each `Item` has an associated name, current stock, notification minimum, and item type, as well as an optional comments section for user convenience. Thus, an `Item` object has each data represented as a private. The item type options for an item are limited to apparel, food and drink, miscellaneous, and souvenirs by only allowing users to choose from labelled radio buttons. These item types are chosen according to the client's current category classifications (see Appendix, evidence of previous method of stock management).

```
private String myName;
private int myCurrentStock;
private int myMinimum;
private String myItemType;
private String myComment;

//constructor, requires all privates to be specified
public Item(String name, int currentStock, int minimum,  String itemType, String comment)
{
    myName = name;
    myCurrentStock = currentStock;
    myMinimum = minimum;
    myItemType = itemType;
    myComment = comment;
```

Complex Data Structures

Throughout the program, `DefaultListModels`, `DefaultTableModels`, and `ArrayLists` are used to organize `Items` and item data. These data structures are dynamic and thus increase efficiency and memory compared to simple data structures. For example, if arrays were used instead, every time an `Item` was added or deleted, a new array would need to be created to replace the old array, accumulating garbage and wasting memory.

Obtaining Data From Files

To obtain the `Strings` of item data in a text file, a `Scanner` is used to read through the file within the method `load(String type)`. The `File` is determined by the item type specified by the `String` parameter. The method returns a `DefaultListModel` with an `Item` type argument of ever item detailed in the text file.

```
//post-condition: returns DefaultListModel representation of all items saved in specified item type file
private DefaultListModel<Item> load(String type)
{
    DefaultListModel<Item> list = new DefaultListModel<Item>();
    File file = new File(type+".txt");//file is named after item type
    try
    {
        Scanner scanner = new Scanner(file);
        while (scanner.hasNextLine())
        {
            String name = scanner.nextLine();
            int currentStock  = Integer.parseInt(scanner.nextLine());
            int minimum = Integer.parseInt(scanner.nextLine());
            String comment = scanner.nextLine();
            list.addElement(new Item(name, currentStock, minimum, type, comment));
            if (scanner.hasNextLine())
                scanner.nextLine();
        }
        scanner.close();
    }
    catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
    return list;
}
```

The `loadMasterList()` method calls `load(String type)` multiple times as a helper method to create a `DefaultListModel<Item>` of every item. The `loadMasterList()` method is called in the constructor to initialize private `DefaultListModel<Item> masterList`, which represents every item in the database. The same technique is used in the method `loadNotificationsList()` to initialize private `DefaultListModel<Item> notificationsList` with the exception that `notificationsList` only contains every item with a notification minimum higher than the current stock.

```
//post-condition: returns DefaultListModel representation of all items saved in all files
private DefaultListModel<Item> loadMasterList()
{
    DefaultListModel<Item> apparel = load("apparel");
    DefaultListModel<Item> foodDrink = load("food and drink");
    DefaultListModel<Item> miscellaneous = load("miscellaneous");
    DefaultListModel<Item> souvenir = load("souvenir");

    DefaultListModel<Item> masterList = new DefaultListModel<Item>();
    for (int i = 0; i < apparel.size(); i++)
        masterList.addElement(apparel.get(i));

    for (int i = 0; i < foodDrink.size(); i++)
        masterList.addElement(foodDrink.get(i));

    for (int i = 0; i < miscellaneous.size(); i++)
        masterList.addElement(miscellaneous.get(i));

    for (int i = 0; i < souvenir.size(); i++)
        masterList.addElement(souvenir.get(i));
    return masterList;
}
```

Item Manipulation

Saving Item Data

Whether adding or editing an item, once the user fills out all required data fields with valid entries and clicks the save button, the method `save()` is called to retrieve text from the text fields and selected radio button. The `save()` method is overwritten in `AddItemWindow` to account for differences between the two functions, but the methodology remains the same. Both `save()` methods heavily utilize procedural abstraction to break down the long method into smaller tasks, to enhance readability, and to allow ease of extensibility. Some helper methods include `displayDigitError()`, `getRBtnType()`, `addItem(Item item, String fileName)`. `EditItemWindow`'s `save()` method also includes multiple if-else statements to account for different editing cases that `AddItemWindow` does not require.

```java
//post-condition: if user input is invalid, errors will display
//                if user input is valid, selected item's edits will be updated in its file
public void save()
{
    boolean success = false;
    String tf = nameTF.getText();
    String name = "";
    if (tf.length() > 0)
        name = tf.substring(0, 1).toUpperCase() + tf.substring(1);
    int stock = -1;
    try
    {
        //converts stockTF text to Integer, checks that stock is non-negative integer and displays error if not
        if (!stockTF.getText().equals(""))
            stock = Integer.parseInt(stockTF.getText());
        displayDigitError(digitErrorLbl, stock);

        int min = -1;
        try
        {
            //converts minimumTF text to Integer, checks that minimum is non-negative integer and displays error if not
            if (!minimumTF.getText().equals(""))
                min = Integer.parseInt(minimumTF.getText());
            //if digitErrorLbl not already visible from invalid stock
            if (digitErrorLbl.isVisible() == false)
                displayDigitError(digitErrorLbl, min);

            String type = getRBtnType();

            //gets comment; if no comment is inputed, then default comment is "no comment"
            String comments;
            if (commentsTF.getText().trim().equals(""))
                comments = "no comment";
            else
                comments = commentsTF.getText();

            String file = type + ".txt";
            displayEmptyError(emptyErrorLbl, nameTF.getText(), stockTF.getText(), minimumTF.getText());
            //only if either item name or type is changed, then there is a possibility of changing item to existing item
            if (!getRBtnType().equals(myItem.getItemType()) || name.equals(myItem.getItemType()))
                    displayExistingError(existsErrorLbl, name, type);

            //if all user input is valid
            if (!digitErrorLbl.isVisible() && !emptyErrorLbl.isVisible() && !existsErrorLbl.isVisible())
            {
                Item item = new Item(name, stock, min, type, comments);
                //name and type is the same (edit in same file in same location)
                if (getRBtnType().equals(myItem.getItemType()) && name.equals(myItem.getName()))
                {
                    editItem(item, file);
                    success = true;
                    myItem = item;
                }
                //type is changed (delete from old type file, add to new type file)
                else if (!getRBtnType().equals(myItem.getItemType()))
                {
                    addItem(item, file);
                    deleteItem(myItem.getItemType()+ ".txt");
                    success = true;
                    myItem = item;
                }
                //name is changed (same file, add to different location)
                else if(!name.equals(myItem.getName()))
                {
                    deleteItem(myItem.getItemType()+ ".txt");
                    addItem(item, file);
                    success = true;
                    myItem = item;
                }
            }
            if (success)
            {
                dispose();
                MainMenu menu = new MainMenu();
            }
        } catch (NumberFormatException e) {
            digitErrorLbl.setVisible(true);
            displayEmptyError(emptyErrorLbl, nameTF.getText(), stockTF.getText(), minimumTF.getText());
        }
    } catch (NumberFormatException e) {
        digitErrorLbl.setVisible(true);
        displayEmptyError(emptyErrorLbl, nameTF.getText(), stockTF.getText(), minimumTF.getText());
    }
}
```

Editing and Adding an Item

The `save()` method will invoke either `editItem(Item item, String fileName)` or `addItem(Item item, String fileName)`to manipulate data. Both employ similar algorithmic thinking to complete their task. `editItem(Item item, String fileName)` is explained below.

The method `editItem(Item item, String fileName)` uses a `BufferedReader` to read lines of `String` in the specified type file based on the organization of the item data as detailed in the "Data Storage" subsection. An `ArrayList<String> list` is used to store the `String` read by the `BufferedReader`. The `BufferedReader` first reads and stores item data for every item whose name precedes that of the new item alphabetically. Then, the old data of the item being edited is skipped, and the new data is added to `list` as `Strings`. The `BufferedReader` then continues to read and store item data in `list` until the end of the file. After the `BufferedReader` is closed, a `PrintWriter` is declared and initialized, which overwrites the same file with every `String` stored in `list`.

```
//post-condition: information of old item is replaced with new item information (based on correct user input) in correct type file;
//                inserted to correct position so that items(with following item information) in file are sorted alphabetically by item name
private void editItem(Item item, String fileName)
{//myItem = old Item to be changed
    try{
        //reads from specified type file
        File file = new File (fileName);
        FileReader scanner = new FileReader(file);
        BufferedReader reader = new BufferedReader(scanner);
        ArrayList<String> list = new ArrayList<String>();
        try {
            //copies every line in file before editing item to ArrayList<String> list
            String next = reader.readLine();
            while (next!=null && next.compareTo(myItem.getName())<0) //next is alphabetically before name
            {
                list.add(next);//name
                list.add(reader.readLine()); //stock
                list.add(reader.readLine()); //min
                list.add(reader.readLine()); //comment
                list.add(reader.readLine()); //blank line
                next = reader.readLine();//next name
            }

            //adds information of editing item to ArrayList<String> list and skips lines of old item information in file
            list.add(item.getName());
            list.add(""+ item.getCurrentStock());
            list.add(""+ item.getMinimum());
            list.add(item.getComment());
            list.add(" ");//blank line

            reader.readLine(); //skip old stock
            reader.readLine(); //skip old min
            reader.readLine(); //skip old comment
            reader.readLine(); //skip old blank line

            //copies every line in file after old item information to ArrayList<String> list if new item is not last
            while ((next = reader.readLine()) != null)
            {
                list.add(next);
            }
            reader.close();

            //overwrites original file with every line of string saved in ArrayList<String> list
            PrintWriter writer = new PrintWriter(file);
            for (String line : list)
                writer.println(line);
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (FileNotFoundException e)
    {
        e.printStackTrace();
    }
}
```

Deleting an Item

A similar methodology of implementing a `BufferedReader` and `PrintWriter` is used to manipulate the contents of a file to delete an item.

Error Handling

Exceptions and Error Messages

In the `MainMenu` window, the `JLabel selectErrorLbl` is displayed if the edit or delete button is clicked, but no item row is selected either in the Notification List or Main List.

The `save()` methods account for invalid input error handling. There are three possible errors that may display when manipulating item data, multiple of which could be displayed at once. The `JLabel digitErrorLbl` will display if either the stock or notification minimum user entries are not positive integers. A try-catch block is used when `Integer.parseInt(String text)` is invoked to prevent non-integers and an if-else statement is used to check that the input is a positive number.

```
try
{  ........
    //gets stock value, checks that stock is non-negative integer digit value and displays error if not
    if (!stockTF.getText().equals(""))
        stock = Integer.parseInt(stockTF.getText());
    displayDigitError(stock);

    ........
} catch (NumberFormatException e) {
    digitErrorLbl.setVisible(true);
    displayEmptyError();
}
 //post-condition: makes digitErrorLbl visible if specified num is  less than 0,
 //                or invisible if num is greater than or equal to 0
 private void displayDigitError(int num)
 {
     if (num < 0)
         digitErrorLbl.setVisible(true);
     else
         digitErrorLbl.setVisible(false);
 }
```

Partial code

Secondly, `JLabel existingErrorLbl` will display if the user attempts to add or edit an item to the same item name and type as an existing item by invoking the `findItem(String name, String type)` method from the `MainMenu` class which returns `null` if an `Item` is not found.

```
//post-condition: returns true if item with same name and same type already exists; else returns false
private void displayExistingError(String name, String type)
{
    MainMenu main = new MainMenu();
    boolean isExisting = false;
    if (main.findItem(name, type)!=null)//exists
        isExisting = true;
    main.exitMain();
    if (isExisting)
        existsErrorLbl.setVisible(true);
    else
        existsErrorLbl.setVisible(false);
}
```

Lastly, `JLabel emptyErrorLbl` will display if any of the required fields (denoted by an asterisk) are left empty using an if-else conditional.

```
//post-condition: makes emptyErrorLbl visible if text nameTF, stockTF or minimumTF text is an empty string or if no radio button is selected,
//                or invisible if not
private void displayEmptyError()
{
    if (nameTF.getText().equals("") ||stockTF.getText().equals("") || minimumTF.getText().equals("") || getRBtnType() == null)
        emptyErrorLbl.setVisible(true);
    else
        emptyErrorLbl.setVisible(false);
}
```

Precautions

On the Main Menu window, to ensure that the edit and delete button will work in either the Notification List or Main List, only one item row in total may be selected. To ensure so, whenever an item is selected in one list, the other list's selection is cleared.

```
public void mouseClicked (MouseEvent event) {
    if (event.getClickCount() == 1 && mainJTable.getSelectedRow() != -1)//mainJTable has selected
                mainJTable.clearSelection();
   ...
```

Partial code

In order to organize the data in the text files, all `Items` are alphabetically sorted by comparing item names with a `compareTo(String item2Name)` method when manipulating items in a file (as explained in "Editing and Adding an Item" subsection).

```
while (next!=null && next.compareTo(myItem.getName())<0) //next is alphabetically before name
{
    list.add(next);//name
    list.add(reader.readLine()); //stock
    list.add(reader.readLine()); //minimum
    list.add(reader.readLine()); //comment
    list.add(reader.readLine()); //blank line
    next = reader.readLine();//next name
}
```

Partial code