Assignment10 (total 220 points)

1. SL275: (total 90 points)

Module 9: Text-Based Applications

Ex. 1 Write a File with Numbered Lines (level 1) **(20 points)**

Module 15: Advanced I/O Streams

Ex. 1 Implement Object Serialization (level 1) **(30 points)**
Ex. 3 Create a Simple Database Program (level 3) **(40 points)**

Requirement:

1. display all products
2. modify a product
3. add a product
4. quit

2. **File I/O in the Gourmet Coffee (total 50 points)**

Using File I/O in the Gourmet Coffee System

Prerequisites, Goals, and Outcomes

Prerequisites: Before you begin this exercise, you need mastery of the following:

- *Java API*
  - Knowledge of the class StringTokenizer
- *File I/O*
  - Knowledge of file I/O
    - How to read data from a file
    - How to write data to a file

Goals: Reinforce your ability to use file I/O

Outcomes: You will master the following skills:

- Produce applications that read data from a file and parse it
- Produce applications that write data to a file

Background

In this assignment, you will create another version of the *Gourmet Coffee System*. In previous versions, the data for the product catalog was hard-coded in the application. In this version, the data will be loaded from a file. Also, the user will be able to write the sales information to a file in one of three formats: plain text, HTML, or XML. Part of the work has been done for you and is provided in the student archive. You will implement the code that loads the product catalog and persists the sales information.

Description

The *Gourmet Coffee System* sells three types of products: coffee, coffee brewers, and accessories for coffee consumption. A file called catalog.dat stores the product data:

- *catalog.dat*. File with product data

Every line in catalog.dat contains exactly one product.

A line for a coffee product has the following format:

Coffee_*code_description_price_origin_roast_flavor_aroma_acidity_body*

where:

- "Coffee" is a prefix that indicates the line type.
- *code* is a string that represents the code of the coffee.
- *description* is a string that represents the description of the coffee.
- *price* is a double that represents the price of the coffee.
- *origin* is a string that represents the origin of the coffee.
- *roast* is a string that represents the roast of the coffee.
- *flavor* is a string that represents the flavor of the coffee.
- *aroma* is a string that represents the aroma of the coffee.
- *acidity* is a string that represents the acidity of the coffee.
- *body* is a string that represents the body of the coffee.

The fields are delimited by an underscore (_). You can assume that the fields themselves do not contain any underscores.

A line for a coffee brewer has the following format:

Brewer_*code_description_price_model_waterSupply_numberOfCups*

where:

- "Brewer" is a prefix that indicates the line type.
- *code* is a string that represents the code of the brewer.
- *description* is a string that represents the description of the brewer.
- *price* is a double that represents the price of the brewer.
- *model* is a string that represents the model of the coffee brewer.
- *waterSupply* is a string that represents the water supply of the coffee brewer.
- *numberOfCups* is an integer that represents the capacity of the coffee brewer in number of cups.

The fields are delimited by an underscore (**_**). You can assume that the fields themselves do not contain any underscores.

A line for a coffee accessory has the following format:

Product_*code_description_price*

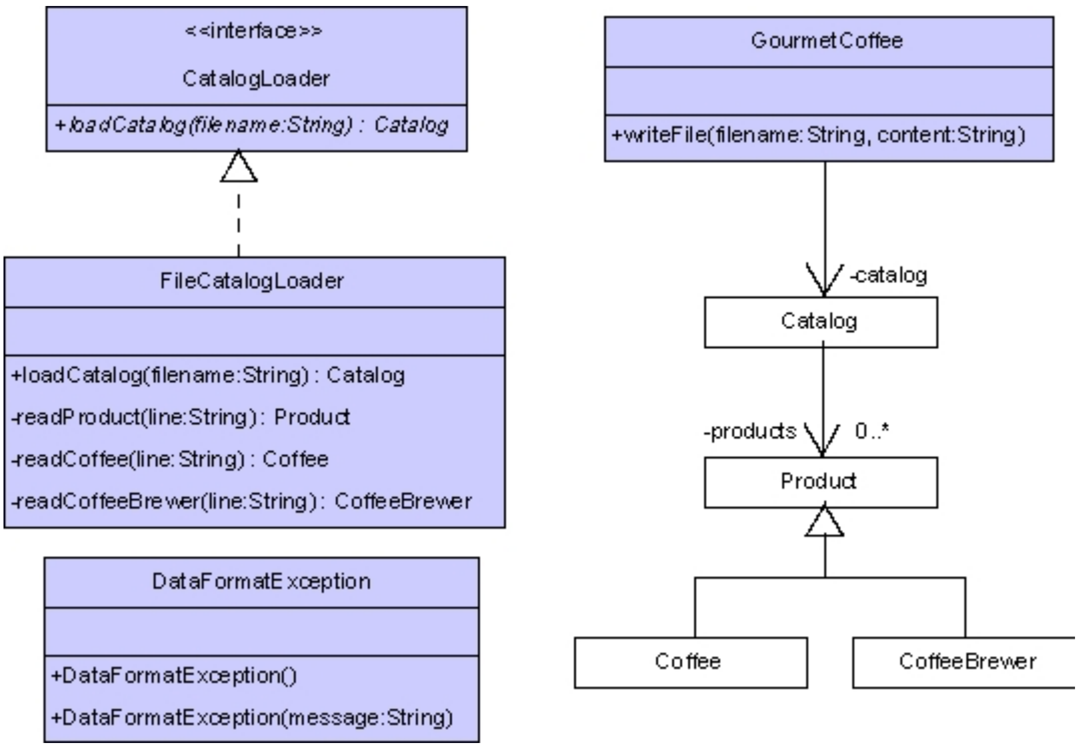
where:

- "Product" is a prefix that indicates the line type.
- *code* is a string that represents the code of the product.
- *description* is a string that represents the description of the product.

- *price* is a double that represents the price of the product.

The fields are delimited by an underscore (_). You can assume that the fields themselves do not contain any underscores.

The following class diagram highlights the elements you will use to load the product catalog and persist the sales information:



<<interface>>
CatalogLoader
+loadCatalog(filename:String) : Catalog
FileCatalogLoader
+loadCatalog(filename:String) : Catalog
-readProduct(line:String) : Product
-readCoffee(line:String) : Coffee
-readCoffeeBrewer(line:String) : CoffeeBrewer
DataFormatException
+DataFormatException()
+DataFormatException(message:String)
GourmetCoffee
+writeFile(filename:String, content:String)
-catalog
Catalog
-products
0..*
Product
Coffee
CoffeeBrewer

Figure 1 *Portion of Gourmet Coffee System class diagram*

In this assignment, you will implement FileCatalogloader and complete the implementation of GourmetCoffee.

Interface CatalogLoader

The interface CatalogLoader declares a method for producing a product catalog. A complete implementation of this interface is provided in the student archive.

Method:

- *Catalog loadCatalog(String fileName)*
- *throws FileNotFoundException,*
- *IOException,*
- *DataFormatException*

Loads the information in the specified file into a product catalog and returns the catalog.

Class DataFormatException

This exception is thrown when a line in the file being parsed has errors:

- The line does not have the expected number of tokens.
- The tokens that should contain numbers do not.

A complete implementation of this class is provided in the student archive.

Class `FileCatalogLoader` （total 40 points)

The class FileCatalogLoader implements interface CatalogLoader. It is used to obtain a product catalog from a file. You should implement this class from scratch:

Methods:

- *private Product readProduct(String line)*
- *throws DataFormatException* （**10 points**）

  This method reads a line of coffee-accessory data. It uses the class StringTokenizer to extract the accessory data in the specified line. If the line is error free, this method returns a Product object that encapsulates the accessory data. If the line has errors, that is, if it does not have the expected number of tokens or the token that should contain a double does not; this method throws a DataFormatException that contains the line of malformed data.
- *private Coffee readCoffee(String line))* （**10 points**）
- *throws DataFormatException*

  This method reads a line of coffee data. It uses the class StringTokenizer to extract the coffee data in the specified line. If the line is error free, this method returns a Coffee object that encapsulates the coffee data. If the line has errors, that is, if it does not have the expected number of tokens or the token that should contain a double does not; this method throws a DataFormatException that contains the line of malformed data.
- *private CoffeeBrewer readCoffeeBrewer(String line)* （**10 points**）
- *throws DataFormatException*

  This method reads a line of coffee-brewer data. It uses the class StringTokenizer to extract the brewer data in the specified line. If the line is error free, this method returns a CoffeeBrewer object that encapsulates the brewer data. If the line has errors, that is, if it does not have the expected number of tokens or the tokens that should contain a number do not; this method throws a DataFormatException that contains the line of malformed data.

1. *public Catalog loadCatalog(String filename)* （**10 points**）
2. *throws FileNotFoundException,*
3. *IOException,*
4. *DataFormatException*

   This method loads the information in the specified file into a product catalog and returns the catalog. It begins by opening the file for reading. It then proceeds to read and process each line in the file. The method String.startsWith is used to determine the line type:
   - If the line type is "Product", the method readProduct is invoked.
   - If the line type is "Coffee", the method readCoffee is invoked.
   - If the line type is "Brewer", the method readCoffeeBrewer is invoked.

After the line is processed, loadCatalog adds the product (accessory, coffee, or brewer) to the product catalog. When all the lines in the file have been processed, load returns the product catalog to the calling method.

This method can throw the following exceptions:

- FileNotFoundException — if the specified file does not exist.
- IOException — if there is an error reading the information in the specified file.

- DataFormatException — if a line in the file has errors (the exception should contain the line of malformed data).

Class `GourmetCoffee`

A partial implementation of class GourmetCoffee is provided in the student archive. You should implement writeFile, a method that writes sales information to a file:

- *private void writeFile(String fileName, String content)*
- *throws IOException*

  This method creates a new file with the specified name, writes the specified string to the file, and then closes the file.

Class `TestFileCatalogLoader`

This class is a test driver for FileCatalogLoader. A complete implementation is included in the student archive student-files.zip. You should use this class to test your implementation of FileCatalogLoader.

Files

The following files are needed to complete this assignment:

- student-files.zip — Download this file. This archive contains the following:
  - Class files
    - Coffee.class
    - CoffeeBrewer.class
    - Product.class
    - Catalog.class
    - OrderItem.class
    - Order.class
    - Sales.class
    - SalesFormatter.class
    - PlainTextSalesFormatter.class
    - HTMLSalesFormatter.class
    - XMLSalesFormatter.class
  - Documentation
    - Coffee.html
    - CoffeeBrewer.html
    - Product.html
    - Catalog.html
    - OrderItem.html
    - Order.html
    - Sales.html
    - SalesFormatter.html
    - PlainTextSalesFormatter.html
    - HTMLSalesFormatter.html
    - XMLSalesFormatter.html
  - Java files

        - CatalogLoader.java. A complete implementation
        - DataFormatException.java. A complete implementation
        - *TestFileCatalogLoader.java*. A complete implementation
        - GourmetCoffee.java. Use this template to complete your implementation.
    - Data files for the test driver
        - catalog.dat. A file with product information
        - empty.dat. An empty file

Tasks

Implement the class FileCatalogLoader and the method GourmetCoffee.writeFile. Document using Javadoc and follow Sun's code conventions. The following steps will guide you through this assignment. Work incrementally and test each increment. Save often.

1. **Extract** the files from student-files.zip
2. **Then**, implement class FileCatalogLoader from scratch**(40 points)** Use the TestFileCatalogLoader driver to test your implementation.
3. **Next**, implement the method GourmetCoffee.writeFile. **(10 points)**
4. **Finally**, compile the class GourmetCoffee, and execute the class GourmetCoffee by issuing the following command at the command prompt:

```
C:\>java GourmetCoffee catalog.dat
```

   Sales information has been hard-coded in the GourmetCoffee template provided by iCarnegie.

   - If the user displays the catalog, the output should be:

```
C001 Colombia, Whole, 1 lb
C002 Colombia, Ground, 1 lb
C003 Italian Roast, Whole, 1 lb
C004 Italian Roast, Ground, 1 lb
C005 French Roast, Whole, 1 lb
C006 French Roast, Ground, 1 lb
C007 Guatemala, Whole, 1 lb
C008 Guatemala, Ground, 1 lb
C009 Sumatra, Whole, 1 lb
C010 Sumatra, Ground, 1 lb
C011 Decaf Blend, Whole, 1 lb
C012 Decaf Blend, Ground, 1 lb
B001 Home Coffee Brewer
B002 Coffee Brewer, 2 Warmers
B003 Coffee Brewer, 3 Warmers
B004 Commercial Coffee, 20 Cups
B005 Commercial Coffee, 40 Cups
A001 Almond Flavored Syrup
A002 Irish Creme Flavored Syrup
A003 Mint Flavored syrup
A004 Caramel Flavored Syrup
```

```
A005 Gourmet Coffee Cookies
A006 Gourmet Coffee Travel Thermo
A007 Gourmet Coffee Ceramic Mug
A008 Gourmet Coffee 12 Cup Filters
A009 Gourmet Coffee 36 Cup Filters
```

- If the user saves the sales information in plain text, a file with the following content should be created:

```
------------------------
Order 1

5 C001 17.99

Total = 89.94999999999999
------------------------
Order 2

2 C002 18.75
2 A001 9.0

Total = 55.5
------------------------
Order 3

1 B002 200.0

Total = 200.0
```

- If the user saves the sales information in HTML, a file with the following content should be created:

```
<html>
  <body>
    <center><h2>Orders</h2></center>
    <hr>
    <h4>Total = 89.94999999999999</h4>
      <p>
        <b>code:</b> C001<br>
        <b>quantity:</b> 5<br>
        <b>price:</b> 17.99
      </p>
    <hr>
    <h4>Total = 55.5</h4>
      <p>
        <b>code:</b> C002<br>
        <b>quantity:</b> 2<br>
        <b>price:</b> 18.75
```

```
            </p>
            <p>
               <b>code:</b> A001<br>
               <b>quantity:</b> 2<br>
               <b>price:</b> 9.0
            </p>
         <hr>
         <h4>Total = 200.0</h4>
            <p>
               <b>code:</b> B002<br>
               <b>quantity:</b> 1<br>
               <b>price:</b> 200.0
            </p>
      </body>
   </html>
```

- If the user saves the sales information in XML, a file with the following content should be created:

```
<Sales>
   <Order total="89.94999999999999">
      <OrderItem quantity="5" price="17.99">C001</OrderItem>
   </Order>
   <Order total="55.5">
      <OrderItem quantity="2" price="18.75">C002</OrderItem>
      <OrderItem quantity="2" price="9.0">A001</OrderItem>
   </Order>
   <Order total="200.0">
      <OrderItem quantity="1" price="200.0">B002</OrderItem>
   </Order>
</Sales>
```

Submission

Upon completion, submit **only** the following:

1. FileCatalogLoader.java, FileCatalogLoader.class
2. GourmetCoffee.java , GourmetCoffee.class
3. a word file including the program running results

3.

Write a java program with the following requirements: **(total 80 points)**

1. Define a class ***Student*** **:** two attributes: name and grade, constructor and Getxxx methods, toString method. **(20 points)**

2. Define a class ***CradeComp implements Comparator<Student>*** to compare student grade. **(10 points)**
3. Define a ***StudentSystem*** class: **(total 50 points)**
    1) Define a ***ArrayList<Student>*** attribute.
    2) Define the ***public void display()*** method to display all students name and grade in the ArrayList. (10 points)
    3) Define the ***public float calculateScoreAverage ()*** method to calculate and return the average grade of the students . (10 points)
    4) Define the ***public void sortArray()*** method to sort the students in the ArrayList in ascending order of grades. (10 points)
    5) Define ***public void readFromFileToArrylist()*** method to read student information from file **StudentInfo.txt** into ArrayList. (10 points)
    6) Define the ***public void writeStudentFromFile()*** method to write the sorted students of ArrayList to the **StudentInfo1.txt** file. (10 points)
    7) main() is defined as follows:

```
public static void main(String[] args) {
     StudentSystem ss=new StudentSystem();
     ss.readFromFileToArrylist();
     ss.display();
     System.out.println(ss.calculateScoreAverage());
     ss.sortArray();
     ss.display();
     ss.writeStudentFromFile();
 }
```

The student information in **StudentInfo.txt** is as follows:

```
Mike_80
Jose_60
Tom_90
Rose_70
```

The student information written into the **StudentInfo1.txt** file after sorting is as follows:

```
Jose    60
Rose    70
Mike    80
Tom     90
```