面向对象程序设计

实验指导书

宋航　　刘国奇

东北大学软件学院

2012.9

目　录

前 言

面向对象的思想可以渗透到需求分析、系统建模、体系结构设计、程序设计与实现、系统测试等多个方面，它是描述现实世界复杂对象的相当直接而且直观的有效手段，对于提高系统质量、开发效率和代码重用率，都有明显的效果。

《面向对象程序设计》课程是软件工程专业的重要专业基础课程之一，该门课程注重实践性和实用性，主要通过面向对象程序设计思想和Java语言结合起来，让学生掌握面向对象程序设计思想，以及熟练使用Java语言进行面向对象的编程，因此学生不能满足于只听懂老师讲授的课堂内容，看懂书上的程序，应将课堂教学与实践环节紧密结合，使得学生加深对讲授内容的理解，学会上机调试程序。也就是善于发现程序中的错误，并且能很快地排除这些错误，使程序能正确运行。

《面向对象程序设计》是结合卡耐基梅隆大学的SSD3而形成的课程，该课程的教学体系和实验体系都很完整，并且东北大学软件学院也提供了良好的教学实验环境，希望同学们能够充分利用实验条件，认真完成实验，从实验中得到应有的锻炼和培养。

希望同学们在使用本实验指导书及进行实验的过程中，能够帮助我们不断地发现问题，并提出建议，使《面向对象程序设计》真正能够帮助同学们学习。

实验要求

《面向对象程序设计》课程实验的目的是为了使学生在课堂学习的同时，通过一系列的实验，使学生加深了解和更好地掌握《面向对象程序设计》课程教学大纲要求的内容。

在《面向对象程序设计》的课程实验过程中，要求学生做到：

（1）预习实验指导书有关部分，认真做好实验内容的准备，就实验可能出现的情况提前作出思考和分析。

（2）仔细观察调试程序过程中出现的各种问题，记录主要问题，作出必要说明和分析。

（3）遵守机房纪律，服从辅导教师指挥，爱护实验设备。

（4）实验课程不迟到，如有事不能出席，所缺实验一般不补。

（5）本实验采用的开发环境为Eclipse，同学在做实验之前要求熟悉该集成开发环境。

Experiment 1 Implementing the Gourmet Coffee System(4 Hours)

Prerequisites, Goals, and Outcomes

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

- *Object Oriented Programming*
    - Knowledge of class design
        - Class attributes
        - Constructors
        - Accessor methods
        - Mutator methods
    - Knowledge of inheritance
        - How to implement a specialization/generalization relationship using inheritance

Goals: Reinforce your ability to implement Java classes using inheritance.

Outcomes: You will demonstrate mastery of the following:

- Implementing the constructors, accessors, and mutators of a Java class
- Using inheritance to implement a specialization/generalization relationship

Background

This assignment asks you to implement some of the classes in the *Gourmet Coffee System* specified on Exercise 2.

Description

In this assessment, you will implement the classes and relationships illustrated in the following class diagram:

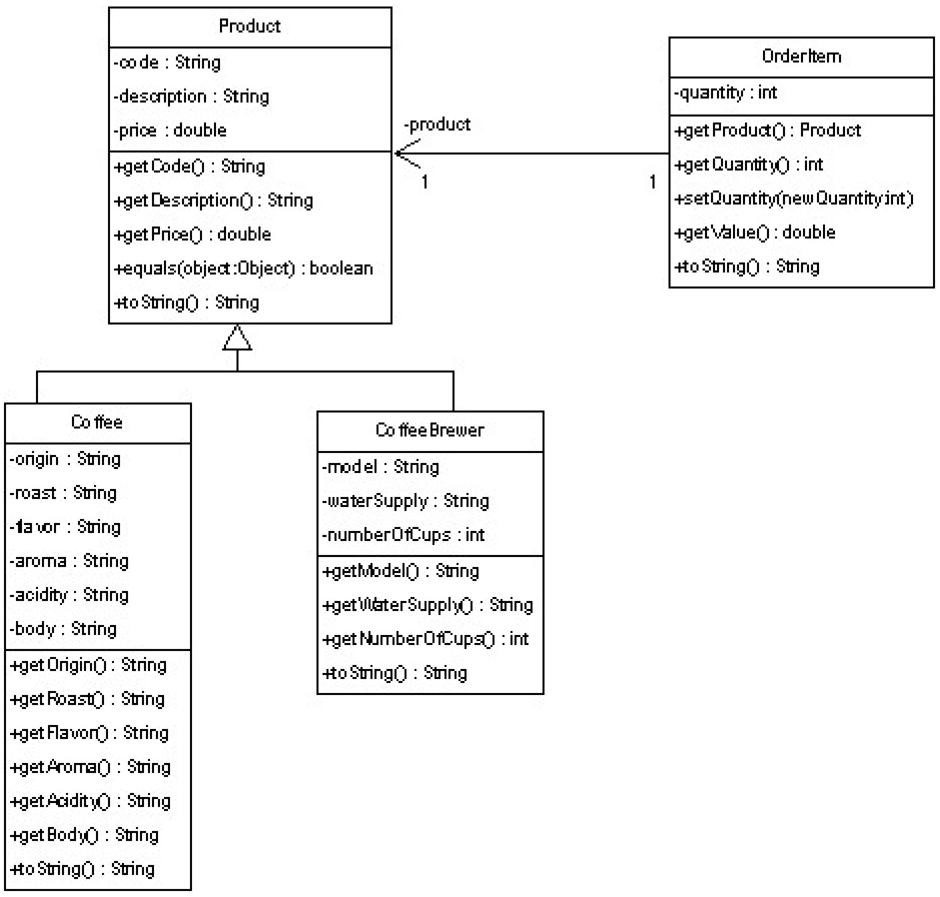


Figure 1 *Portion of Gourmet Coffee System class diagram*

The class specifications are as follows:

Class Product

The class Product models a generic product in the store.

Instance variables:

- *code*. The unique code that identifies the product
- *description*. A short description of the product
- *price*. The price of the product

Constructor and methods:

- *public Product(String initialCode,*
- *String initialDescription,*
- *double initialPrice)*

Constructor that initializes the instance variables code, description, and price.

- *public String getCode()*. Returns the value of instance variable code.
- *public String getDescription()*. Returns the value of instance variable description.
- *public double getPrice()*. Returns the value of instance variable price.
- *boolean equals(Object object)*. Overrides the method equals in the class Object. Two Product objects are equal if their codes are equal.
- *String toString()*. Overrides the method toString in the class Object. Returns the string representation of a Product object. The String returned has the following format:

  code_description_price

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

Class Coffee

The class Coffee models a coffee product. It extends class Product.

Instance variables:

- *origin*. The origin of the coffee
- *roast*. The roast of the coffee
- *flavor*. The flavor of the coffee
- *aroma*. The aroma of the coffee
- *acidity*. The acidity of the coffee
- *body*. The body of the coffee

Constructor and methods:

- *public Coffee(String initialCode,*
- *String initialDescription,*
- *double initialPrice,*
- *String initialOrigin,*
- *String initialRoast,*
- *String initialFlavor,*
- *String initialAroma,*
- *String initialAcidity,*
- *String initialBody)*

  Constructor that initializes the instance variables code, description, price, origin, roast, flavor, aroma, acidity, and body.

- *public String getOrigin()*. Returns the value of instance variable origin.
- *public String getRoast()*. Returns the value of instance variable roast.
- *public String getFlavor()*. Returns the value of instance variable flavor.
- *public String getAroma()*. Returns the value of instance variable aroma.
- *public String getAcidity()*. Returns the value of instance variable acidity.
- *public String getBody()*. Returns the value of instance variable body.
- *String toString()*. Overrides the method toString in the class Object. Returns the string representation of a Coffee object. The String returned has the following format:

  code_description_price_origin_roast_flavor_aroma_acidity_body

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

Class CoffeeBrewer

Class CoffeeBrewer models a coffee brewer. It extends class Product.

Instance variables:

- *model*. The model of the coffee brewer
- *waterSupply*. The water supply (*Pour-over* or *Automatic*)
- *numberOfCups*. The capacity of the coffee brewer

Constructor and methods:

- public CoffeeBrewer(String initialCode,
- String initialDescription,
- double initialPrice,
- String initialModel,
- String initialWaterSupply,
- int initialNumberOfCups)

  Constructor that initializes the instance variables code, description, price, model, waterSupply, and numberOfCups.

- *public String getModel()*. Returns the value of instance variable model.

- *public String getWaterSupply()*. Returns the value of instance variable waterSupply.

- *public int getNumberOfCups()*. Returns the value of instance variable numberOfCups.

- *String toString()*. Overrides the method toString in the class Object. Returns the string representation of a CoffeeBrewer object. The String returned has the following format:

  code_description_price_model_waterSupply_numberOfCups

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

Class OrderItem

Class OrderItem models an item in an order.

Instance variables:

- *product*. This instance variable represents the one-way association between OrderItem and Product. It contains a reference to a Product object.

- *quantity*. The quantity of the product in the order.

Constructor and methods:

- *public OrderItem(Product initialProduct,*
- *int initialQuantity)*

  Constructor that initializes the instance variables product and quantity.

- *public Product getProduct()*. Returns the value of the instance variable product, a reference to a Product object.

- *public int getQuantity()*. Returns the value of the instance variable quantity.

- *public void setQuantity(int newQuantity)*. Sets the instance variable quantity to the value of parameter newQuantity.

- *public double getValue()*. Returns the product of quantity and price.

- *String toString()*. Overrides the method toString in the class Object. Returns the string representation of an OrderItem object. The String representation has the following format:

  quantity product-code product-price

  The fields are separated by a space. You can assume that the fields themselves do not contain any spaces.

Test driver classes

Complete implementations of the following test drivers are provided in the student archive. Use these test drivers to verify that your code works correctly.

- Class TestProduct
- Class TestCoffee
- Class TestCoffeeBrewer
- Class TestOrderItem

Files

The following files are needed to complete this assignment:

- *student-files.zip* — Download this file. This archive contains the following:
  - *TestProduct.java*
  - *TestCoffee.java*
  - *TestCoffeeBrewer.java*
  - *TestOrderItem.java*

Tasks

Implement classes Product, Coffee, CoffeeBrewer, and OrderItem. 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 by issuing the following command at the command prompt:

   ```
   C:\>unzip student-files.zip
   ```

2. **Then**, implement class Product from scratch. Use TestProduct driver to test your implementation.

3. **Next**, implement class Coffee from scratch. Use TestCoffee driver to test your implementation.

4. **Then**, implement class `CoffeeBrewer` from scratch. Use `TestCoffeeBrewer` driver to test your implementation.

5. **Finally**, implement class `OrderItem` from scratch. Use `TestOrderItem` driver to test your implementation.

Submission

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

1. `Product.java`
2. `Coffee.java`
3. `CoffeeBrewer.java`
4. `OrderItem.java`

Experiment 2 Implementing the Collections in the Gourmet Coffee System (4 Hours)

Prerequisites, Goals, and Outcomes

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

- *Collections*
  - Use of class `ArrayList`
  - Use of iterators

Goals: Reinforce your ability to implement classes that use collections

Outcomes: You will demonstrate mastery of the following:

- Implementing a Java class that uses collections

Background

In this assignment, you will implement the classes in the *Gourmet Coffee System* that use collections.

Description

The following class diagram of the *Gourmet Coffee System* highlights the classes that use collections:

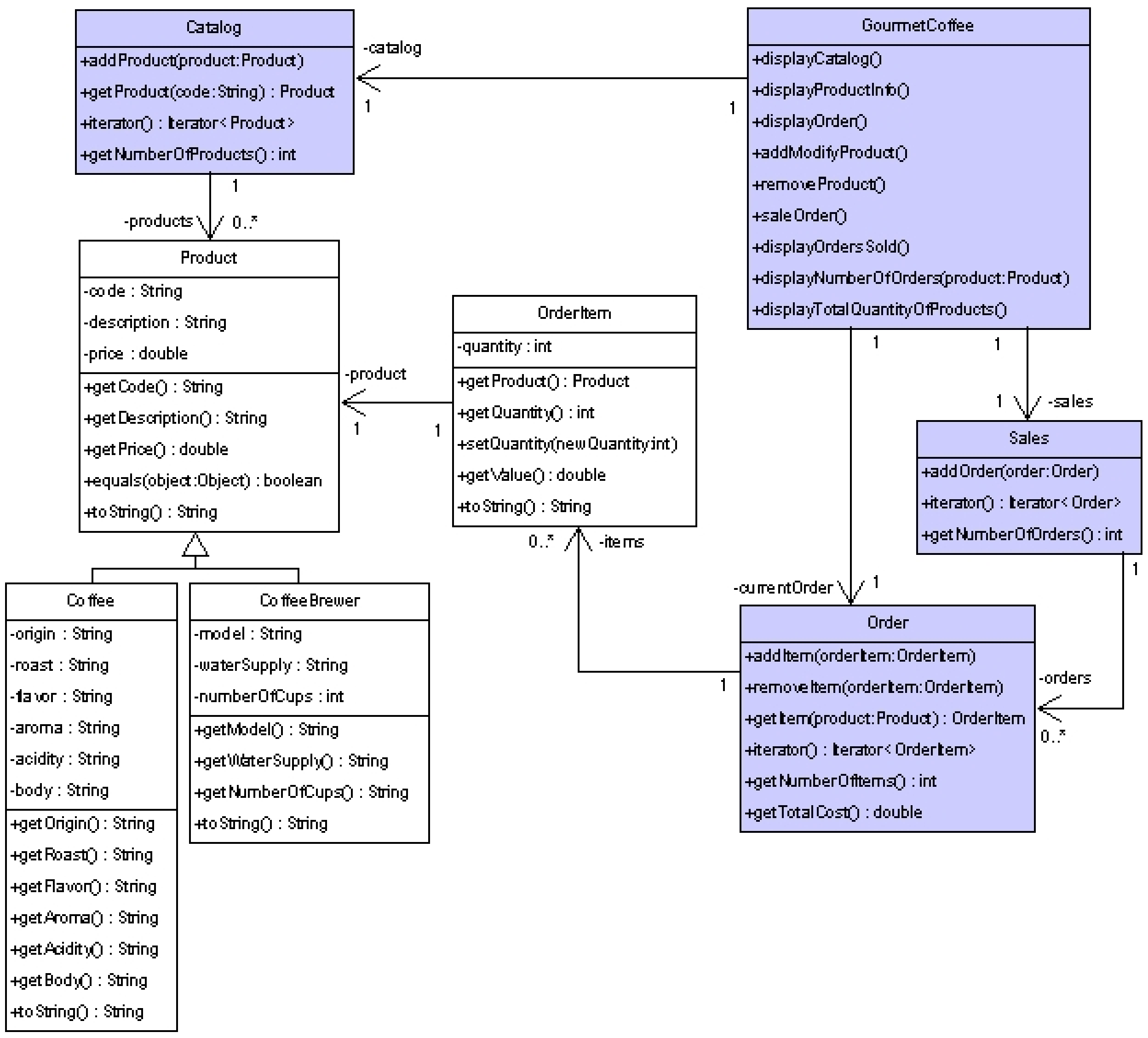


Figure 2 *Gourmet Coffee System class diagram*

Complete implementations of the following classes are provided in the student archive:

- `Coffee`
- `CoffeeBrewer`
- `Product`
- `OrderItem`
- `GourmetCoffee`

In this assignment, you will implement the following classes:

- `Catalog`
- `Order`
- `Sales`
- `GourmetCoffee`

The class specifications are as follows:

Class Catalog

The class Catalog models a product catalog. This class implements the interface Iterable<Product> to being able to iterate through the products using the for-each loop.

Instance variables:

- *products* — An ArrayList collection that contains references to instances of class Product.

Constructor and public methods:

- *public Catalog()* — Creates the collection products, which is initially empty.
- *public void addProduct(Product product)* — Adds the specified product to the collection products.
- *public Iterator<Product> iterator()* — Returns an iterator over the instances in the collection products.
- *public Product getProduct(String code)* — Returns a reference to the Product instance with the specified code. Returns null if there are no products in the catalog with the specified code.
- *public int getNumberOfProducts()* — Returns the number of instances in the collection products.

Class Order

The class Order maintains a list of order items. This class implements the interface Iterable<OrderItem> to being able to iterate through the items using the for-each loop.

Instance variables:

- *items* — An ArrayList collection that contains references to instances of class OrderItem.

Constructor and public methods:

- *public Order()* — Creates the collection items, which is initially empty.

- *public void addItem(OrderItem orderItem)* — Adds the specified order item to the collection `items`.
- *public void removeItem(OrderItem orderItem)* — Removes the specified order item from the collection `items`.
- *public Iterator<OrderItem> iterator()* — Returns an iterator over the instances in the collection `items`.
- *public OrderItem getItem(Product product)* — Returns a reference to the `OrderItem` instance with the specified product. Returns `null` if there are no items in the order with the specified product.
- *public int getNumberOfItems()* — Returns the number of instances in the collection `items`.
- *public double getTotalCost()* — Returns the total cost of the order.

Class `Sales`

The class `Sales` maintains a list of the orders that have been completed. This class implements the interface `Iterable<Order>` to being able to iterate through the orders using the `for-each` loop.

Instance variables:

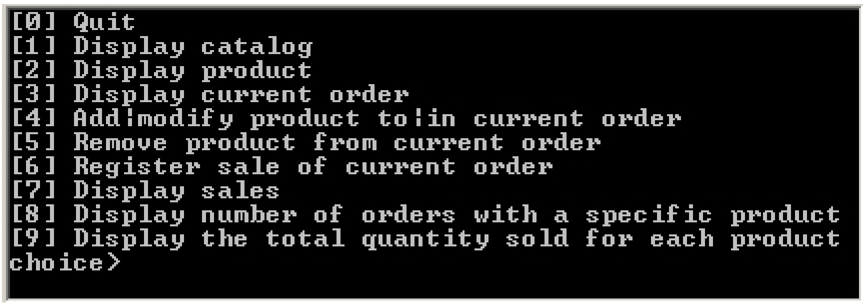
- *orders* — An `ArrayList` collection that contains references to instances of class `Order`.

Constructor and public methods:

- *public Sales()* — Creates the collection `orders`, which is initially empty.
- *public void addOrder(Order order)* — Adds the specified order to the collection `orders`.
- *public Iterator<Order> iterator()* — Returns an iterator over the instances in the collection `orders`.
- *public int getNumberOfOrders()* — Returns the number of instances in the collection `orders`.

Class `GourmetCoffee`

The class `GourmetCoffee` creates a console interface to process store orders. Currently, it includes the complete implementation of some of the methods. The methods displayNumberOfOrders and displayTotalQuantityOfProducts are incomplete and should be implemented. The following is a screen shot of the interface:



```
[0] Quit
[1] Display catalog
[2] Display product
[3] Display current order
[4] Add|modify product to|in current order
[5] Remove product from current order
[6] Register sale of current order
[7] Display sales
[8] Display number of orders with a specific product
[9] Display the total quantity sold for each product
choice>
```

Figure 3 *Execution of GourmetCoffee*

Instance variables:

- *catalog* — A Catalog object with the products that can be sold.
- *currentOrder* — An Order object with the information about the current order.
- *sales* — A Sales object with information about all the orders sold by the store.

Constructor and public methods:

- *public GourmetCoffeeSolution()* — Initializes the attributes catalog, currentOrder and sales. This constructor is complete and should not be modified.

- *public void displayCatalog* — Displays the catalog. This method is complete and should not be modified.

- *public void displayProductInfo()* — Prompts the user for a product code and displays information about the specified product. This method is complete and should not be modified.

- *public void displayOrder()* — Displays the products in the current order. This method is complete and should not be modified.

- *public void addModifyProduct()* — Prompts the user for a product code and quantity. If the specified product is not already part of the order, it is added; otherwise, the quantity of the product is updated. This method is complete and should not be modified.

- *public void removeProduct()* — Prompts the user for a product code and removes the specified product from the current order. This method is complete and should not be modified.

- *public void saleOrder()* — Registers the sale of the current order. This method is complete and should not be modified.

- *public void displayOrdersSold()* — Displays the orders that have been sold. This method is complete and should not be modified.

- *public void displayNumberOfOrders(Product product)* — Displays the number of orders that contain the specified product. This method is incomplete and should be implemented.

- *public void displayTotalQuantityOfProducts()* — Displays the total quantity sold for each product in the catalog. This method is incomplete and should be implemented.

Test driver classes

Complete implementations of the following test drivers are provided in the student archive:

- Class `TestCatalog`
- Class `TestOrder`
- Class `TestSales`

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*
    - *OrderItem.class*
  - Documentation
    - *Coffee.html*
    - *CoffeeBrewer.html*
    - *Product.html*
    - *OrderItem.html*
  - Java files
    - *GourmetCoffee.java* — An incomplete implementation.
    - *TestCatalog.java* — A complete implementation.
    - *TestOrder.java* — A complete implementation.
    - *TestSales.java* — A complete implementation.

Tasks

Implement classes `Catalog`, `Order`, and `Sales`. Document your code 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 by issuing the following command at the command prompt.

   ```
   C:\>unzip student-files.zip
   ```

2. **Then**, implement class `Catalog` from scratch. Use `TestCatalog` to test your implementation.

3. **Next**, implement class `Order` from scratch. Use `TestOrder` to test your implementation.

5. **Then**, implement class `Sales` from scratch. Use `TestSales` to test your implementation.

6. **Finally**, complete class `GourmetCoffee`. It uses a `Catalog` object created by method `GourmetCoffee.loadCatalog` and a `Sales` object generated by method `GourmetCoffee.loadSales`. To complete class `GourmetCoffee`, implement the following methods:

   - *public void displayNumberOfOrders(Product product)* — This method displays the number of orders in the `sales` object that contain the specified product. Compile and execute the class `GourmetCoffee`. Verify that the method *displayNumberOfOrders* works correctly. The following is the output that should be displayed for the product with product code `A001` and the orders preloaded by the method `loadSales`:

     ```
     [0] Quit
     [1] Display catalog
     [2] Display product
     [3] Display current order
     [4] Add|modify product to|in current order
     [5] Remove product from current order
     [6] Register sale of current order
     [7] Display sales
     [8] Display number of orders with a specific product
     [9] Display the total quantity sold for each product
     choice> 8

     Product code> A001
     Number of orders that contains the product A001: 4
     ```

   - *public void displayTotalQuantityOfProducts()* — This method displays the total quantity sold for each product in the catalog. The information of each product must be displayed on a single line in the following format:

ProductCode Quantity

The following is a description of the information included in the format above:

- *ProductCode* — the code of the product
- *Quantity* — the total quantity of product that has been sold in the store

Compile and execute the class `GourmetCoffee`. Verify that the method `displayTotalQuantityOfProducts` works correctly. The following is the output that should be displayed for the orders preloaded by the method `loadSales`:

```
[0] Quit
[1] Display catalog
[2] Display product
[3] Display current order
[4] Add|modify product to|in current order
[5] Remove product from current order
[6] Register sale of current order
[7] Display sales
[8] Display number of orders with a specific product
[9] Display the total quantity sold for each product
choice> 9

C001 9
C002 4
C003 5
C004 0
C005 8
B001 2
B002 1
B003 2
A001 12
A002 6
A003 5
A004 6
A005 0
```

Submission

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

1. `Catalog.java`
2. `Order.java`

3. Sales.java
4. GourmetCoffee.java

Experiment 3 Using Design Patterns in the Gourmet Coffee System (2 Hours)

Prerequisites, Goals, and Outcomes

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

- *Object-oriented Programming*
    - How to define interfaces
    - How to implement interfaces

- *Design Patterns*:
    - Knowledge of the singleton pattern
    - Knowledge of the strategy pattern

Goals: Reinforce your ability to use the singleton and strategy patterns

Outcomes: You will demonstrate mastery in the following:

- Producing applications that use the singleton pattern
- Producing applications that use the strategy pattern

Background

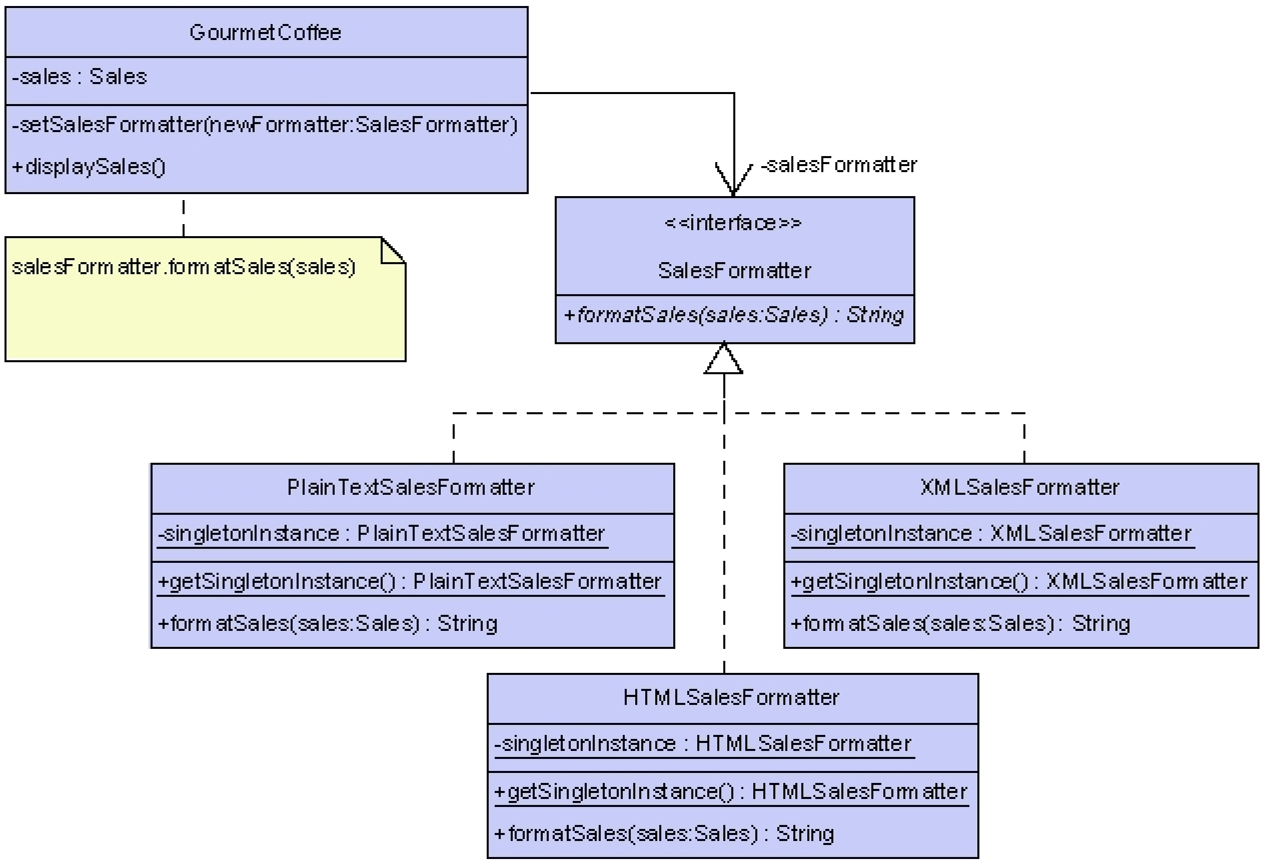
In this assignment, you will create another version of the *Gourmet Coffee System*. This version will present the user with four choices:

```
[0] Quit
[1] Display sales (Plain Text)
[2] Display sales (HTML)
[3] Display sales (XML)
choice>
```

The user will be able to display the sales information in 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 formats the sales information. This code will use the singleton and strategy patterns.

Description

The following class diagram shows how the singleton and strategy pattern will be used in your implementation:



GourmetCoffee
-sales : Sales
-setSalesFormatter(newFormatter:SalesFormatter)
+displaySales()
salesFormatter.formatSales(sales)
-salesFormatter
<<interface>>
SalesFormatter
+formatSales(sales:Sales) : String
PlainTextSalesFormatter
-singletonInstance : PlainTextSalesFormatter
+getSingletonInstance() : PlainTextSalesFormatter
+formatSales(sales:Sales) : String
XMLSalesFormatter
-singletonInstance : XMLSalesFormatter
+getSingletonInstance() : XMLSalesFormatter
+formatSales(sales:Sales): String
HTMLSalesFormatter
-singletonInstance : HTMLSalesFormatter
+getSingletonInstance() : HTMLSalesFormatter
+formatSales(sales:Sales) : String

Figure 4 *Portion of Gourmet Coffee System class diagram*

The elements of the pattern are:

- Interface SalesFormatter declares a method called formatSales that produces a string representation of the sales information.
- Class PlainTextSalesFormatter implements formatSales. Its version returns the sales information in a plain-text format.
- Class HTMLSalesFormatter implements formatSales. Its version returns the sales information in an HTML format.
- Class XMLSalesFormatter implements formatSales. Its version returns the sales information in an XML format.
- Class GourmetCoffee is the context class. It also contains client code. The client code calls:
    - Method GourmetCoffee. setSalesFormatter to change the current formatter
    - Method GourmetCoffee. displaySales to display the sales information using the current formatter

In this assignment, you should implement the following interface and classes:

- SalesFormatter
- PlainTextSalesFormatter

- HTMLSalesFormatter
- XMLSalesFormatter
- GourmetCoffee (a partial implementation is provided in the student archive)

Complete implementations of the following classes are provided in the student archive:

- Coffee
- CoffeeBrewer
- Product
- Catalog
- OrderItem
- Order
- Sales

Interface SalesFormatter

Interface SalesFormatter declares the method that every "Formatter" class will implement.

Method:

- *public String formatSales(Sales sales)*. Produces a string representation of the sales information.

Class PlainTextSalesFormatter

Class PlainTextSalesFormatter implements the interface SalesFormatter. This class is implemented as a singleton so a new object will not be created every time the plain-text format is used.

Static variable:

- *singletonInstance*. The single instance of class PlainTextSalesFormatter.

Constructor and methods:

- *static public PlainTextSalesFormatter getSingletonInstance()*. Static method that obtains the single instance of class PlainTextsalesFormatter.

- *private PlainTextSalesFormatter()*. Constructor that is declared private so it is inaccessible to other classes. A private constructor makes it impossible for any other class to create an instance of class PlainTextSalesFormatter.

- *public String formatSales(Sales sales)*. Produces a string that contains the specified sales information in a plain-text format. Each order in the sales information has the following format:
- ------------------------
- Order *number*
-
- *quantity1 code1 price1*
- *quantity2 code2 price2*
- ...
- *quantityN codeN priceN*
-

  Total = *totalCost*

  where

  - *number* is the order number.
  - *quantityX* is the quantity of the product.
  - *codeX* is the code of the product.
  - *priceX* is the price of the product.
  - *totalCost* is the total cost of the order.

  Each order should begin with a dashed line. The first order in the sales information should be given an order number of 1, the second should be given an order number of 2, and so on.

Class HTMLSalesFormatter

Class HTMLSalesFormatter implements the interface SalesFormatter. This class is implemented as a singleton so a new object will not be created every time the HTML format is used.

Static variable:

- *singletonInstance*. The single instance of class HTMLSalesFormatter.

Constructor and methods:

- *static public HTMLSalesFormatter getSingletonInstance()*. Static method that obtains the single instance of class HTMLSalesFormatter.

- *private HTMLSalesFormatter()*. Constructor that is declared private so it is inaccessible to other classes. A private constructor makes it impossible for any other class to create an instance of class HTMLSalesFormatter.

- *public String formatSales(Sales sales)*. Produces a string that contains the specified sales information in an HTML format.

- The string should begin with the following HTML:
- `<html>`
- `<body>`
  `<center><h2>Orders</h2></center>`

- Each order in the sales information should begin with horizontal line, that is, an `<hr>` tag.
- Each order in the sales information should have the following format:

```
<hr>
<h4>Total = totalCost</h4>
  <p>
    <b>code:</b> code1<br>
    <b>quantity:</b> quantity1<br>
    <b>price:</b> price1
 </p>
      ...
 <p>
    <b>code:</b> codeN<br>
    <b>quantity:</b> quantityN<br>
    <b>price:</b> priceN
 </p>
```

where:

- *quantityX* is the quantity of the product.
- *codeX* is the code of the product.
- *priceX* is the price of the product.
- *totalCost* is the total cost of the order.

- The string should end with the following HTML:

```
  </body>
</html>
```

Class XMLSalesFormatter

Class `XMLSalesFormatter` implements the interface `SalesFormatter`. This class is implemented as a singleton so a new object will not be created every time the XML format is used.

Static variable:

- *singletonInstance*. The single instance of class `XMLSalesFormatter`.

Constructor and methods:

- *static public XMLSalesFormatter getSingletonInstance()*. Static method that obtains the single instance of class XMLSalesFormatter.

- *private XMLSalesFormatter()*. Constructor that is declared private so it is inaccessible to other classes. A private constructor makes it impossible for any other class to create an instance of class XMLSalesFormatter.

- *public String formatSales(Sales sales)*. Produces a string that contains the specified sales information in an XML format.

  - The string should begin with the following XML:

    ```
    <Sales>
    ```

  - Each order in the sales information should have the following format:

  -
    ```
      <Order total="totalCost">
        <OrderItem quantity="quantity1"
    price="price1">code1</OrderItem>
        ...
        <OrderItem quantity="quantityN"
    price="priceN">codeN</OrderItem>
      </Order>
    ```

    where:

    - *quantityX* is the quantity of the product.
    - *codeX* is the code of the product.
    - *priceX* is the price of the product.
    - *totalCost* is the total cost of the order.

  - The string should end with the following XML:

    ```
    </Sales>
    ```

Class GourmetCoffee

Class GourmetCoffee lets the user display the sales information in one of three formats: plain text, HTML, or XML. A partial implementation of this class is provided in the student archive.

Instance variables:

- *private Sales sales*. A list of the orders that have been paid for.

- *private SalesFormatter salesFormatter*. A reference variable that refers to the current formatter: a PlainTextSalesFormatter, HTMLSalesFormatter, or XMLSalesFormatter object.

Constructor and methods:

The following methods and constructor are complete and require no modification:

- *public static void main(String[] args) throws IOException*. Starts the application.

- *private GourmetCoffee()*. Initialize instance variables sales and salesFormatter.

- *private Catalog loadCatalog()*.Populates the product catalog.

- *private void loadSales(Catalog catalog)*. Populates the sales object.

- *private int getChoice() throws IOException*. Displays a menu of options and verifies the user's choice.

The following methods should be completed:

- *private void setSalesFormatter(SalesFormatter newFormatter)*. Changes the current formatter by updating the instance variable salesFormatter with the object specified in the parameter newFormatter.

- *private void displaySales()*. Displays the sales information in the standard output using the method salesFormatter. formatSales to obtain the sales information in the current format.

- *private void run() throws IOException*. Presents the user with a menu of options and executes the selected task

    - If the user chooses option 1, run calls method setSalesFormatter with the singleton instance of class PlainTextSalesFormatter, and calls method displaySales to display the sales information in the standard output.

    - If the user chooses option 2, run calls method setSalesFormatter with the singleton instance of class HTMLSalesFormatter, and calls method displaySales to display the sales information in the standard output.

    - If the user chooses option 3, run calls method setSalesFormatter with the singleton instance of class XMLTextSalesFormatter, and calls method displaySales to display the sales information in the standard output.

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*
  - Documentation
    - *Coffee.html*
    - *CoffeeBrewer.html*
    - *Product.html*
    - *Catalog.html*
    - *OrderItem.html*
    - *Order.html*
    - *Sales.html*
  - *GourmetCoffee.java*. A partial implementation of the class GourmetCoffee.

Tasks

Implement the interface SalesFormatter and the classes PlainTextSalesFormatter, HTMLSalesFormatter, XMLSalesFormatter. Finish the implementation of class GourmetCoffee. 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 by issuing the following command at the command prompt:

   ```
   C:\>unzip student-files.zip
   ```

2. **Then**, implement interface SalesFormatter from scratch.

3. **Next**, implement class PlainTextSalesFormatter from scratch.

4. **Then**, implement class HTMLSalesFormatter from scratch.

5. **Next**, implement class XMLSalesFormatter from scratch.

6. **Then**, complete the method GourmetCoffee.setSalesFormatter.

7. **Next**, complete the method `GourmetCoffee.displaySales`.

8. **Then**, complete the method `GourmetCoffee.run`.

9. **Finally**, compile and execute the class `GourmetCoffee`. Sales information has been hard-coded in the `GourmetCoffee` template provided by iCarnegie.
   - If the user chooses to display the sales information in plain text, the output should be:

```
o      ------------------------
o      Order 1
o
o      5 C001 17.99
o
o      Total = 89.94999999999999
o      ------------------------
o      Order 2
o
o      2 C002 18.75
o      2 A001 9.0
o
o      Total = 55.5
o      ------------------------
o      Order 3
o
o      1 B002 200.0
o
   Total = 200.0
```

   - If the user chooses to display the sales information in HTML, the output should be:

```
o      <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 chooses to display the sales information in XML, the output should be:
-
```
   <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. SalesFormatter.java
2. PlainTextSalesFormatter.java
3. HTMLSalesFormatter.java
4. XMLSalesFormatter.java
5. GourmetCoffee.java

Experiment 4 Using File I/O in the Gourmet Coffee System (2 Hours)

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:

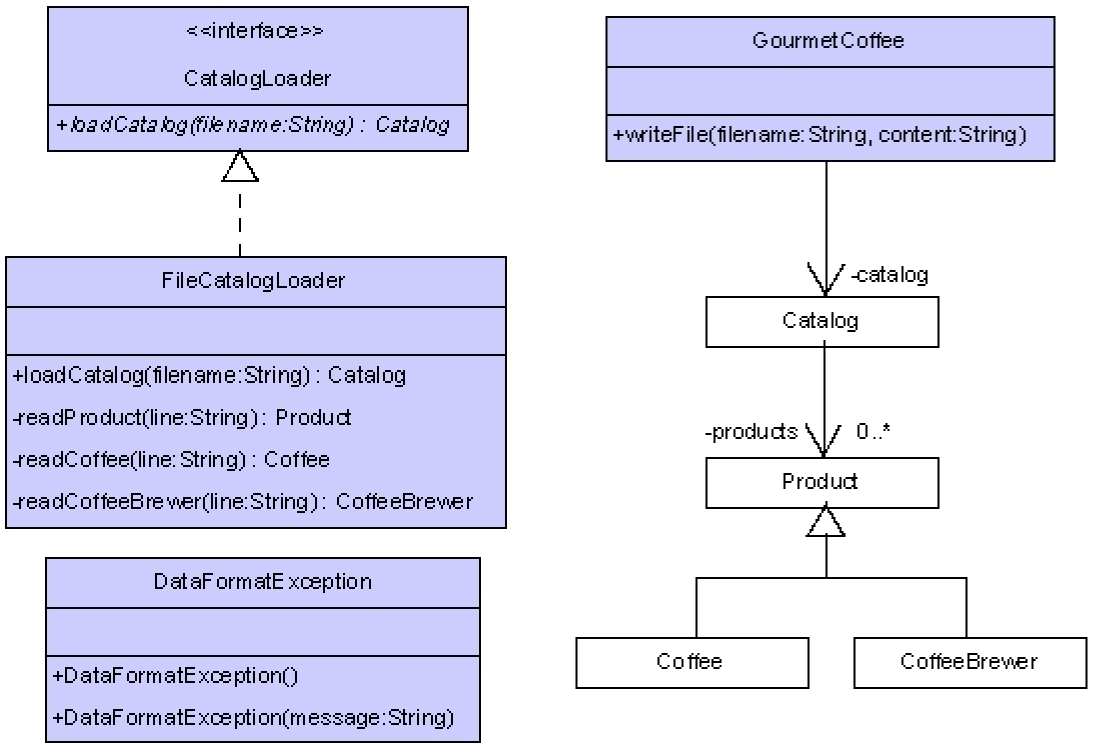
Product_*code*_*description*_*price*

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 5 *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`

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*

  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)*
- *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)*
- *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.

- *public Catalog loadCatalog(String filename)*
- *throws FileNotFoundException,*
- *IOException,*
- *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 by issuing the following command at the command prompt:

```
C:\>unzip student-files.zip
```

2. **Then**, implement class FileCatalogLoader from scratch. Use the TestFileCatalogLoader driver to test your implementation.

3. **Next**, implement the method GourmetCoffee.writeFile.

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:

   -
     ```
     ------------------------
     ```

```
o     Order 1
o
o     5 C001 17.99
o
o     Total = 89.94999999999999
o     ------------------------
o     Order 2
o
o     2 C002 18.75
o     2 A001 9.0
o
o     Total = 55.5
o     ------------------------
o     Order 3
o
o     1 B002 200.0
o
   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
2. GourmetCoffee.java

Experiment 5 Implementing a GUI for the Gourmet Coffee System (Part I) (2 Hours)

Prerequisites, Goals, and Outcomes

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

- *Graphical user interface (GUI)*
    - Knowledge of Swing components and containers

Goals: Reinforce your ability to create a GUI using Swing

Outcomes: You will master the following skills:

- Produce applications that use a Swing GUI

Background

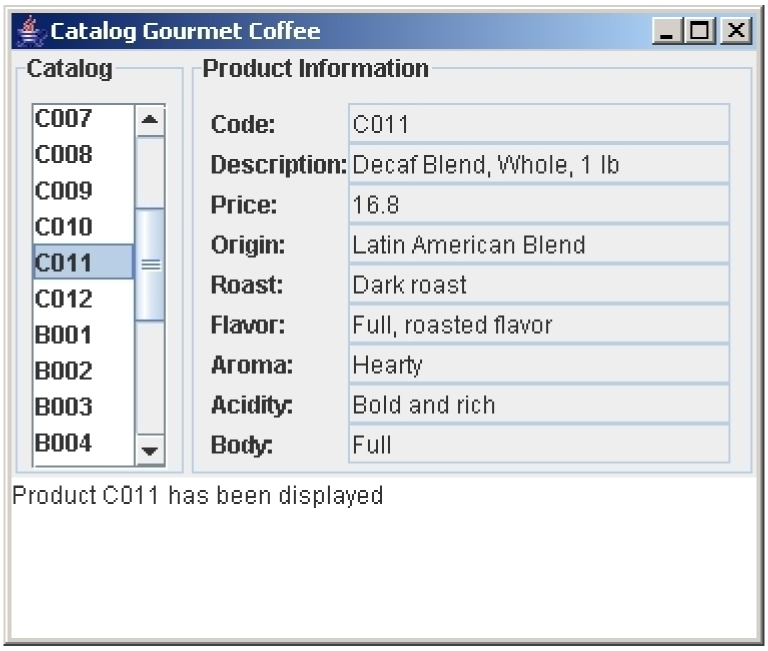
In this assignment, you will create a GUI that displays the *Gourmet Coffee System*'s product catalog. Part of the work has been done for you and is provided in the student archive. You will complete the code that creates a graphical presentation of the product details.

Description

Class `CatalogGUI` lets the user display the product details of every product in the gourmet coffee store's product catalog. This simple GUI contains the following components:

- a `JList` that displays the product code of every product in the catalog
- a `JPanel` that presents product details
- a `JTextArea` that serves as a status area

To examine the details of a particular product, the user selects the product code in the list. The application responds by displaying the product details in the `JPanel` and a status message in the `JTextArea`.



Code:	C011
Description:	Decaf Blend, Whole, 1 lb
Price:	16.8
Origin:	Latin American Blend
Roast:	Dark roast
Flavor:	Full, roasted flavor
Aroma:	Hearty
Acidity:	Bold and rich
Body:	Full

```
Product C011 has been displayed
```

Figure 6 *Execution of CatalogGUI*

The following class diagram highlights the classes used to implement the GUI:

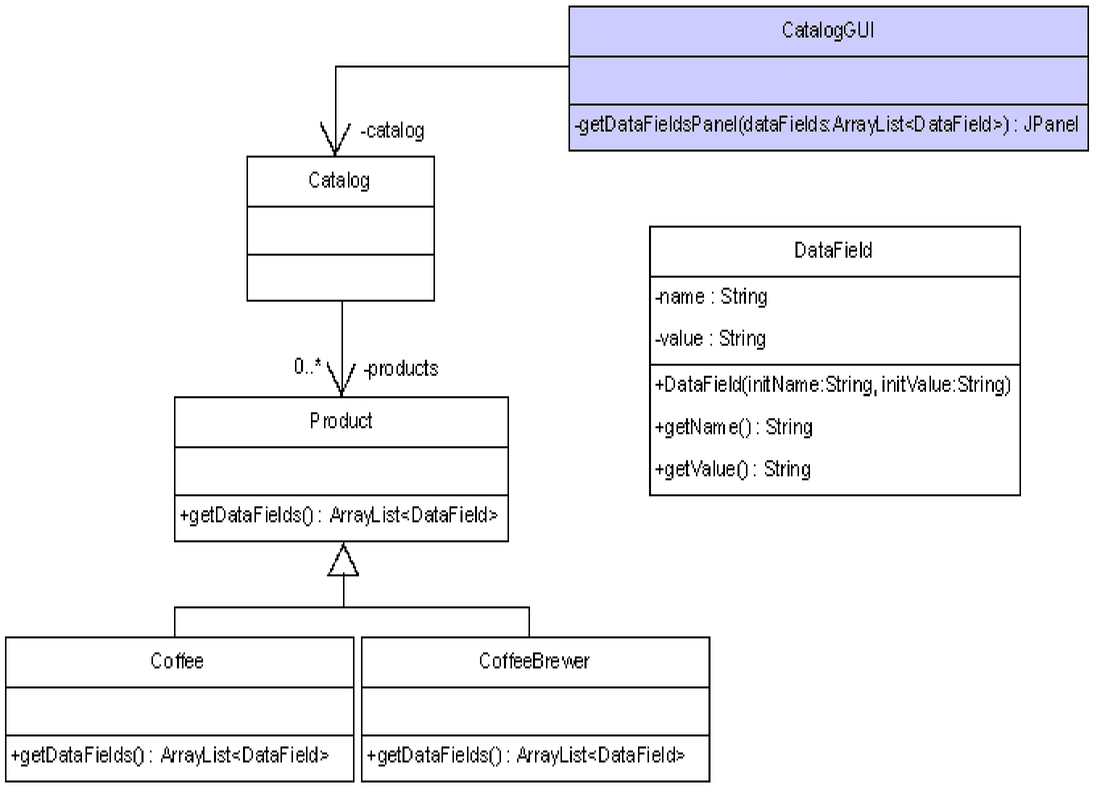

CatalogGUI
-getDataFieldsPanel(dataFields:ArrayList<DataField>) : JPanel
-catalog
Catalog
0..*
-products
Product
+getDataFields() : ArrayList<DataField>
DataField
-name : String
-value : String
+DataField(initName:String, initValue:String)
+getName() : String
+getValue() : String
Coffee
+getDataFields() : ArrayList<DataField>
CoffeeBrewer
+getDataFields() : ArrayList<DataField>

Figure 7 *Portion of Gourmet Coffee System class diagram*

Class CatalogGUI instantiates the Swing components, arranges the components in a window, and handles the events generated by the list. In this assignment, you will implement the method getDataFieldsPanel that returns a JPanel displaying the product details. An incomplete implementation of CatalogGUI is provided in the student archive.

Class DataField contains a name/value pair that represents a single attribute that is stored within an object. A complete implementation of DataField is provided in the student archive.

Classes Product, Coffee, and CoffeeBrewer have been enhanced to add a method called getDataFields that returns an ArrayList of DataField objects with the product details:

- In class Product, the method getDataFields returns an ArrayList with DataField objects for the attributes "code", "description" and "price".
- In class Coffee, the method getDataFields returns an ArrayList with DataField objects for the attributes "code", "description", "price", "origin", "roast", "flavor", "aroma", "acidity" and "body".
- In class CoffeeBrewer, the method getDataFields returns an ArrayList with DataField for the attributes "code", "description", "price", "model", "waterSupply" and "numberOfCups".

A complete implementation of classes Product, Coffee, and CoffeeBrewer are provided in the student archive.

Class Catalog has been enhanced: it now contains a method called getCodes that returns an array of product codes (all the product codes in the product catalog) which CatalogGUI uses to populate the JList. A complete implementation of Catalog is provided in the student archive.

Files

The following files are needed to complete this assignment:

- *student-files.zip* — Download this file. This archive contains the following:
  - Class files
    - *Product.class*
    - *Coffee.class*
    - *CoffeeBrewer.class*
    - *Catalog.class*
    - *CatalogLoader.class*
    - *FileCatalogLoader.class*
    - *DataFormatException.class*
    - *DataField.class*
  - Documentation

        - *Product.html*
        - *Coffee.html*
        - *CoffeeBrewer.html*
        - *Catalog.html*
        - *CatalogLoader.html*
        - *FileCatalogLoader.html*
        - *DataFormatException.html*
        - *DataField.html*
    - Java files
        - *CatalogGUI.java*— a complete implementation
    - Data files for the test driver
        - *catalog.dat* — a file with product information for every product in the product catalog

Tasks

The following steps will guide you through this assignment. Work incrementally and test each increment. Follow Sun's code conventions. Save often.

1. **Extract** the files by issuing the following command at the command prompt:

   ```
   C:\>unzip student-files.zip
   ```

2. **Then**, complete the implementation of the method `getDataFieldsPanel` in the class `CatalogGUI`.

   `private JPanel getDataFieldsPanel(ArrayList<DataField> dataFields)` — Returns a reference to a `JPanel` object that shows the names and values of the `DataField` objects stored in the parameter `dataFields`:

        - For a coffee product, `dataFields` contains nine `DataField` objects with the names: "Code", "Description", "Price", "Origin", "Roast", "Flavor", "Aroma", "Acidity" and "Body".

        - For a coffee brewer, `dataFields` contains six `DataField` objects with the names: "Code", "Description", "Price", "Model", "Water supply" and "Number of cups".

        - For a generic product, `dataFields` contains three `DataField` objects with the names: "Code", "Description" and "Price".

   For each `DataField` object stored in `dataFields`, the `JPanel` should contain a `JLabel` object with the name of the attribute and an uneditable `JTextField` object with the value of the attribute. Use the `JPanel` layering technique to build the `JPanel`. The arrangement of the `JPanel` need not match Figure 1 exactly; any well-organized presentation of the attributes will be acceptable.

3. **Finally**, compile and execute the class `CatalogGUI`.

Submission

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

1. `CatalogGUI.java`

Experiment 6 Implementing a GUI for the Gourmet Coffee System (Part II) (2 Hours)

Prerequisites, Goals, and Outcomes

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

- *Graphical user interface*
  - Knowledge of Swing components and containers
  - Knowledge of Swing event handling
  - Knowledge of `JFileChooser` dialog

Goals: Reinforce your ability to create a GUI with event handling

Outcomes: You will master the following skills:

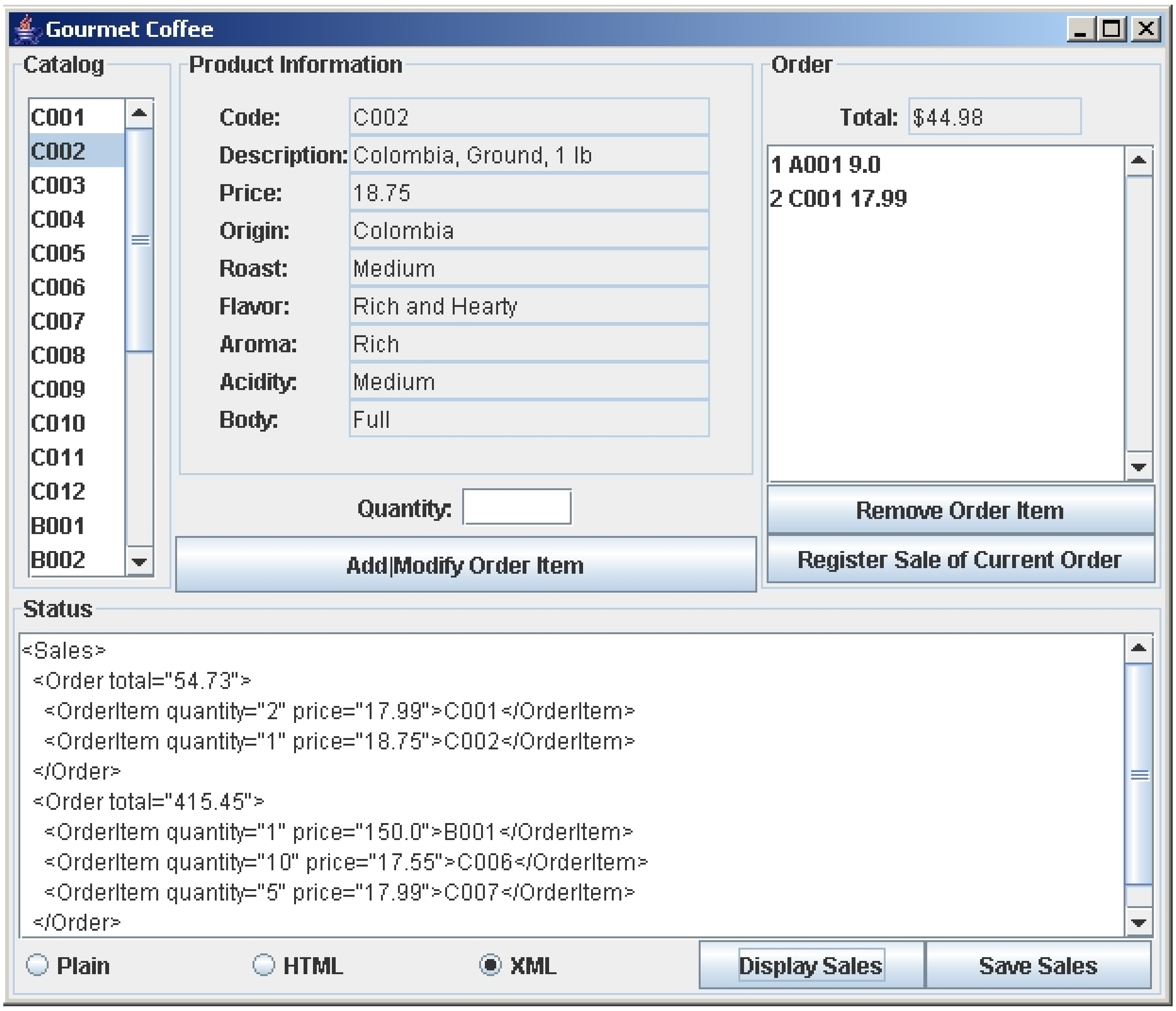
- Produce interactive applications that use a Swing GUI

Background

In this assignment, you will create a comprehensive GUI for the *Gourmet Coffee System*. Part of the work has been done for you and is provided in the student archive. You will implement the code that handles the button events.

Description

Class `GourmetCoffeeGUI` implements a GUI for the *Gourmet Coffee System*. This GUI lets the user display the product catalog, process orders, and track the store's sales:



C001
C002
C003
C004
C005
C006
C007
C008
C009
C010
C011
C012
B001
B002

Code:	C002
Description:	Colombia, Ground, 1 lb
Price:	18.75
Origin:	Colombia
Roast:	Medium
Flavor:	Rich and Hearty
Aroma:	Rich
Acidity:	Medium
Body:	Full

Quantity:

Total: $44.98

```
1 A001 9.0
2 C001 17.99
```

```
<Sales>
  <Order total="54.73">
    <OrderItem quantity="2" price="17.99">C001</OrderItem>
    <OrderItem quantity="1" price="18.75">C002</OrderItem>
  </Order>
  <Order total="415.45">
    <OrderItem quantity="1" price="150.0">B001</OrderItem>
    <OrderItem quantity="10" price="17.55">C006</OrderItem>
    <OrderItem quantity="5" price="17.99">C007</OrderItem>
  </Order>
```

Figure 8 *Execution of GourmetCoffeeGUI*

The GUI is divided into four panels: Catalog, Product Information, Order, and Status:

- The Catalog panel contains a list of product codes.
- The Product Information panel contains product details. Positioned immediately beneath this panel are a text field to specify the quantity of the selected product and a button to add the selected product to the current order.
- The Order panel contains the current order, its cost, and buttons to edit the order and register its sale.
- The Status panel contains a text area to display messages and sales information; command buttons to display and save sales information; and radio buttons to specify the format of the sales information when it is displayed or saved.

When a user clicks one of the following command buttons, `GourmetCoffeeGUI` handles the associated event as described below:

- *addModifyButton*. Adds the selected product to the current order. If the selected product is already part of the order, modifies the quantity of that product in the order.
- *removeButton*. Removes the selected item from the current order.
- *registerSaleButton*. Adds the current order to the store's sales and empties the current order.
- *displaySalesButton*. Lists all orders that have been sold in the specified format (plain text, HTML, or XML).
- *saveSalesButton*. Saves all orders that have been sold in a file with the specified format (plain text, HTML, or XML).

When a user clicks one of the following radio buttons, GourmetCoffeeGUI handles the associated event as described below:

- *plainRadioButton*. Changes the format of the sales information to plain text.
- *HTMLRadioButton*. Changes the format of the sales information to HTML.
- *XMLRadioButton*. Changes the format of the sales information to XML.

When a user selects an element in the following list, GourmetCoffeeGUI handles the associated event as described below:

- *catalogList*. Displays the details of the selected product in the "Product Information" panel.

GourmetCoffeeGUI defines the following Swing components:

- *catalogList*. Displays the product code of every product in the product catalog.
- *orderList*. Displays the items in the current order.
- *quantityTextField*. Allows user to specify the quantity of the selected product.
- *totalTextField*. Displays the total cost of the current order.
- *statusTextArea*. Displays status messages and sales information.
- *fileChooser*. A JFileChooser object. Allows user to specify the name and location of the file in which the sales information will be saved. This dialog box appears when the “Save Sales” button is clicked.

GourmetCoffeeGUI defines the following instance variables:

- *catalog*. Contains the product catalog, a Catalog object.
- *currentOrder*. An Order object that contains the items to be purchased.
- *sales*. A Sales object that contains the orders that have been sold.
- *salesFormatter*. A SalesFormatter object that specifies the format to be used when the sales information is displayed or saved.
- *dollarFormatter*. A NumberFormat object used to find the dollar representation of a number. (The dollar representation of 1.0 is "$1.00".)

New methods have been added to classes Catalog and Order:

- The method getCodes has been added to the class Catalog. It returns an array of product codes (all the product codes in the product catalog) which is used by GourmetCoffeeGUI to populate the catalog JList.
- The method getItems has been added to the class Order. It returns an array of the OrderItem objects (all the items in the current order) which is used by GourmetCoffeeGUI to populate the order JList.

The button events are handled using named inner classes in class GourmetCoffeeGUI. The following inner classes are complete and should not be modified:

- *DisplayProductListener*. Listener for the catalog list
- *RegisterSaleListener*. Listener for the button "Register Sale of Current Order"
- *PlainListener*. Listener for the radio button "Plain"
- *HTMLListener*. Listener for the radio button "HTML"
- *XMLListener*. Listener for the radio button "XML"
- *DisplaySalesListener*. Listener for button "Display Sales"

In this assignment, you should complete the implementation of the following inner classes:

- *AddModifyListener*. Listener for button "Add|Modify Order Item"
- *RemoveListener*. Listener for button "Remove Order Item"
- *SaveSalesListener*. Listener for button "Save Sales"

Files

The following files are needed to complete this assignment:

- *exe-gourmet-coffee-gui.jar*. Download this file. It is the sample executable.
- *student-files.zip*. Download this file. This archive contains the following:
  - Class files
    - *Coffee.class*
    - *CoffeeBrewer.class*
    - *Product.class*
    - *Catalog.class*. A modified version of class Catalog
    - *OrderItem.class*
    - *Order.class*. A modified version of class Order
    - *Sales.class*
    - *CatalogLoader.class*
    - *FileCatalogLoader.class*
    - *DataFormatException.class*
    - *SalesFormatter.class*
    - *PlainTextSalesFormatter.class*
    - *HTMLSalesFormatter.class*
    - *XMLSalesFormatter.class*
    - *DataField.class*

- Documentation
  - *Coffee.html*
  - *CoffeeBrewer.html*
  - *Product.html*
  - *Catalog.html*. A modified version of class `Catalog`
  - *OrderItem.html*
  - *Order.html*. A modified version of class `Order`
  - *Sales.html*
  - *CatalogLoader.html*
  - *FileCatalogLoader.html*
  - *DataFormatException.html*
  - *SalesFormatter.html*
  - *PlainTextSalesFormatter.html*
  - *HTMLSalesFormatter.html*
  - *XMLSalesFormatter.html*
  - *DataField.html*
- Java files
  - *GourmetCoffeeGUI.java*. Use this template to complete your implementation.
- Data files for the test driver
  - *catalog.dat*. A file with product information for every product in the product catalog

Tasks

Complete the implementation of `AddModifyListener`, `RemoveListener`, and `SaveSalesListener`. The messages displayed in the status area by these inner classes should match the messages displayed by the sample executable.

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 by issuing the following command at the command prompt:

   ```
   C:\>unzip student-files.zip
   ```

2. **Run** the sample executable by issuing the following command at the command prompt:

   ```
   C:\>java -jar exe-gourmet-coffee-gui.jar
   ```

   Note: This application requires the file *catalog.dat* which you will find in `student-files.zip`.

3. **Then**, copy the code of method `getDataFieldsPanel` that you created in the previous exercise.

4. **Next**, implement method `actionPerformed` in the inner class `AddModifyListener`: If the current order does not contain an item with the selected product, then create a new order item and add it to the current order. Otherwise, locate the item in the order with the selected product and changes its quantity to the value specified by the user. Finally, update the current order list (use method `Order.getItems` to obtain an array containing all the items in this order), display a status message in the status area, and update the display of the order's total cost. Use the following code to display the order's total cost in dollars:

```
totalTextField.setText(dollarFormatter.format(currentOrder.getT
otalCost()));
```

This method should display an error message in the status area when it detects one of the following errors:

- o The quantity text field does not contain an integer
- o The quantity text field contains a negative integer or zero.
- o The user has not selected a product.

5. **Then**, implement method `actionPerformed` in the inner class `RemoveListener`: Remove the selected item from the current order, display a status message in the status area, update the current order list (use method `Order.getItems` to obtain an array containing all the items in this order), and update the display of the order's total cost. Use the following code to display the order's total cost in dollars:

```
totalTextField.setText(dollarFormatter.format(currentOrder.getT
otalCost()));
```

This method should display an error message in the status area when it detects one of the following errors:

- o The current order is empty.
- o The user has not selected an item in the order.

6. **Next**, implement method `actionPerformed` in the inner class `SaveSalesListener`: Save the sales information in a file. Begin by opening a file chooser so the user can specify the name and location of the file in which the sales information will be saved. Next, save the sales information in the specified file. The file should be saved in the format indicated by the radio button selection. Finally, display a status message in the status area.

This method should display an error message in the status area when it detects one of the following errors:

- o No orders have been sold.

- o The user closes the file chooser without selecting a file.
- o The file specified by the user cannot be created or opened.

7. **Finally**, compile and execute the class `GourmetCoffeeGUI`.

Submission

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

1. `GourmetCoffeeGUI.java`