Assignment 4

(total 191 points)

Exercise 1: Exception

(31 points)

Purpose: after finishing this exercise, you will know how to define a custom exception, how to throw an exception from a method, how to catch an exception and handle it, and the function of finally clause.

Tasks:

1. Please use the `extends` keyword to create a custom exception class `MyException`. **(5 points)**

   Please create a second exception class `MySpecialException` which is a subclass of `MyException`. **(5 points)**

2. Please create a class `ExceptionTest`.

   1. The class has two methods `testMException`() and `testMSException`(). The method `testMException` prints a message `"MyException is thrown"` and throws `MyException`. The method `testMSException` prints a message `"MySpecialException is thrown"` and throws `MySpecialException`. (5 points)
   2. In the main() method of the class, the two methods `testMException()` and `testMSException()` are invoked. `testMException()` and `testMSException()` are put into two different try-catch blocks. For each method, first `MySpecialException` is caught and then `MyException` is caught and then a **finally** clause is used at the end of all the exception handlers. In the catch clause to catch `MySpecialException` , a message `"MySpecialException is caught"` is printed. In the catch clause to catch `MyException` , a message `"MyException is caught"` is printed. In the finally clause, a message `"Here is finally clause"` is printed. **(10 points)**
   3. After finally clause, print a message "This is the end of main() method". **(1 points)**

3. Please run the program, observe the result and make clear why such a result is produced.

4. Please modify the program, move the two methods `testMException()` and `testMSException()` into one try-catch block and remove the redundant try-catch block. Please compare the result with the former result and understand why such a result is produced. **(5 points)**

Exercise 2: Collections

(80 points, each function 10 points)

Background

This assignment asks you to implement a set of methods to process Java collections.

Description

In this assignment, you will finish the implementation of `StudentArrayList`, a class with methods for processing an `ArrayList` of `Student` objects. A test driver and the class `Student` are provided.

Class `Student`

A complete implementation of `this` class is included in the student archive *student-files.zip*. Stop *now* and review its documentation:

- *Student.html*. Documentation of class `Student`

Class `StudentArrayList`

A partial implementation of this class is included in the student archive *student-files.zip*. You should complete the implementation of each method.

Class `TestStudentArrayList`

This class is a test driver for class `StudentArrayList`. It contains test cases for each method in the class. A complete implementation is included in the student archive *student-files.zip*. You should use this class to test your implementation of class `StudentArrayList`.

Files

The following files are needed to complete this assignment:

- *student-files.zip*. Download this file. This archive contains the following:
  - *Student.java*. A complete implementation
  - *StudentArrayList.java*. Use this template to complete your implementation.
  - *TestStudentArrayList.java*. A complete implementation

Tasks

Implement all methods in class `StudentArrayList`. 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. **Test** each method as soon as you finish writing it by issuing the following command at the command prompt:

   ```
   C:\>java TestStudentArrayList
   ```

3. **Implement** the method `makeArrayList`: **(10 points)**

- `public static ArrayList<Student> makeArrayList(Student first, Student second, Student third)`

  This method takes three `Student` objects and returns an array list with three elements. The first element of the array list contains a reference to the first argument; the second element contains a reference to the second argument; and the third element contains a reference to the third argument.

  For example, consider the following three `Student` objects:

  - `Student[328,Galileo Galilei,80]`
  - `Student[123,Albert Einstein,100]`
  - `Student[96,Isaac Newton,90]`

If these objects are passed to `makeArrayList` in the indicated order, `makeArrayList` will return an array list with three Student objects:

```
ArrayList{Student[328,Galileo Galilei,80],
   Student[123,Albert Einstein,100],
   Student[96,Isaac Newton,90]}
```

Note: `Student[ID,name,grade]` is a representation of a `Student` object.

4. **Implement** the method `makeArrayListFromArray`: **(10 points)**

- `public static ArrayList<Student> makeArrayListFromArray(Student[] array)`

  This method takes an array of `Student` objects and returns an array list with the same elements in the same order.

  For example, consider the following array with three Student objects:

```
{Student[328,Galileo Galilei,80],
Student[123,Albert Einstein,100],
Student[96,Isaac Newton,90]}
```

  If `makeArrayListFromArray` is passed this array, it will return an array list with three Student objects:

```
ArrayList{Student[328,Galileo Galilei,80],
   Student[123,Albert Einstein,100],
   Student[96,Isaac Newton,90]}
```

5. **Implement** the method `hasStudent`. Use a for-each loop to implement this method. **(10 points)**

- `public static boolean hasStudent(ArrayList<Student> arrayList, int id)`

  This method takes an array list of Student objects and a student ID, returns true if the specified array list contains a student whose ID matches the specified ID.

  For example, consider the following array list:

```
ArrayList{Student[328,Galileo Galilei,80],
    Student[123,Albert Einstein,100],
    Student[96,Isaac Newton,90]}
```

  If `hasStudent` is passed this array list and the id `123`, it will return `true`.

  If `hasStudent` is passed this array list and the id `321`, it will return `false`.

6. **Implement** the method `countGradeGreaterOrEqual`. Use a for-each loop to implement this method. **(10 points)**

- `public static int countGradeGreaterOrEqual(ArrayList<Student> arrayList, int grade)` This method takes an array list of `Student` objects and a grade, and returns the number of students in the specified array list whose grade is greater than or equal to the specified grade.

  For example, consider the following array list:

```
 ArrayList{Student[328,Galileo Galilei,80],
    Student[123,Albert Einstein,100],
    Student[96,Isaac Newton,90]}
```

If `countGradeGreaterOEqual` is passed this array list and the grade `100`, it will return `1`.

If `countGradeGreaterOEqual` is passed this array list and the grade `70`, it will return `3`.

7. **Implement** the method `getMinGrade`. Use an iterator to implement this method. **(10 points)**

- `public static int getMinGrade(ArrayList<Student> arrayList)`

This method takes an array list of `Student` objects and returns the smallest grade of the students in the specified array list. To simplify the code, you may assume that the array list is not empty.

For example, consider the following array list:

```
ArrayList{Student[328,Galileo Galilei,80],
    Student[123,Albert Einstein,100],
    Student[96,Isaac Newton,90]}
```

If `getMinGrade` is passed this array list, it will return `80`.

8. **Implement** the method `getGradeAverage`. Use a for-each loop to implement this method. **(10 points)**

- `public static double getGradeAverage(ArrayList<Student> arrayList)`

This method takes an array list of Student objects and returns the average grade of the students in the specified array list.

For example, consider the following array list:

```
ArrayList{Student[328,Galileo Galilei,80],
    Student[123,Albert Einstein,100],
    Student[96,Isaac Newton,90]}
```

If `getGradeAverage` is passed this array list, it will return `90.0`.

9. **Implement** the method `removeGradeLess`. Use an iterator to implement this method. **(10 points)**

- `public static void removeGradeLess(ArrayList<Student> arrayList, int grade)`

This method takes an array list of Student objects and a grade, and removes all students in the specified array list whose grade is less than the specified grade.

For example, consider the following array list:

```
ArrayList{Student[328,Galileo Galilei,80],
    Student[123,Albert Einstein,100],
    Student[96,Isaac Newton,90]}
```

If `removeGradeLess` is passed this array list and the grade 100, the array list will be modified as follows:

```
ArrayList{Student[123,Albert Einstein,100]}
```

10. **Implement** the method `displayAll`. Use an iterator to implement this method. **(10 points)**

- `public static String displayAll(ArrayList<Student> arrayList)`

  This method takes an array list of Student objects and returns a string representation of the specified array list. To simplify the code, you may assume that every element in the specified array list contains a valid reference to a Student object.

  Use the method `toString` in the class `Student` to obtain the string representation of each object in the array list. A new line character (\n) should separate the string representations of the objects.

  For example, consider the following array list:

  ```
  ArrayList{Student[328,Galileo Galilei,80],
      Student[123,Albert Einstein,100]}
  ```

  If this array list is passed to `displayAll`, the following string will be returned:

  ```
  "Student[328,Galileo Galilei,80]\nStudent[123,Albert Einstein,100]"
  ```

  Note that the string does *not* end with a new line character (\n).

Submission

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

1. *StudentArrayList.java*, *StudentArrayList.class*
2. a word file including the program running result.

Exercise 3: the Collections in the Gourmet Coffee System

(80 points, each class 20 points)

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:

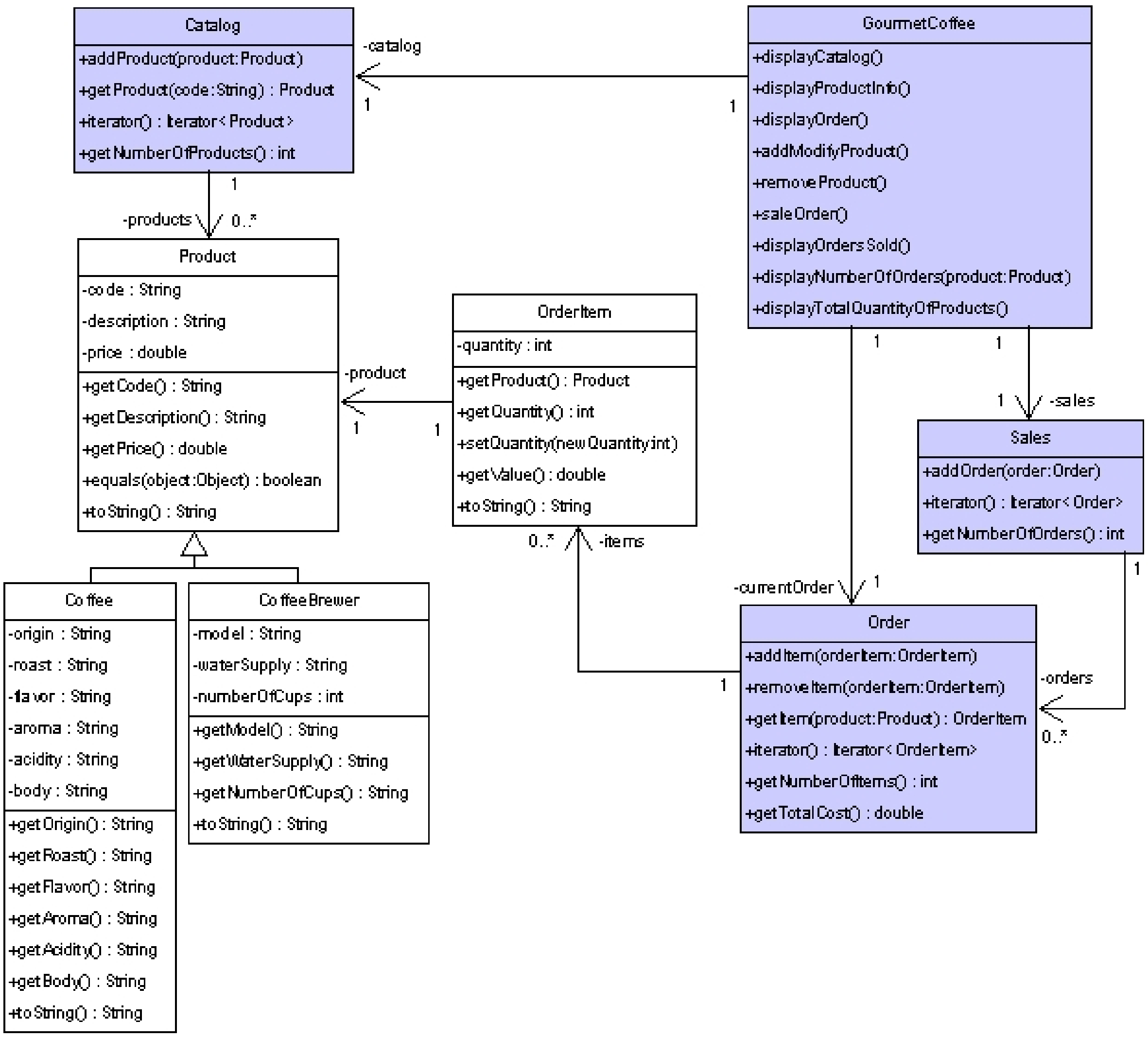


Figure 1 *Gourmet Coffee System class diagram*

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`.

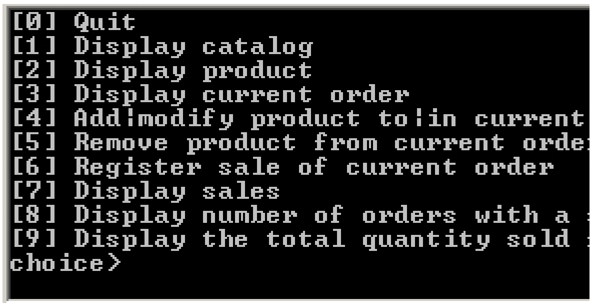
Constructor and public methods:

- `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
[5] Remove product from current orde
[6] Register sale of current order
[7] Display sales
[8] Display number of orders with a
[9] Display the total quantity sold
choice>
```

Figure 2 *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 from *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.
4. **Then** , implement class `Sales` from scratch. Use `TestSales` to test your implementation.
5. **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*, *Catalog.class*
2. *Order.java*, *Order.class*
3. *Sales.java*, *Sales.class*
4. *GourmetCoffee.java*, *GourmetCoffee.class*
5. a word file including the program running results