Assignment 2

Object Oriented Programming

University of Southern Denmark

Due date: 19th Dec. 2022, 23:59

This assignment considers a storage system that can store food and non-food items which might be added and removed.

Similar to assignment 1, as part of this assignment, you must document your code. This can either be done in a separate document or, alternatively, you can add the documentation directly as comments in the code. The documentation must describe:

- The overall logic of each class (a few lines)
- The overall logic and purpose of each constructor in a class (a few lines)
- The overall logic of each method in a class (a few lines)
- The property that each field variable represents (one line)

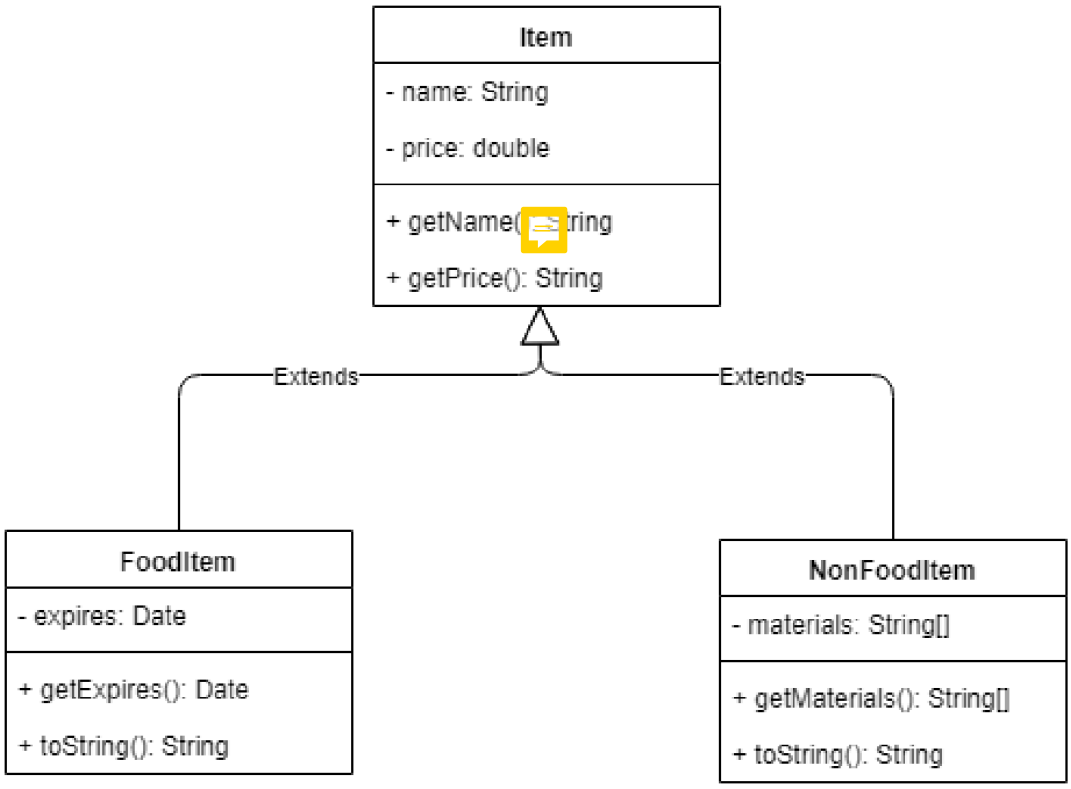
As far as possible, try to mention in the comments which step the different parts of the produced code belong to.

Furthermore, you must upload your project files after each part has been completed, essentially handing in three separate projects. This is due to the fact that some steps in one part require changes to solutions from a previous part, and it must be evident that you have conducted all the steps.

Part 1 - Inheritance

Step 1

The UML diagram below shows parts of a storage system where inheritance is included. You must implement the classes from the diagram as they are shown.



The `toString`-methods related to `FoodItem` and `NonFoodItem` must be annotated with `@Override` as they override the method from the `Object` class (not shown in the diagram).

The `toString` method in `FoodItem` must return the name, price and expiration date as a `String`.

The `toString` method in `NonFoodItem` must return the name, price, and list of materials as a `String`.

Step 2

In your `main` method, create an array that can contain 10 `FoodItem` objects. Fill each space in the array with a `FoodItem` object using a loop. In another loop, call `toString(…)` on each of the `FoodItem` objects in your array and print it using `System.out.println(String)`.

Note: Since you cannot set the name and price fields directly in the Item class from the two subclasses (you are not allowed to define Setter methods), you should instead set the value of the two fields using a constructor in the `Item` class that is called from a constructor in the two subclasses by using the `super` keyword (i.e., `super(inputName, inputPrice)`).

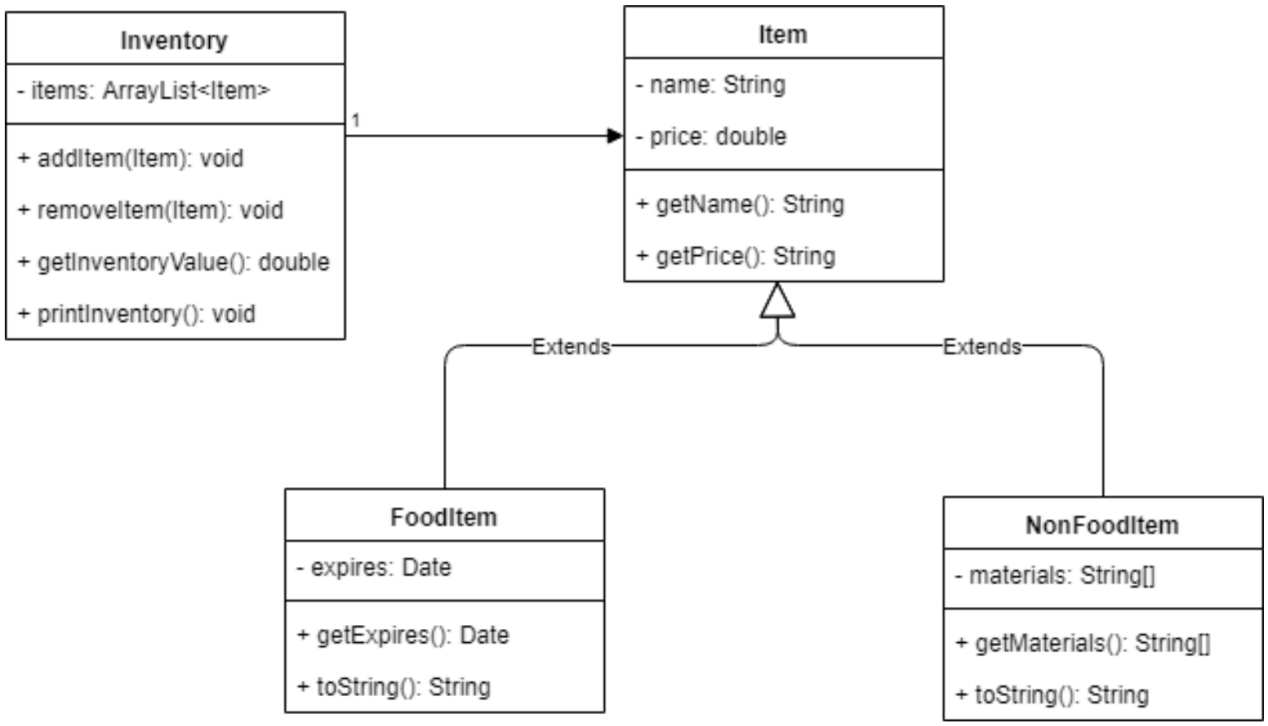
Step 3

Do the same as in Task 2, but this time for `NonFoodItem` objects (do not overwrite the code from Task 2 so that your program does it for both types of objects).

Part 2 – Polymorphism

Step 4

You should now expand your program to match the UML class diagram below:



The method `getInventoryValue()` must run through all `Item` objects in the list of items, call `getPrice()` on each of the objects and add the values for all the objects and return it.

`printInventory()` must print the text representation of all the objects in the list of items using `System.out.println(…)`. Again, a loop must be used here.

`addItem(Item)` and `removeItem(Item)` must add and remove objects to the items list, respectively.

Step 5

Update your `main` method so that you add some items of both `FoodItem` and `NonFoodItem` to an item list in an instance of `Inventory`.

Call `printInventory()` and `getInventoryValue()` and validate their functionality.

Step 6

The materials attribute in `NonFoodItem` uses an array. Replace the type here with an `ArrayList` and make the changes to your code that are needed for it to still work as intended.

Part 3 – Abstract Classes and Interfaces

Step 7

To ensure that products are always instantiated as either a `FoodItem` object or a `NonFoodItem` object, you must make the `Item` class abstract. What happens if you then try to create an object of type `Item` by calling the constructor on `Item` direct? Why is that the case?

Step 8

You must now create an interface called `Expireable`, with a method that has the following signature: `public boolean isExpired();`

Step 9

You must then implement the interface in the `Item` class. Implementing an interface means that you must use the implements keyword in `Item`, and that you must override the `isExpired()` method from the interface. The body of `isExpired()` in `Item` should contain only one line of code: `throw new UnsupportedOperationException("Item does not support this operation.");`

Step 10

Since `FoodItem` and `NonFoodItem` inherit from Item, they also inherit the `isExpired()` method. You must now override this method in the `FoodItem` class so that it uses the `expireDate` attribute to determine if the product is too old. Hint: The `Date` type in Java can be used to represent dates.

Step 11

In the `Inventory` class, you must implement a method, `public void removeExpiredFoods()`, that iterates through the list of `Item` objects and calls `isExpired()` on each of the objects. If `isExpired` returns `true`, that `Item` object must be removed from the list of `Item` objects. Since `Item` itself does not implement `isExpired()` but throws an `Exception`, you must make sure to use `try`/`catch` around the call to `isExpired()` to ensure that the program does not crash when dealing with `NonFoodItem` objects.