Programming the .Net Framework using C#

# Drinks Vending Machine

Object Oriented programming in C#

## Instructions

Write a simple (Graphical) application that simulates a hot-drinks vending machine.

The machine should offer different types of hot drinks, (e.g. tea, coffee, chocolate drink etc.), each with its own name, price and preparation method.

Clients choose their preferred drink, pay, and the machine should prepare it.

New drinks are added to the machine from time to time. Build a general mechanism to support these adaptations.

Use a Graphical User Interface to your system.

Think of resources relevant for all types of drinks (cups, sugar, etc.). Who’s responsible for managing those?

What happens in extreme circumstances (e.g. no cups or sugar left? Drink selected not available?)

Create a UML diagram for your application, showing the different classes (methods and data members) and relationships between them.

Implement the project based on your design.

## Detailed Instructions

The following is an **overview** of the problem. You might need **more** classes, and / or add methods and fields to the classes below. View the following is a **starting point** to your design**. The rest is up to you!**

**The Beverage class:** (One base class for all Beverages, specific derived class for each Specific type of drink: Coffee, tea etc.).

* Each Beverage has a Name
* Each Beverage has a Price
* Possibly: a Picture/Icon for display
* All Beverages have the method: "prepare". Preparation steps are always the same: (we do not want any subclass to change the order of the steps taken!):
  + adding Ingredients,
  + adding hot water,
  + and stirring.

Yet , for each **type** of beverage (coffee, tea – **derived classes**) , the implementation of these steps (add ingredient, stir, heat water) might be different (for example: the ingredients list may be different, water temperature needed might be different).

For simplicity, the preparation method above might be implemented as creating a string of steps to do (“adding coffee beans”, “adding tea leaves” etc.). The method should return the string. The string should be later displayed on the User Interface by the caller of this method (Not by the Beverage itself).

* Each Beverage requires a list of ingredients – the resources from which a drink is made of: coffee beans, hot water, sugar, tea leaves, etc. You may decide, for each type of drink you implement, what are the necessary ingredients (for example: define for coffee that it requires coffee beans and milk).
* Override Object's ToString and Equals methods for Beverages.
* You may add more properties and methods as you feel needed.

**The vending machine class:**

* Should manage Beverages of different kinds. Make it as general as possible. The machine class itself should not be familiar with specific beverages such as tea or coffee, so that you might add more types of beverages to it without changing this class.
* The Vending machine should have a private collection (array, for example) of Beverages, with indication how many Beverages are already in the collection (for example: keep an index)
* Have a method for selecting a drink and preparing it, using the beverage "prepare" method.
* Should have a method to add a beverage, a method to remove a beverage (so that we can add beverages to the class in the future, without changing its code).
* Make sure you handle situations when a beverage is required from the machine and it is not available. Use Exceptions if needed.
* It might be helpful to implement an indexer to retrieve the beverages stored in the vending machine.
* You may add more properties and methods as you feel needed. For example: who is responsible for cups?
* **Optional: (not required, possible enhancement if time permits):** Add ability to manage the resources stock: the materials from which beverages are made from (tea leaves, coffee beans, sugar etc). Try to define this as general as possible, so that new stock items (resources) may be added.

**A Manager class (Suggested)**

It might be helpful to add a “Manager” class that manages the Vending machine. The Manager will be responsible to create the Vending Machine, add drinks to the machine, add resources etc.

While we do not want the vending machine code to change each time a new type of drink is added, the manager may change. (The Manager is the place where you create the specific drinks, and call the Vending machine methods in order to add them, operate the Vending Machine options, etc).

You may think of the Manager as the "Program" class in some applications.

**General Considerations:**

Exceptions

* Describe and handle Exceptional cases using Exceptions

Testing your program:

* (In the Manager Class code?) Define at least two different types of Beverages. Add them to the vending machine. Test you system to see if a user can select a beverage, pay for it, and have the system prepare it.
* The GUI should show the string returned by the Beverage when asked to prepare it (For example: when you chose coffee, show the string created by the prepare method, such as “Adding coffee beans, boiling water to 100 degrees, stirring 3 time…” etc)
* The GUI should show the types of drinks available.
* Show what happens if a drink is selected, but cannot be made (no cups, no more sugar, drink not available, etc).

## Features in C# that may be used in this project:

* Classes, Object Based programming, encapsulation
* Object Oriented programming, inheritance and polymorphism (**very important in this Exercise)**
* Arrays and Strings
* Indexers (optional)
* Enums (optional)
* Exception Handling - required

## Important Notes

* Build your classes based on the principles we learned in class.
* Define private and public as needed.
* Use virtual methods if needed, override, etc.
* Define your inheritance tree carefully.
* Define the UML diagram before you implement your code. Make sure your design is feasible before you code
* Design the Vending Machine in general terms, using Beverages Base class only – no knowledge of beverage sub classes should be included in the vending machine class.
* Test your design: think if adding a new type of beverage requires any change **to the vending machine** **class**. The answer in a good design should be, of course, no!
* As always, make sure your code is readable, using meaningful names, indentation and documentation.