# Problem Overview

Create a simulator of an automatic coffee dispensing machine, called the "*Barista-matic"*.

The machine maintains an inventory of drink ingredients, and is able to dispense a fixed set of possible drinks by combining these ingredients in different amounts.

# Problem Statement in Points

1. Upon startup, the Barista-matic should display a list of its current inventory, followed by a menu to allow the user to select a drink.
2. As drinks are dispensed, the inventory should be updated.
3. Only drinks for which there is sufficient inventory can be dispensed.
4. Initially the Barista-matic should contain 10 units of all ingredients.
5. Restocking the machine should restore each ingredient to a maximum of 10 units.
6. If the user enters an invalid command, then the program should display a single-line message with the following format:

Invalid selection: <characters that were entered>

1. If the user selects a valid drink number, and the machine has the required ingredients to make the drink, then the program should display a single-line message with the following format:

Dispensing: <drink name>

1. if the drink order cannot be completed, then the program should display a single line message with the following format:

Out of stock: <drink name>

# Restrictions for Input and Output Formats

1. The solution should be a command-line program written in Java.
2. The specified input and output formats for the Barista-matic must be followed exactly.
3. All output should be written to the standard output stream.
4. The output generated by program should not have any whitespace at the beginning of a line.
5. The inventory list should be displayed in alphabetic order by ingredient name.
6. The drink menu should be displayed in alphabetic order by drink name.
7. Drinks should be numbered sequentially, starting at 1, in the order they are displayed in the menu.
8. The in-stock indicator should be either "true" or "false".
9. Should read from the standard input stream, one command per line.
10. No prompts or other extraneous user messages should be displayed.
11. Blank input lines should be ignored.
12. The system should re-display the inventory and menu immediately after any applicable message.

# Technical restrictions

* It is not required that the initial machine configuration (inventory counts, available drinks and prices, etc.) be dynamic. In particular, it is acceptable to perform this initialization in code, rather than reading the configuration from an external file or database.
* Your program should be flexible enough to allow new drinks to be added to the menu without requiring extensive code changes.
* Make sure your program works correctly for all combinations of inputs. You may include automated tests
* Extensive inline or method-level comments are not required, unless you want to include them to highlight particular aspects of your design or implementation.

# Main interfaces, classes and Exceptions

* BaristaMachine - an interface which defines public API of Barista-Matic Machine.
* BaristaMachineImpl - a general purpose implementation of BaristaMachine interface.
* Drink - An abstract class which defines public API of dispensed drink.
* DispensedDrink - A concrete class representing dispensed drink and extending Drink class.
* DrinkData - A helper package scope class to provide all data that used to make drink.
* Ingredient - A base decorator class that extends the base drink class for representing any ingredient and can act as drink at any time.
* DrinkRefName – A type-safe Enum to represent reference names of available drinks that are supported by machine.
* IngredientData - A helper package scope class to provide all data that used to build ingredients.
* IngredientRefName – A type-safe Enum to represent reference names of available ingredients that are used for preparing drinks supported by machine.
* IngredientInventory - A singleton class that hides all the complicated operations that need to be done on the inventory and to keep tracking the ingredients amounts, deduction entries, and the availability of ingredients in the inventory.
* Inventory - A type-safe inventory for holding objects, which is an ADAPTER or WRAPPER over java.util.Map.
* InvalidDrinkNumberException - thrown when user enters an invalid command.
* OutOfStockException - thrown when machine doesn't have enough ingredients for the selected drink.

# Data structures used

* EnumMap data structure is used to implement
  + Ingredient, unit price pair.
* [Map](http://java67.blogspot.com/2015/01/how-to-convert-map-to-list-in-java.html) data structure is used to implement
  + Ingredient, units pair for drink.
  + Ingredient, unit price pair.
* List is used to representing list of available drinks.

# Design patterns used and design decisions

* Decorator Design pattern by using the Decorator pattern, the application can extend the functionality of Drink class at runtime, based on predefined ingredients units, beside that this pattern makes the design very flexible in the future in case of any updates could take place on the machine to give the customer an option to choose the ingredients’ units for drink.
* Singleton design pattern is used to be sure that only one instance of IngredientInventory will be globally available at any time of execution since this class gives the chance for the machine to delegate all complicated inventory operations to it such as:

1. Applying the ingredients unit deduction for drink from inventory with possibility of throwing OutOfStockException.
2. To share the deduction entries with the machine.
3. To make it easy to restore the ingredients units in inventory at startup of the machine or upon user request.

* [Factory design pattern](http://javarevisited.blogspot.com/2011/12/factory-design-pattern-java-example.html) is used to encapsulate creation logic of Drink.
* Adapter pattern is used to create Inventory by wrapping java.util.Map
* “java.lang.Enum” is used to represent as reference name for Ingredient, because of following benefits:  
  1- Compile time safety against entering an invalid ingredient name.  
  2- No need to write code for checking if selected ingredient is valid.  
  3- Reusable and well encapsulated.
* [BigDecimal](http://javarevisited.blogspot.com/2012/02/java-mistake-1-using-float-and-double.html) to represent money instead of double primitive to follow best practices suggested in many resources.

# Assumptions

* The coffee dispensing machine is able to dispense a fixed set of possible drinks by combining these ingredients in different amounts but ingredients are not customizable i.e. in case of the user asking for “Coffee” drink, he cannot change the amounts of any of the ingredients such as adding one more unit of sugar or removing the sugar at all.

# Reference Documentation

For further reference, please consider the documents folder inside the project folder:

* UML class diagram
* Generated JavaDoc