Homework 4 – Inventory System

**Due Sunday, October 27th by 11:59PM**

You’re going to create a program that simulates an inventory system in a game, which can hold different types of items. While you’ll be using inheritance to provide a common base class for all items, you’ll also use an enum to classify items in a different way. Your inventory system should throw exceptions if it cannot properly perform a particular task.

# Task Overview

This is a brief overview of the tasks you must complete for this assignment. Specifics are given in the corresponding sections later in this document. You should ***read the entire document*** before starting.

* Create an **enum** which represents different categories of items
* Create an **abstract parent class** for all items
* Create **three specific item classes** which appropriately inherit from the base class
* Create an **Inventory class**, which takes care of storing and retrieving items
* Appropriately **throw and catch exceptions** where necessary

# Enum

Create an enum called ItemCategory. Its exact values are up to you, but they’ll need to make sense with the items you’ll be creating later. Be sure to define it inside the namespace but outside of any classes, as it’ll be used in multiple classes. You could create a new file for it (and remove the class inside that file), or simply put it in the base Item class file which you’ll make in the next step.

For instance, your enum might have values such as Food, Weapon and Potion.

# Item Class

The base Item class should be abstract. It will need fields that can describe any item in your “game”. Come up with at least two fields (name, weight, etc.) in addition to an ItemCategory field. Create “get” properties for them. Your Item class will also need a parameterized constructor to set up all the fields.

All items will need a Use() method. Since each child class of item will be different, you can’t write a useful Use() method here. Instead, create an abstract Use() method in the Item class so that each sub class must override it. This method can be void with no parameters.

Override ToString as well, returning a string with the name of the item and some information about its category. For example, a Sword’s ToString might read (although yours should be more detailed):

The Sword of Bored Hordes, a Weapon-type item

# Item Sub Classes

Create at least 3 subclasses of Item, each with an extra field relating to what kind of item it is. For instance, if you have a Sword class it might need a durability field, whereas a Sandwich class might have a “number of bites left” field. Each sub class should correspond to an item category in your custom enum from above.

Create properties as necessary, as well as a parameterized constructor that sends any required info to the base class’s constructor and saves the rest.

Override the Item class’s Use() method, and have it both update one of the item’s fields and print out information about the item (using ToString). For example, using a Sword might reduce the durability and print out a message like:

You have swung The Sword of Bored Hordes, a Weapon-type item. Only 5 more swings before it’ll need to be repaired.

If the Sword’s durability is already zero, you’d print out a different message:

You attempt to swing The Sword of Bored Hordes, a Weapon-type item, but it feels so fragile from battle that you decide against it.

# Inventory Class

The Inventory class will hold any amount of items, and should contains methods to retrieve items randomly. You’ll need two fields: a List<Item>, and a Random object. The constructor should initialize both fields.

## Count Property

This property of the Inventory class should have only a “get”, which returns the current count of items in the inventory’s list.

## void Add( Item item )

Accepts an Item object and adds it to the inventory’s List, as long as the item being added is not null. If the item parameter is null, create and throw an ArgumentNullException. Be sure to use the constructor which takes *two strings*: the name of the parameter which was null, and an appropriate error message.

## Item RemoveRandomItem( )

This method should pick a random item in the inventory, remove that item from the list and then return it. If there are no items in the list, throw an InvalidOperationException with an appropriate message.

The exception’s message should mention that the inventory is empty.

## Item RemoveRandomItem( ItemCategory category )

This method takes an ItemCategory as a parameter, and should return a random item from that category. You’ll need to go through the list and find all items of the specified category before choosing one at random. If no items of that category are found, throw an InvalidOperationException with an appropriate message.

The exception’s message should mention that there are no more items of the specified type left in the inventory.

# Main Method

## Item Creation

Your main method should create an Inventory object, instantiate 10 different objects from the various item sub-classes you’ve created and add those items to the Inventory. Each item should be slightly different, and you should make at least two objects of each sub-class. Feel free to hard-code the creation of these objects.

## User Input

Loop and prompt the user to enter a command to determine which type of Item to randomly use from their inventory. The user should be able to type “random”, or a string that corresponds to any value in the ItemCategory enum you created. Be sure to accept any capitalization of the words.

**Shortcut:** You can use Enum.GetNames(typeof(ItemCategory)) to get an array of strings that correspond to the names of the enum values. This could be useful to print a small menu so the user knows what they can type.

## Checking User Input

If the user typed “random”, get a random item from their inventory.

If the user typed something other than random, you’ll need to determine if it corresponds to a valid value in the ItemCategory enum, and if so get a random item from that category. There are two ways of going about this:

* You could set up a series of if statements (or a switch) to check for valid strings, which works but it means you need to hardcode all of the possible enum values.
* Another way is to attempt to parse the string into an enum value directly. The Enum class has two methods, .Parse() and .TryParse<T>(), which may be of use to you. They work a little differently than int.Parse(), so check the documentation if you go this route. Also remember that .Parse() could throw an exception, be sure to try/catch appropriately.

## Acting on User Input

Regardless of which kind of item the user is getting, an exception could be thrown if the inventory is empty. Use try/catch blocks to determine if the operation completed successfully. If no exception is thrown, inform the user they were successful and then Use( ) the item. If there was an exception, print the exception’s error message. See the example output below.

After using the item, ask the user if they would like to put the item back in their inventory (and call Add if necessary). This should only happen if the user actually retrieved a valid item.

The overall loop should continue until the user is out of items in their inventory, or they type “done”.

# Example Output

Your inventory currently has 10 items.   
What kind of item would you like to use: Random, Weapon, Food, Potion? **Random**

You have taken a bite of **The Baconator, a Food-type item**. It has 92 bites left.

Would you like to put **The Baconator, a Food-type item** back? **Yes**

You put **The Baconator, a Food-type item** back in your inventory.

Your inventory currently has 10 items.   
What kind of item would you like to use: Random, Weapon, Food, Potion? **Potion**

You have drunk **The Potion of Weird Smells, a Potion-type item**, and you now smell like moldy cheese. The next time you drink it, you’ll smell like something different.

Would you like to put **The Potion of Weird Smells, a Potion-type item** back? **Yes**

You put **The Potion of Weird Smells, a Potion-type item** back in your inventory.

Your inventory currently has 10 items.   
What kind of item would you like to use: Random, Weapon, Food, Potion? **Potion**

You have drunk **The Potion of Weird Smells, a Potion-type item**, and you now smell like an old sock. The next time you drink it, you’ll smell like something different.

Would you like to put **The Potion of Weird Smells, a Potion-type item** back? **No**

You have discarded **The Potion of Weird Smells, a Potion-type item**.

Your inventory currently has 9 items.   
What kind of item would you like to use: Random, Weapon, Food, Potion? **Potion**

You are out of potions.

Your inventory currently has 9 items. What kind of item would you like to use:

Random, Weapon, Food, Potion? **done**

Thanks for playing!