5. LINQ and Collections

Exercise 5.1 – FlavorOps Flavor Operations

1. Create a new class FlavorOps that will contain operations on the type Flavor. Use the following code to flesh out the new operations.
2. The class is static because it does not have any instance data and only operates on the type Flavor, not values of type Flavor. Compare this relationship to the .NET Framework class Math (i.e., int is to Math as Flavor is to FlavorOps).
3. Because the class FlavorOps is so tightly connected to the type Flavor, you can place it in the same source code file.

```
public static class FlavorOps
 {
     private static List<Flavor> _allFlavors = new List<Flavor>();

     static FlavorOps()

     // method to convert a string value into an enumeral
     public static Flavor ToFlavor(string FlavorName)

     // property to return a List<Flavor> of all of the Varieties
     public static List<Flavor> AllFlavors

 }
```

4. Since this new class is adding additional functionality (not changing existing functionality), the rest of your vending machine code should behave exactly as before. In the next exercise, we will use these operations in CanRack, with one last modification to CanRack.

Exercise 05.2 – Change CanRack Implementation to Dictionary

1. Change the implementation of CanRack (I know, but this is the last change to CanRack, I promise!) from array of ints to the generic collection class Dictionary using the following declaration:

```
private Dictionary<Flavor,int> rack = null;
```

2. Change the code in CanRack to reflect this change in implementation. The changes occur in only a few places, and you'll be deleting as many lines as you are modifying.
3. Using the FlavorOps code, the foreach loops will now be much more readable. They will look like this:

```
foreach (Flavor aFlavor in FlavorOps.AllFlavors)
```

4. This change to the use of Dictionary will allow you to treat rack as if it were an array indexed by Flavor values rather than Flavor values typecast as ints.

```
rack[Flavor.ORANGE]
```

5. Also, this change allows us to avoid the need for several utility methods (e.g,, there is no longer a need to convert Flavor values into ints). There should no longer be any calls to the Enum class inside of CanRack.
6. The purpose of the CanRack implementation exercises (from int fields, to int arrays, to Dictionary collection) is twofold:
   1) To get an understanding of what collections are, how they are constructed, and how powerful they can be, and
   2) to demonstrate the power of hiding implementation details (note how little the surrounding code needs to change despite our modifications of CanRack).

Exercise 05.3 – Create a CoinBox

1. Add a class CoinBox to hold coins inserted into the vending machine. This class will use a generic List collection to hold the coins. Use LINQ queries in Withdraw(), and each of the properties that returns the number of coins of a particular denomination.

```
class CoinBox
{
      private List<Coin> box;

      // constructor to create an empty coin box
      public CoinBox()

      // constructor to create a coin box with some coins in it
      public CoinBox(List<Coin> SeedMoney)

      // put a coin in the coin box
      public void Deposit(Coin ACoin)

      // take a coin of the specified denomination out of the box
      public Boolean Withdraw(Coin.Denomination ACoinDenomination)

      // number of half dollars in the coin box
      public int HalfDollarCount

      // number of quarters in the coin box
      public int QuarterCount

      // number of dimes in the coin box
      public int DimeCount


      // number of nickels in the coin box
      public int NickelCount


      // number of worthless coins in the coin box
      public int SlugCount

      // total amount of money in the coin box
      public decimal ValueOf
}
```

2. Modify your Main() method to drop coins into the CoinBox as they are inserted by the user. At the end of Main(), report the number of coins of each denomination and the total value of change in the coin box.

OPTIONAL ----------

3. If you coded each property that returns the number of coins of that denomination separately, there is a lot of repetition code in those properties. Refactor this code into a base method that can be called by each property.

Exercise 05.4 Exceptions

- User input is notorious for throwing exceptions. While we could add code to filter the input that the user types in to assure that it is acceptable (most notably using TryParse() methods), for the purposes of this exercise, use Exception handling try blocks inside of loops to allow the user to retype any bad input.
- This will allow the program to recover from the mistyping of the name of a coin or of a flavor.

OPTIONAL ----------

1. Define a custom exception VENDBADFLAVORException.
2. Throw this exception in any method that takes a flavor value when that flavor value is not in the enumerated type. For example, if someone called

    CanRack.AddACanOfFlavor(“Oragne”)

3. Catch this exception in the appropriate methods.