1: Vending Machines (with Tutor)

In this exercise, we will implement a class that simulates a *Vending Machine*.

The exercise covers the topics of *Getter* and *Setter methods*, *methods overloading,* and the usage of the *this* keyword. Its second part provides you with the opportunity to practice with *exceptions*, as we will revisit how to *catch* Exceptions (seen during your independent study in Week 6) and learn how to *throw* them.

Requirements

A vending machine holds *cans* of soda. The customer must insert a token into a machine to buy a can. When the token is inserted, a *can* drops into the product delivery slot from the can reservoir. A vending machine can be *filled* with more *can*s. The goal is to determine how many *can*s and *token*s are in a *Vending Machine* at any given time and identify if a machine needs to be refilled with more cans.

Consider what **attributes** are needed to keep the machine's state and what **methods** should be defined to simulate the above-described operations. The source of a testing *Program* class is provided to help identify the required methods, as well as their parameters and return values.

When implementing the *VendingMachine* class please note the following:

- It should have two *overloaded constructors*. The first one has one parameter to set the initial number of soda cans in the machine. The second one assumes that the can's reservoir will be empty. Use *this* to chain the two constructors.
- It should have *Getter* and *Setter* methods associated with the attributes.
- It should have a method *AddCans(int c)* to refill the can's reservoir with *c* cans.
- It should have a method *InsertToken(int t)* to simulate the insertion of *t* tokens to buy *t* soda cans. The method returns a boolean value indicating the result of the operation. There is an overloaded version of this method for buying a single soda can.
- It should have a method *ToString* that returns a string representation of the status of the vending machine. When defining the method, you must add a keyword *override*—ignore why this keyword is required for now.

Error Handling

Whilst the first *VendingMachine* object is created with an empty can's reservoir, the second one has an initial number of soda cans received as user input. Modify the program to deal with potential error conditions related to the input format (i.e., it must be a positive integer).

Handling Exceptions

Use the exception-handling approach learned in Week 6 to deal with this unexpected program condition and ensure the user is prompted to type in a new value until the provided input is valid.

Throwing Exceptions

To deal with a negative number of soda cans, modify the constructor(s) of the *VendingMachine* class so that it *throws* an exception when the provided argument is less than zero. The instruction required for throwing a new exception is as follows:

```
throw new ArgumentException("The number of cans must be a positive integer");
```

At home: think about how the program should be modified so that the method *InsertToken* generates an *InvalidOperationException* instead of using error codes to indicate an error with a can purchase.

Program.cs

```
using System;

namespace VendingMachines
{
   class Program
   {
      public static void Main(string[] args)
      {
         Random r = new Random();

         /* creating first vending machine */
         VendingMachine m1 = new VendingMachine();
         m1.SetName("Machine 1");
         m1.SetLocation("4th Floor");
         // Randomly adding cans to m1 (1 to 10)
         m1.AddCans(r.Next(1, 11));

         /* creating second vending machine */
         // reading initial number of cans as input
         Console.Write("How many cans in Machine 2? ");
         int cans = Convert.ToInt32(Console.ReadLine());
         VendingMachine m2 = new VendingMachine(cans);
         m2.SetName("Machine 2");
         m2.SetLocation("Ground Floor");

         // Printing a summary of the status of the vending machines
         Console.WriteLine(m1.ToString());
         Console.WriteLine(m2.ToString());
         Console.WriteLine();

         // now buying some cans from the two machines
         int tokensToInsert = r.Next(1, 10);
         Console.WriteLine($"Attempting to insert {tokensToInsert} tokens into {m1.GetName()}");
         if (m1.InsertToken(tokensToInsert))
         {
            Console.WriteLine($"Successfully bought {tokensToInsert} cans from {m1.GetName()}");
         }
         else
         {
            Console.WriteLine("Error: Not enough cans in " + m1.GetName());
         }

         Console.WriteLine();

         Console.WriteLine($"Attempting to insert 1 token into {m2.GetName()}");
         if (m2.InsertToken())
         {
            Console.WriteLine("Successfully bought one can from " + m2.GetName());
         }
         else
         {
            Console.WriteLine("Error: Not enough cans in " + m2.GetName());
         }

         Console.WriteLine();

         // checking what machines need refelling
         if (m1.GetCans() <= 3)
         {
            Console.WriteLine($"Refill {m1.ToString()}");
         }

         if (m2.GetCans() <= 3)
         {
            Console.WriteLine($"Refill {m2.ToString()}");
         }
      }
   }
}
```

2: Primitive and Reference types (independent work)

Starting from the *BankAccount* class implemented during the Week 5 tutorial, add two new methods:

```
public void MoveAccount(SavingAccount otherAccount)
```

```
public bool IsOpen()
```

Refer to this week's (Week 7) lecture and the code of the *Program* class provided below to identify the content of the methods. Please note that the code of the *SavingAccount* class has been given to you (see below).

After you write the required code in the *BankAccount* class, **draw a diagram** that shows what the allocated variables are when the *Main* method is executed. Consider value and reference types, their allocation on the stack or heap memory, and how they are passed as parameters between various method invocations. Specifically, focus on the execution of the following instructions:

```
BankAccount account1 = new BankAccount("A0123", 1000.50);
SavingAccount account2 = new SavingAccount("A0456", 490.10, 3.8);

…

if ( … )
{
   account1.Withdraw(50.0);
   account1.MoveAccount(account2);
}
```

Without representing the memory status during the invocation of the methods *GetBalance* and *IsOpen*.

Hint

*While solving this problem, please remember the following. When a method is invoked, the default parameter passing mechanism used in C# is **pass by value**. More specifically, the value of each argument is copied to a **new variable** allocated on the local variable frame of that method. For **value types**, the content of the variable would be a copy of the argument value. Similarly, **reference types** are also copied, but the reference to an object (not the actual object) is copied instead.*

*Consider the difference between copying a value type and a reference type and why that reference can be used **inside** a method to **change** the state of the referred object even after that method terminates.*

Program.cs

```
namespace Bank;
class Program
{
   static void Main(string[] args)
   {
      BankAccount account1 = new BankAccount("A0123", 1000.50);
      SavingAccount account2 = new SavingAccount("A0456", 490.10, 3.8);

      Console.WriteLine("Account " + account1.GetNumber() + " open? " + account1.IsOpen());
      Console.WriteLine("Account " + account1.GetNumber() + " balance: " + account1.GetBalance());

      Console.WriteLine("Account " + account2.GetNumber() + " open? " + account2.IsOpen());
      Console.WriteLine("Account " + account2.GetNumber() + " savings: " + account2.GetBalance());


      Console.WriteLine();

      if (account1.IsOpen() && account2.IsOpen())
      {
         account1.Withdraw(50.0);
         account1.MoveAccount(account2);
      }

      Console.WriteLine("Account " + account1.GetNumber() + " open? " + account1.IsOpen());
      Console.WriteLine("Account " + account1.GetNumber() + " balance: " + account1.GetBalance());

      Console.WriteLine("Account " + account2.GetNumber() + " open? " + account2.IsOpen());
      Console.WriteLine("Account " + account2.GetNumber() + " savings: " + account2.GetBalance());

   }
}
```

SavingAccount.cs

```
namespace Bank
{
    class SavingAccount
    {
        private string number;
        private double balance;
        private double interest;
        private bool open;

        public SavingAccount(string num, double bal, double i)
        {
            number = num;
            balance = bal;
            interest = i;
            open = true;
        }

        public void Save(double amount)
        {
            // apply the interest rate
            amount += (amount * interest) / 100;
            // add new amount with interest to the balance
            balance += amount;
        }

        public double GetBalance()
        {
            return balance;
        }

        public string GetNumber()
        {
```

```
            return number;
        }

        public bool IsOpen()
        {
            return open;
        }
    }
}
```

3: Passing Value and Reference types to methods (at-home independent work)

Given the definitions of the *Program*, *Mutator* and *Cat* classes reported below, try to guess what the program's output will be when the *Main* of the *Program* class is executed. After you write it down, run the program and verify your guess. Be sure you fully understand the output and any differences you notice with your thoughts.

Program.cs

```
using System;
namespace ReferenceTest
{
   class Program
   {
      static void Main(string[] args)
      {
         Mutator mutator = new Mutator();

         int n = 100;
         Console.WriteLine("n before increment: " + n);
         mutator.Increment(n);
         Console.WriteLine("n after increment: " + n);
         Console.WriteLine();

         string river = "mississippi";
         Console.WriteLine("river before uppercase: " + river);
         mutator.UpperCase(river);
         Console.WriteLine("river after  uppercase: " + river);
         Console.WriteLine();

         Cat foo = new Cat("pink");
         Console.WriteLine("cat colour before mutation: " + foo.GetColour());
         mutator.ChangeColour(foo);
         Console.WriteLine("cat colour after mutation: " + foo.GetColour());
         Console.WriteLine();

         int[] array = new int[3];
         array[0] = 100;
         array[1] = 200;
         array[2] = 300;

         Console.WriteLine("Int Array:");
         foreach (int e in array)
            Console.Write(e + " ");

         mutator.ChangeArrayElements(array);

         Console.WriteLine();
         Console.WriteLine("Int Array after changeArrayElements:");
         foreach (int e in array)
            Console.Write(e + " ");
         Console.WriteLine();

         Cat[] cats = new Cat[3];
         cats[0] = new Cat("brown");
         cats[1] = new Cat("red");
         cats[2] = new Cat("white");

         Console.WriteLine();
         Console.WriteLine("Cats Array:");
         foreach (Cat c in cats)
            Console.Write(c.GetColour() + " ");

         mutator.ChangeArrayElements(cats);
```

```
            Console.WriteLine();
            Console.WriteLine("Cats Array after changeArrayElements:");
            foreach (Cat c in cats)
                Console.Write(c.GetColour() + " ");
        }
    }
}
```

Mutator.cs

```
using System;
namespace ReferenceTest
{
    public class Mutator
    {
        public void Increment(int n)
        {
            n++;
        }

        public void UpperCase(string s)
        {
            s.ToUpper();
        }

        public void ChangeColour(Cat c)
        {
            c.SetColour("black");
        }

        public void ChangeArrayElements(int[] a)
        {
            for (int i=0; i < a.Length; i++)
                a[i] = -1;
        }

        public void ChangeArrayElements(Cat[] cs)
        {
            foreach (Cat cat in cs)
                cat.SetColour("black");
        }
    }
}
```

Cat.cs

```
using System;
namespace ReferenceTest
{
    public class Cat
    {
        string colour;

        public Cat(string c)
        {
            colour = c;
        }

        public void SetColour(string c)
        {
            colour = c;
        }

        public string GetColour()
        {
            return colour;
```

```
        }
    }
}
```