# 1: Information Hiding (with Tutor)

In this exercise, we will develop a program that uses classes defined in different *namespaces* and different Visual Studio Projects (*assemblies*). We will then demonstrate the concept of *information and implementation hiding*. Specifically, we will show that changing the internal implementation of a class does not affect other classes that use it if the interface exposed by that class remains unchanged.

a) We start by creating a new project *Person* that contains the code of the *Person* class developed in Week 5 (refer to the code available on Blackboard for this). We can call the VS Solution containing this project *Week8*.

b) In this project, we define the *Program* class that contains the *Main* method as described below in the listing Program.cs.

c) We run the project *Person* and check the output.

d) We then create an additional project within the same Solution *Week8*, called *AddressProject*. The project contains the definition of the class *Address* provided below in the listing Address.cs.

e) In the *Person* project, we now need to **reference** the other project *AddressProject*. This can be done by right-clicking on the project (on the sidebar), following Add->Project Reference and finally selecting from the list the project we want to add a reference to (*AddressProject* in our case).

f) We modify the implementation of the *Person* class so that the *address* attribute is an object of the *Address* class instead of a *string*.

g) We need to verify that the *Address* class in the referenced assembly is visible from the other project and that the appropriate *namespace* is used when referring to it.

h) We run the *Main* of the *Program* class (of the *Person* project) again and check the output.

## Program.cs

```
namespace Person
{
   internal class Program
   {
      static void Main(string[] args)
      {
         Person[] people = new Person[3];
         people[0] = new Person("James", "Brown", 1990);
         people[1] = new Person("Frederic", "Chopin", 1810);
         people[2] = new Person("Tom", "Jones", 1940);

         // this will generate a null reference Exception
         // because the address attribute has not been set yet
         foreach (Person p in people)
            p.Display();

         people[0].SetAddress("30 Hampstead Ln; London; N6 4NX");
```

```
            people[1].SetAddress("25 Castlegate; Knaresborough; HG5 8AR");
            people[2].SetAddress("59 Pier Rd; Littlehampton; BN17 5LP");

            foreach (Person p in people)
                p.Display();

        }
    }
}
```

## Address.cs

```
using System;

namespace AddressProject
{
    public class Address
    {
        private int number;
        private string street;
        private string city;
        private string postcode;

        public Address(string address)
        {
            string[] tokens = address.Split(";");
            string firstLine = tokens[0];

            string[] firstLineTokens = firstLine.Split(" ");
            number = Convert.ToInt32(firstLineTokens[0]);
            street = firstLineTokens[1];

            city = tokens[1];
            postcode = tokens[2];

            // checks isValid and raise an error in case
        }

        // should check whether it is a valid address
        // always returns true for now
        private bool IsValid()
        {
            return true;
        }

        public override string ToString()
        {
            return $"Number: {number}, Street: {street}, City: {city}, Postcode: {postcode}\n";
        }
    }
}
```

## 2: Static Attributes (independent work)

Write the code of a *Ticket* class. Each ticket has its **unique** number (attribute), which is generated automatically when a new Ticket is created. The numbers are int values in ascending order starting from 1. Example:

- ticket 1
- ticket 2
- ticket 3
- ...
- ticket 100

*Hint: Look at the notes of Lecture 8 on static attributes to understand how to track the number of tickets sold.*

Test your Ticket class using the following *Program* class.

```
using System;

namespace Tickets
{
   class Program
   {
      static void Main(string[] args)
      {
         Ticket t1 = new Ticket();
         Console.WriteLine("Ticket number: " + t1.GetTicketNumber());
         Console.WriteLine("Tickets sold: " + t1.GetNumberSold());

         Ticket t2 = new Ticket();
         Console.WriteLine("Ticket number: " + t2.GetTicketNumber());
         Console.WriteLine("Tickets sold: " + t2.GetNumberSold());

         Console.WriteLine("Ticket number: " + t1.GetTicketNumber());
         Console.WriteLine("Tickets sold: " + t1.GetNumberSold());

         Ticket t3 = new Ticket();
         t3.PrintInfo();
      }
   }
}
```

*Output*

```
Ticket number: 1
Tickets sold: 1
Ticket number: 2
Tickets sold: 2
Ticket number: 1
Tickets sold: 2
ticket #3; 3 ticket(s) sold.
```

## 3: Static methods (independent work)

Implement the utility class *CalcManager* from this week lecture that provides the following static methods:

- IsEven(int n): returns true if n is even, otherwise it returns false.
- Cube(int n): returns the cube of n.
- Add(double[] x): returns the sum of the elements of the array provided as argument

Add the following Main to the class *CalcManager*:

```
static void Main(string[] args)
{
   int number = 3;
   double[] values = { 0.4, 3.5, 7.8, 0.5 };

   Console.WriteLine(IsEven(number));
   Console.WriteLine(CalcManager.IsEven(10));
   Console.WriteLine(CalcManager.Add(values));
}
```

And explain what the meaning of the above methods invocation is.

If the *CalcManager* class had a non-static attribute, would you be able to access it directly from any of those static methods you defined?

## 4: Static methods (at home)

Create a new project containing the *VendingMachine* and *Program* classes from Tutorial Week 7. Add a new class to the project, called *ProgramWithMethods*, that contains a refactored version of the *Program* class as follows, where **the code needs to be adapted to use multiple static methods**. For simplicity, no error and exception handling are done in the class *ProgramWithMethods*.

```
namespace VendingMachines
{
   internal class ProgramWithMethods
   {
      static void BuyCans(VendingMachine vm, int cansTobuy)
      {
         // Complete reusing relevant code from the old Program class
      }


      static void CheckMachine(VendingMachine vm)
      {
         // Complete reusing relevant code from the old Program class
      }


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

         VendingMachine m1 = new VendingMachine();
         VendingMachine m2 = new VendingMachine(7);

         // Setting machines names and locations
         m1.SetName("Machine 1");
         m2.SetName("Machine 2");

         m1.SetLocation("4th Floor");
         m2.SetLocation("Ground Floor");

         // Randomly adding cans to m1 (1 to 10)
         m1.AddCans(r.Next(1, 11));

         // printing machines status
         Console.WriteLine(m1.ToString());
         Console.WriteLine(m2.ToString());
         Console.WriteLine();

         // trying to buy a random number of cans from machine 1
         int tokensToInsert = r.Next(1, 10);
         BuyCans(m1, tokensToInsert);

         // trying to buy 1 can from machine 2
         BuyCans(m2, 1);

         // printing machines status
         Console.WriteLine(m1.ToString());
         Console.WriteLine(m2.ToString());

         // checking what machines need refilling
         CheckMachine(m1);
         CheckMachine(m2);
```

```
    }
  }
}
```

Note that it is possible to have multiple classes with a *Main* method within the same project. However, the .NET environment must know which one should be invoked when the project is started. This can be done by adding the following element to the Project XML file

<StartupObject>Namespace.Class</StartupObject>

Try this by running either the *Main* defined in the *Program* class and the *Main* defined in the *ProgramWithMethods* class.