1: Interfaces

A hospital needs a software system to print (show on the console terminal) information about letters and therapies associated with their patients.

Part 1 (with tutor)

Define an *IPrintable* interface that includes a *void Print()* method.

Write the code for two classes, *Letter* and *Therapy*, which implement the above interface. The *Letter* class defines a string *text* attribute, while the *Therapy* class includes a string attribute corresponding to the *name* of the therapy (e.g., "Flu Therapy") and an array of 10 string elements corresponding to the medications associated with that therapy.

IPrintable defines a design contract for both the *Letter* and *Therapy* (and potentially other) classes. Hence, they must implement the *Print()* method of the interface so that all the relevant details for an object of the corresponding class (i.e., *text* for *Letter*, *name* and *medications* for *Therapy*) will be shown on the screen when *Print()* is called.

We discussed in the lecture that abstract classes and interfaces are alternative ways of describing design contracts. In addition to normal instance methods, abstract classes can have abstract methods (without a body). Interfaces only have methods without a body (however, no abstract keyword is specified). How does this affect the implementation of the contract when using abstract classes or interfaces? Do you need to use the override keyword in both cases?

Test your code using the *PrinterProgram* class below:

```
namespace Hospital
{
   class PrinterProgram
   {
      static void PrintJob(IPrintable p)
      {
         p.Print();
      }

      static void Main(string[] args)
      {
         Letter letter = new Letter("text of the letter");

         string[] medications = new string[10];
         medications[0] = "Ibuprofen";
         medications[1] = "Paracetamol";
         medications[2] = "Vitamin D";
         Therapy therapy1 = new Therapy("Flu Therapy", medications);

         PrintJob(letter);
         PrintJob(therapy1);
      }
   }
}
```

Think about how the above *PrintJob* method has been designed and how this is related to the concepts of *abstraction*, *design contracts* and *polymorphism* seen in Weeks 10 and 11.

Part 2 (independent work)

Modify the previous *Therapy* class so that, in addition to the *IPrintable* interface, it also implements a new interface *IPrescribable* that includes the following methods:

- void PrescribeMedication(string m)
- void DeprescribeMedication(string m)
- bool IsPrescribed(string m)

2: Inheritance: construction of objects (independent work)

Part 1

Given the source code below, try to guess what the output of the program would be. Then run the program and verify your guess. Make sure that you fully understand what is happening.

```
using System;
namespace Constructors1
{
   public class Cell
   {
      public Cell()
      {
         Console.WriteLine("Cell constructor called");
      }
   }

   public class TinyCell : Cell
   {
      public TinyCell()
      {
         Console.WriteLine("TinyCell constructor called");
      }
   }

   public class MicroscopicCell : TinyCell
   {
      public MicroscopicCell()
      {
         Console.WriteLine("MicroscopicCell constructor called");
      }
   }

   public class CellTest
   {
      public static void Main(string[] args)
      {
         Cell c = new MicroscopicCell();
      }
   }
}
```

Part 2

Now modify the above source code as follows:

```
using System;

namespace Constructors2
{
```

```
   public class Cell2
   {
      private int x;

      public Cell2(int x)
      {
         this.x = x;
         Console.WriteLine("Cell2 constructor called");
      }
   }

   public class TinyCell2 : Cell2
   {
      public TinyCell2()
      {
         Console.WriteLine("TinyCell2 constructor called");
      }
   }

   public class MicroscopicCell : TinyCell2
   {
      public MicroscopicCell()
      {
         Console.WriteLine("MicroscopicCell constructor called");
      }
   }

   public class CellTest2
   {
      public static void Main(string[] args)
      {
         Cell2 c = new MicroscopicCell();
      }
   }
}
```

You will see an error when you try to run the program's second version. Do you understand what is happening? (hint: the issue is related to the chained invocation of constructors through the inheritance tree and the number of arguments).

Try to fix the errors by:

1. modifying the definition of the *Cell2* class;
2. leaving the definition of the *Cell2* class as it is and modifying the definition of the *TinyCell2* class constructor.

3: Object class: ToString and Equals methods (at home)

Consider the *BankAccount* class developed in our previous tutorials, whose code is reported below:

```
using System;

namespace Bank
{
   public class BankAccount
   {
      private string number;
      private double balance;
      private bool open;
```

```
private static int accountsCreated = 0;

public BankAccount(string num, double bal)
{
   number = num;
   balance = bal;
   open = true;
   accountsCreated++;
}

public BankAccount(string num)
{
   number = num;
   balance = 0;
   open = true;
   accountsCreated++;
}

public static int GetAccountsCreated()
{
   return accountsCreated;
}

public void Deposit(double amount)
{
   balance += amount;
}

public bool Withdraw(double amount)
{
   if (amount < balance)
   {
      balance -= amount;
      return true;
   }

   return false;
}

public double GetBalance()
{
   return balance;
}

public string GetNumber()
{
   return number;
}

public void Close()
{
   balance = 0;
   open = false;
}

public void MoveAccount(BankAccount otherAccount)
{
   otherAccount.Deposit(balance);
   Close();
}

public bool IsOpen()
{
```

```
            return open;
        }
    }
}
```

Part 1

Inside the above *BankAccount* class, implement an overridden version of the *ToString* method (already defined in the *Object* class). The overridden version should return a string consisting of the relevant information associated with a *BankAccount* object, i.e., number, balance and whether the account is open. Test your implementation by passing a reference to a *BankAccount* object to the *Console.WriteLine* method and check that *ToString* is invoked automatically.

Part 2

Consider the code reported below and define an overridden version of the *Equals* method inside the *BankAccount* class.

```
using System;

namespace Bank
{
    class Program
    {
        static void Main(string[] args)
        {
            BankAccount account1 = new BankAccount("A0123", 1000.50);
            Console.WriteLine(account1); // will invoke ToString() on account1

            BankAccount account2 = new BankAccount("A0123", 1000.50);
            Console.WriteLine(account2);

            if (account1 == account2) // references are different
            {
                Console.WriteLine("[1] account1 reference is equal to account2 reference");
            }

            if (account1.Equals(account2)) // Equals will check the attributes and return true
            {
                Console.WriteLine("[2] account1 is equal to account2");
            }

            account1 = account2;

            if (account1 == account2) // references are the same now
            {
                Console.WriteLine("[3] account1 reference is equal to account2 reference");
            }

            if (account1.Equals(account2)) // Equals will check the attributes and return true
            {
                Console.WriteLine("[4] account1 is equal to account2");
            }

        }
    }
}
```

If the *Equals* method is implemented properly, the above program will produce the following output when executed:

```
[Account Number: A0123, Open: True, Balance: 1000.5]
[Account Number: A0123, Open: True, Balance: 1000.5]
[2] account1 is equal to account2
[3] account1 reference is equal to account2 reference
[4] account1 is equal to account2
```