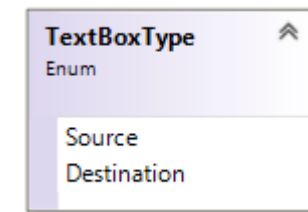
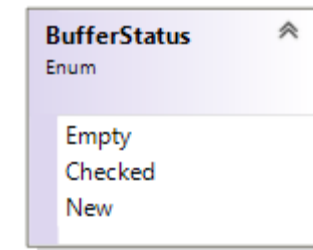
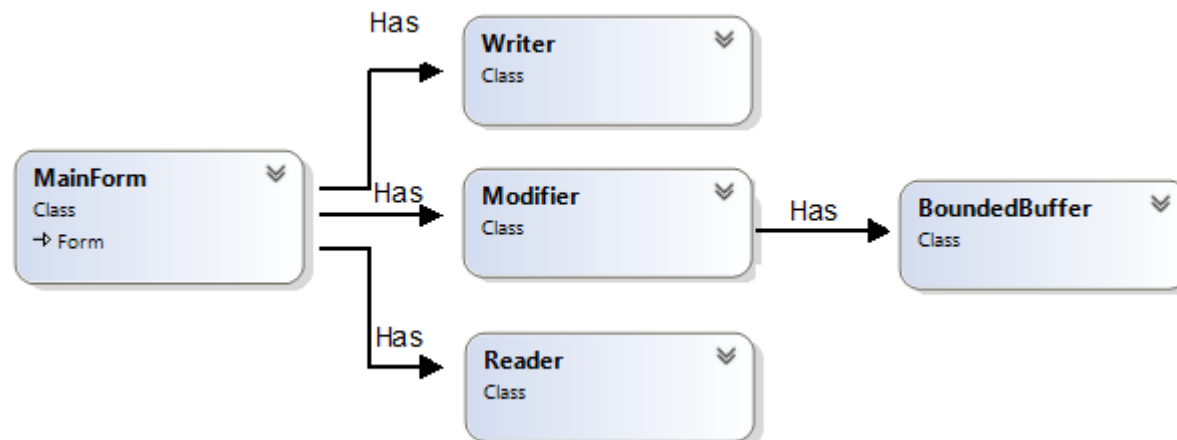


## MONITORS – HELP AND GUIDANCE

### 1. FILE COPIER

The instructions given here are based on a C# project but it should not make much a difference in applying the methods in a Java project. Below is a class diagram showing the classes you might need to write, a GUI (MainForm) a bounded buffer, and a writer, a reader and a modifier class.



#### 1.1 The BoundedBuffer class:

Begin writing the **BoundedBuffer** class. This class serves mainly as a container for shared data and methods that process the shared data. The following code snippet is an example showing the shared variables as instance fields. Remember that you can make your own design and optimize the code if you have better ideas. Use of so many single instance variables leaves space for an optimization!

```
public class BoundedBuffer
{
    /// <summary>
    /// Fields
    /// </summary>
    private string[] strArr;           // The actual string buffer array
    private BufferStatus[] status;     // An array of BufferStatus objects,
                                      // one for each element in buffer

    private int max;                  // Elements in buffer
    private int writePos;              // The position pointers for each thread
    private int readPos;
    private int findPos;

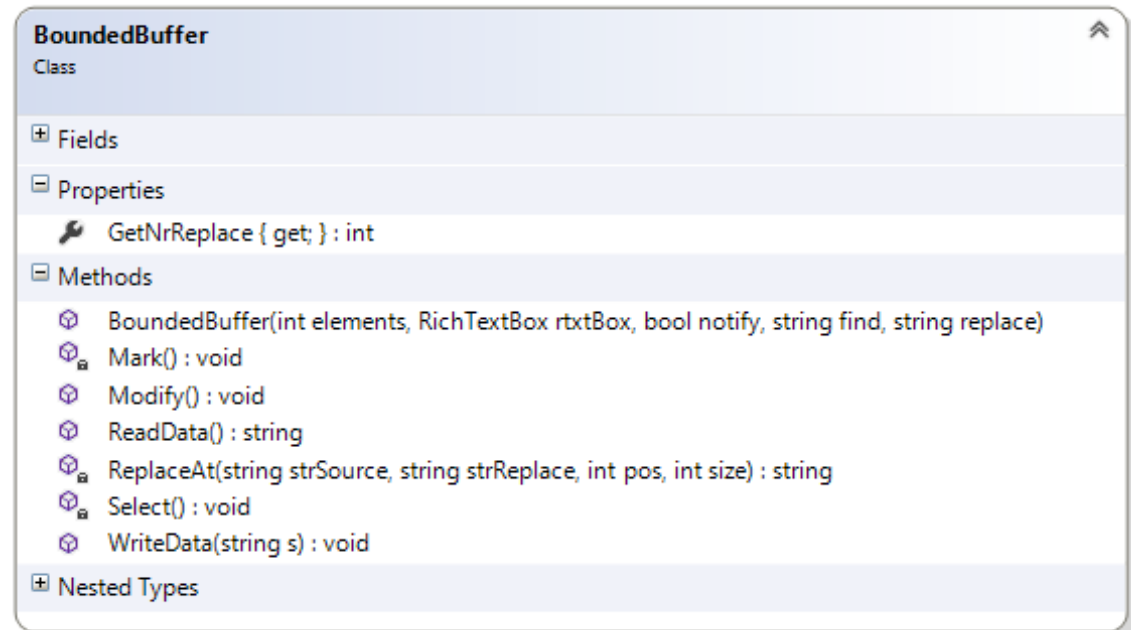
    private RichTextBox rtxBox;       // The rich text box to mark in
    private string findstring;        // The string to search for, if any
    private string replacestring;     // The replace string, if any
    private int start;                // The start position in textbox for marking
    private int nbrReplacements;      // Replacement counter
    private bool notify;              // User notify
    private object lockObject;        // For mutual exclusion

    private delegate void Marker();    // Delegate used in textbox invoke for MARKING in Rtx box
    private delegate void Selector();  // Delegate used in textbox invoke for SELECTING in Rtx box

    /// <summary> Parametric constructor
    1 reference
    public BoundedBuffer (int elements, RichTextBox rtxtBox, bool notify, string find, string replace)
    {
```

The instance variables are shared data and all threads will have access to them. Use local variables wherever possible they will not be shared among the threads. The delegates are a part of the C# solution!

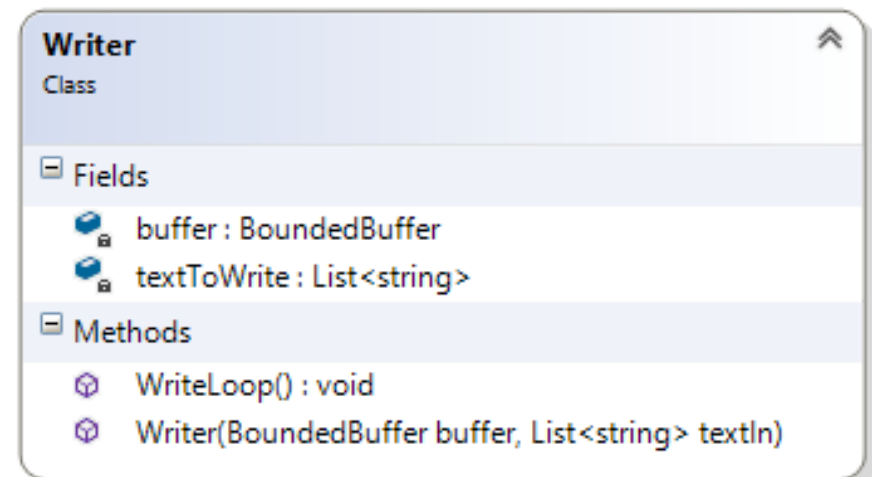
Define and implement methods for reading, writing and modifying the buffer data and other necessary operations. The following image presents a suggestion of some useful methods.



```
class BoundedBuffer {
    // Fields
    // Properties
    GetNrReplace { get; } : int
    // Methods
    BoundedBuffer(int elements, RichTextBox rtxtBox, bool notify, string find, string replace)
    Mark() : void
    Modify() : void
    ReadData() : string
    ReplaceAt(string strSource, string strReplace, int pos, int size) : string
    Select() : void
    WriteData(string s) : void
    // Nested Types
}
```

## 1.2 The Writer Class

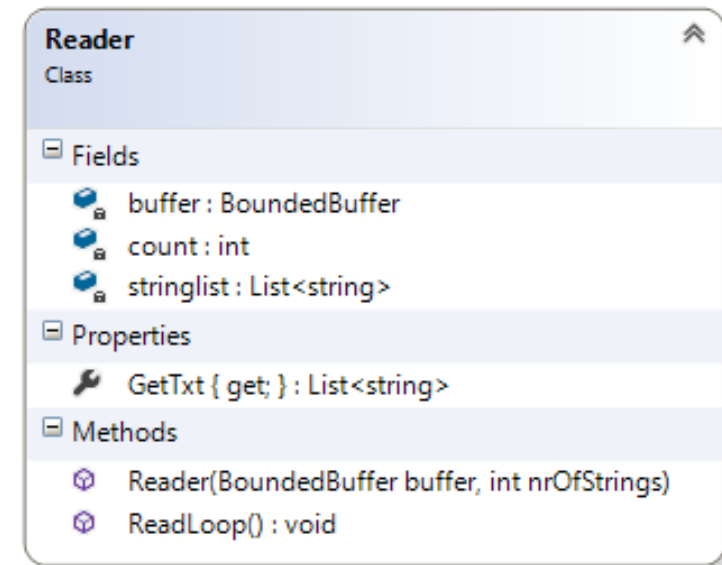
Create the **Writer** class. It will be run by a thread and needs to have a reference to the shared buffer and also the list to be filled with the strings from the text file. It then has a simple method for the writer thread to run. The method simply loops through the strings and calls the write method of the bounded buffer.



```
class Writer {
    // Fields
    buffer : BoundedBuffer
    textToWrite : List<string>
    // Methods
    WriteLoop() : void
    Writer(BoundedBuffer buffer, List<string> textIn)
}
```

### 1.3 The Reader Class

The **Reader** class is about the same as the **Writer**, but it needs to know the total number of strings to read. This method will be run by the reader thread and all it needs to do is to simply run a loop and call the reader part of the buffer to read a line in each iteration.

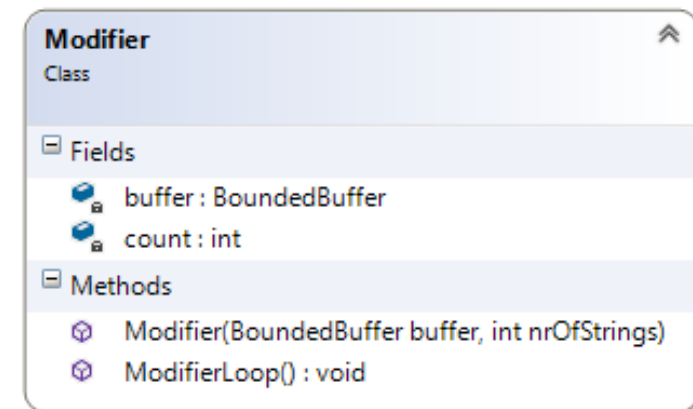


### 1.4 The Modifier Class

The modifier class is just like the reader class. It has a method that calls the buffer-class' **modify** method. The entire class is given to you as a present, and this time in Java! 🎁🤖

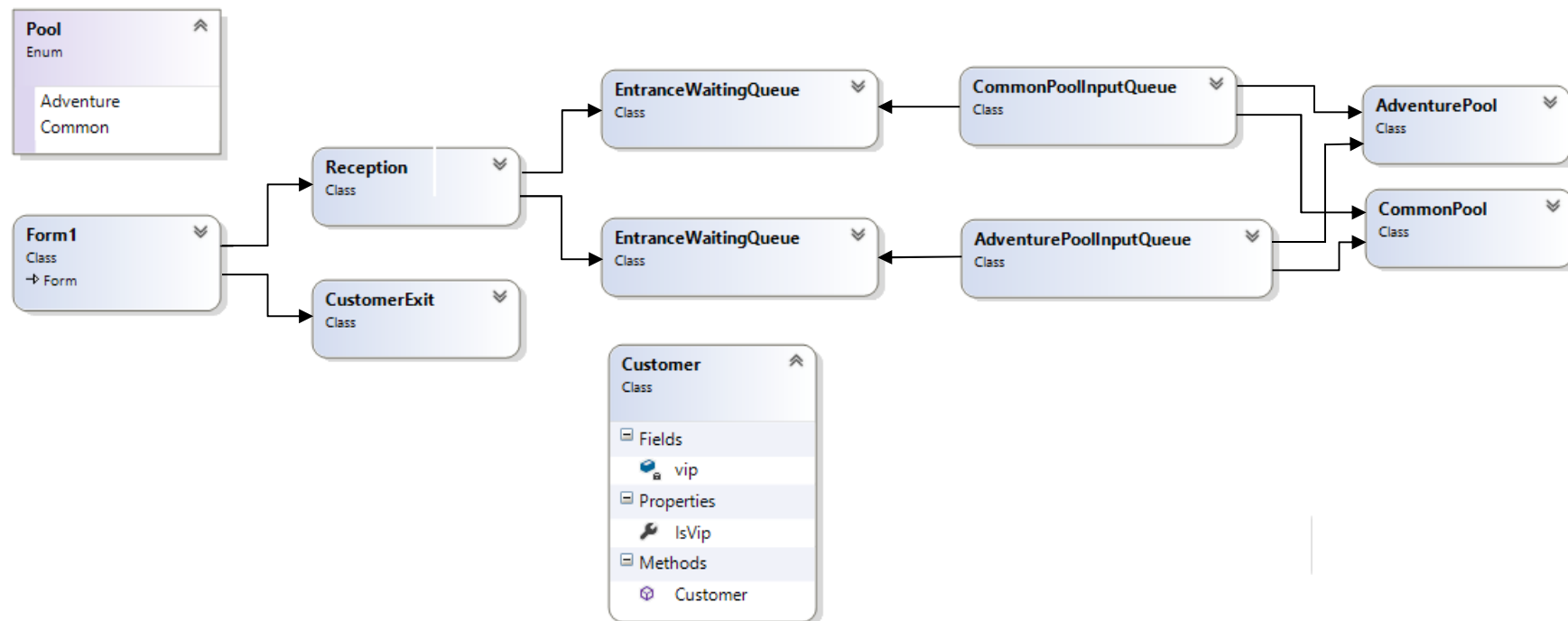
In C#, the `RichTextBox` control has two properties, **SelectionStart**, **SelectionLength**, that can be used to select a part of a string. It also has the properties **SelectionBackColor** and **SelectionColor** that can be used to mark (highlight) the selected substring.

If you face difficulties, skip this feature and it is ok if you don't get the visual part displayed correctly.



## 2. SWIMMING POOL

This diagram below shows the classes involved. A description of each one is done in following text.

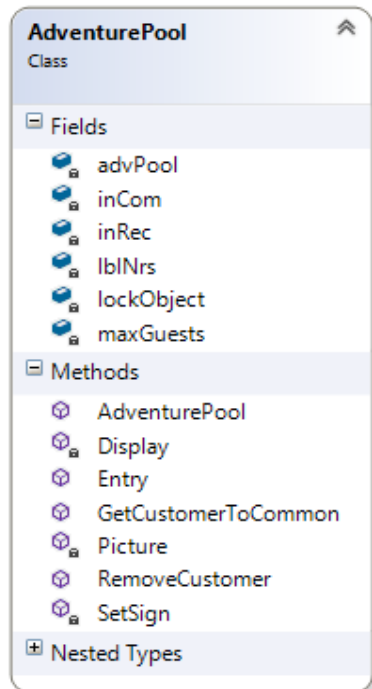


### 2.1 The Customer class

Start writing this simple class first; it only contains a **bool**, here called **vip**. When a customer enters the adventure pool a customer with **vip** set to true is created and put in pool's queue, meaning this customer can be in both pools. A customer entering the common pool has this field set to **false**, allowing him/henne to only use the common pool.

## 2.2 The AdventurePool and CommonPool classes

These classes are the shared object of this application. They contain a queue of customers and some handy methods. A sample (part) of the Adventure pool is given here.



```

/// <summary>
/// The adventure pool shared data
/// </summary>
public class AdventurePool
{
    private Queue<Customer> advPool;           // The queue of customers
    private int maxGuests;                     // The limit
    private Label lblNrs;                      // Label for writing
    actual nr of customers
    private object lockObject;                 // For mutual exclusion
    private PictureBox inRec, inCom;           // Just for changing signs

    private delegate void Disp(int i);         // For interacting with GUI
    private delegate void SetPicture(Image img, PictureBox box);

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="max">Limit</param>
    /// <param name="lbl">nr label</param>
    /// <param name="recbox">sign from reception</param>
    /// <param name="combox">sign from common pool</param>
    public AdventurePool(int max, Label lbl, PictureBox recbox, PictureBox combox)
    {
        advPool = new Queue<Customer>();
        maxGuests = max;
        lblNrs = lbl;
        lockObject = new object();
        inRec = recbox;
        inCom = combox;
    }
}

```

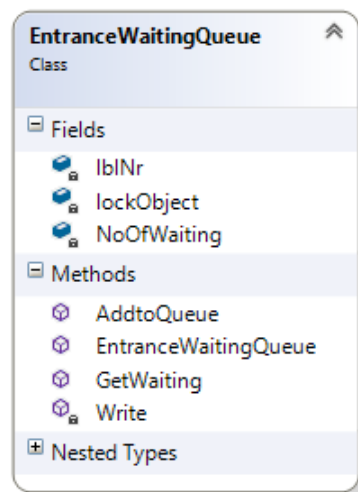
## 2.3 The AdventurePoolInputQueue and CommonPoolInputQueue classes

Here is the main working methods. They use the pools as shared resources, and the threads must wait if the pool is full.

One implementation of the **AdventurePoolInputQueue** class is shown here. The **CommonPoolInputQueue** is similar.

## 2.4 The EntranceWaitingQueue class

This class is just the actual “person counter” (**NoOfWaiting**) that the reception uses to increase each time a new customer arrives, and the **PoolInputQueues** use for decrease when adding a new Customer to the pools.



```

/// <summary>
/// The class handling input to Adventure pool
/// </summary>
public class AdventurePoolInputQueue
{
    private AdventurePool aPool;    // The common data
    private CommonPool cPool;       // Can enter from common pool
    private EntranceWaitingQueue aWait; // or from queue from reception
    private Random rnd;

    /// <summary>
    /// Constructor
    /// </summary>
    /// <param name="apool">Adventure pool</param>
    /// <param name="cpool">Common pool</param>
    /// <param name="q">adventure waiting queue</param>
    public AdventurePoolInputQueue(AdventurePool apool, CommonPool cpool,
    EntranceWaitingQueue q)
    {
        aPool = apool;
        cPool = cpool;
        rnd = new Random();
        aWait = q;

        /// <summary>
        /// The adventure input thread running method
        /// </summary>
        public void RunAdventureInput()
        {
            Customer cust;
            while (true)
            {
                // If to get from reception
                if ((Pool)rnd.Next(2) == Pool.Adventure)
                {
                    if (aWait.GetWaiting())
                        aPool.Entry(new Customer(true)); // Only enter if queue not
empty, this customer IS VIP
                }
                else
                {
                    // Sometimes try to get a customer from Common pool, but only if
he is VIP
                    if ((cust = cPool.GetCustomerToAdventure()) != null)
                        aPool.Entry(cust);
                    Thread.Sleep(1000);
                }
            }
        }
    }
}

```

## 2.5 The Reception class

This is also a simple class only holding two **EntranceWaitingQueues** and a thread loop that randomly puts customers in the waiting queues (increase a counter **NoOfWaiting** in the class **EntranceWaitingQueue**) when the facility is open.

## 2.6 The CustomerExit class

This last class is also very simple, it just holds a thread loop that (slowly) exits customers from the pools.

If the user closes the pools, the reception thread goes idle, but if the **NoOfWaiting** in the **EntranceWaitingQueues** are NOT zero, customers from these are still able to enter the facility until all queues are zero and all customers has left the facility.

The Main class only creates the instances needed for each class and put a reference in the construction of other instances. Then the four thread is created and started. All this is done as initialization. The only more thing for main to do is to handle keypress, that just toggles the texts and the running parameter for the reception thread.

The hints given here are just suggestions. If you come up with a better or another solution it is ok, as long as you use monitors for synchronization.

Good Luck!

Farid Naisan,  
Course Responsible and Instructor

