# Sofia University
# Department of Mathematics and Informatics

**Course** : **OO Programming C#.NET**

**Date**: **January 13, 2019**

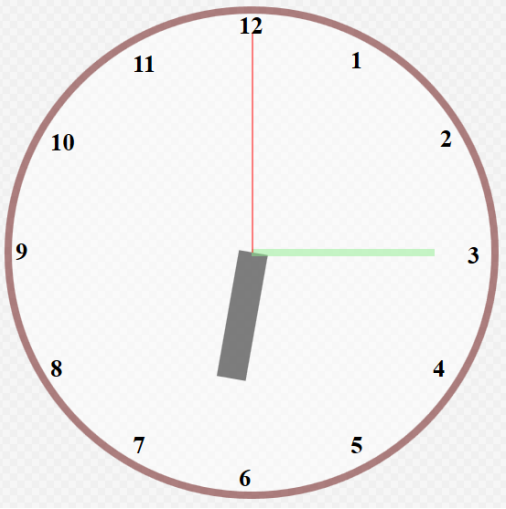
**Student Name:**

**Lab No. 14a**

**Submit the all C# .NET files developed to solve the problems listed below. Use comments and** Modified-Hungarian notation.

**Problem No.1a**

Design the following analog clock **in Blend for Visual Studio 2017** and package it as a WPF user control




A) **Create a Thread** to update the current time every 1000 ms and provide methods Resume and Suspend to start and stop the thread. Create a WPF application making use of this User control by adding buttons Start and Stop to start and stop the clock from running.

B) **Use a** `System.Windows.Threading.DispatcherTimer` to update the current time every 1000 ms and provide methods Resume and Suspend to start and stop the thread. Create a WPF application making use of this User control by adding buttons Start and Stop to start and stop the clock from running.

C) **Use a** `System.Timers.Timer` to update the current time every 1000 ms and provide methods Resume and Suspend to start and stop the thread. Create a WPF application making use of this User control by adding buttons Start and Stop to start and stop the clock from running.

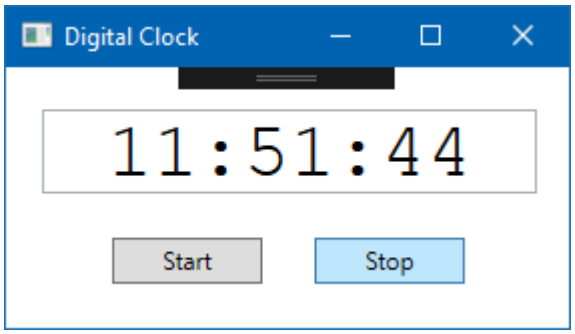
**Problem No.1b**

Design the following digital clock **for Visual Studio 2017** and package it as a WPF user control

A) **Create a Thread** to update the current time every 1000 ms and provide methods Resume and Suspend to start and stop the thread. Create a WPF application making use of this User control by adding buttons Start and Stop to start and stop the clock from running.

B) **Use a** `System.Windows.Threading.DispatcherTimer` to update the current time every 1000 ms and provide methods Resume and Suspend to start and stop the thread. Create a WPF application making use of this User control by adding buttons Start and Stop to start and stop the clock from running.

C) **Use a a** `System.Timers.Timer` to update the current time every 1000 ms and provide methods Resume and Suspend to start and stop the thread. Create a WPF application making use of this User control by adding buttons Start and Stop to start and stop the clock from running.




**Problem No.2 (study WPFLoginControl.pdf)**

Create a WPF **UserControl** called **LoginPasswordUserControl.** The **LoginPasswordUserControl** contains a **Label (lblLogi**n) that displays **String "Login:**", a **Text-Box (txtLogin**) where the user inputs a ***login name***, a **Label (lblPassword**) that displays the **String "Password:"** and finally, a **TextBox (txtPassword**) where a user inputs ***a password*** (*do not forget to set property* ***PasswordChar*** *to* ***"*"*** *in the* ***TextBox****'s* ***Properties*** *window*). **LoginPasswordUserControl** must provide **Public** read-only properties **Login** and **Password** that allow an application to retrieve the user input from **txtLogin** and **txtPassword. There** should be **button OK** to export to an application the values input by the user in **LoginPasswordUserControl. Accordingly, button Cancel, clears the strings in the textboxes (txtLogin, txtPassword)** and **exports empty strings** to the **host application** (the one that embeds the user control)

**Write** a C#.NET WPF application to **test** the ***LoginPasswordUserControl***- the main form of the application contains an instance of the ***UserControl***. In **addition** to the user control the main form should h**ave a *ListBox* and a *Combobox*.** On clicking the **OK** of the ***LoginPasswordUserControl*** the **strings for the username/password** input by the user in the user control are added accordingly to the ***ListBox*** and a ***Combobox***. On clicking the ***Cancel*** of the ***LoginPasswordUserControl*** a ***MessageBox*** displays **a warning message** that no username/password are entered.

**Problem No.*3***

**Complete tutorial**s:

***Simple Async Await Example for Asynchronous Programming***

***Implementing a Form That Uses a Background Operation***

**Problem No.*4***

Here is an example of multi currency conversion is simulated; which does processing of currency conversion. An exchange unit is modeled by a class called `ExchangeUnit` which carries the information of input & output currency and amount to be converted. Multiple objects of `ExchangeUnit` are created with different currency information and are fed into a `ThreadPool` for doing the multiple conversions. The `ExchangeUnit` class provides a method called `ThreadPoolCallback` that performs the conversion. An object representing each exchange required is created, and the `ThreadPoolCallback` method is passed to `QueueUserWorkItem`, which assigns an available thread in the pool to execute the method.

There is another class called `MulticurrencyExchanger` which maintains an internal list of latest conversion rates among the currencies. This list is periodically updated by a different thread which arrives at the percentage change in exchange rates based on a random number principle. In real world application latest exchange rates would be fed typically fed by a web service

**Problem No.*5***

**Given** two **matrices *A*** and ***B,*** where ***A is a* matrix** with ***M* rows** and ***K* columns** and **matrix B** contains ***K* rows** and ***N* columns**, the **matrix product** of ***A*** and ***B*** is **matrix C,** where ***C*** contains ***M* rows** and ***N* columns**. The entry in ***matrix C*** for **row *i* column *j* (** $C_{i,j}$ **)** is the sum of the products of the elements for row ***i*** in matrix ***A*** and column ***j*** in matrix ***B***. That is,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} * B_{n,j}$$

For **example**, if ***A*** were a ***3-by-2* matrix** and **B** were a ***2-by-3*** matrix, element $C_{3,1}$ would be the sum of $A_{3,1} * B_{1,1}$ and $A_{3,2} * B_{2,1}$.

**For this problem**, **calculate each element** $C_{i,j}$ **in a separate worker thread**. This will involve **creating** *M x N* **worker threads**. The main- or parent- thread will **initialize the matrices *A* and B** and **allocate sufficient memory** for matrix C, which will **hold the product**

of matrices ***A* and *B***. These matrices should be **declared as global data** so that each worker thread has access to A, B, and C.

Matrices *A* and B can be initialized, as shown below:

```
int  A[][] ={{1,4},{2,5},{3,6}};

int  B[][] = { {8,7,6}, {5,4,3} };

int  C[][];
```

Alternatively, they can be p**opulated by reading in values from a file.**

**Passing Parameters to Each Thread**

The **parent thread** will create ***M x N* worker threads**, passing each worker the values of ***row i*** and ***column j*** that it is to use in **calculating the matrix product**. This requires passing two parameters to each thread. One approach is for the main thread to create and initialize the matrices *A, B,* and C. This **main thread will then create the worker threads**, passing the three matrices- along with row ***i*** and column ***j***- to the constructor for each worker. Thus, the outline of a worker thread appears as follows:

```
public class WorkerThread

{

      private int row;

      private int col;

      private int [][] A;

      private int [][] B;

      private int [][] C;

      public WorkerThread(int row, int col, int[][] A,
                          int[][] B, int[][] C) {

      this.row = row;

      this.col = col;

      this.A = A;

      this.B = B;

      this.C = C;

}
```

```
public void compute() {

    /* calculate the matrix product in C[row][col] */

    }

}
```

### Waiting for Threads to Complete

Once all worker threads have completed, the main thread will output the product contained in matrix *C*. This requires the main thread to wait for all worker threads to finish before it can output the value of the matrix product. Several different strategies can be used to enable a thread to wait for other threads to finish, as shown below.

```
using System;
using System.Threading;

class IsThreadPool
{
    static void Main()
    {
        AutoResetEvent autoEvent = new AutoResetEvent(false);

        Thread regularThread =
            new Thread(new ThreadStart(ThreadMethod));
        regularThread.Start();
        ThreadPool.QueueUserWorkItem(new WaitCallback(WorkMethod),
            autoEvent);

        // Wait for foreground thread to end.
        regularThread.Join();

        // Wait for background thread to end.
        autoEvent.WaitOne();
    }

    static void ThreadMethod()
    {
        Console.WriteLine("ThreadOne, executing ThreadMethod, " +
            "is {0}from the thread pool.",
            Thread.CurrentThread.IsThreadPoolThread ? "" : "not ");
    }

    static void WorkMethod(object stateInfo)
    {
        Console.WriteLine("ThreadTwo, executing WorkMethod, " +
            "is {0}from the thread pool.",
            Thread.CurrentThread.IsThreadPoolThread ? "" : "not ");

        // Signal that this thread is finished.
        ((AutoResetEvent)stateInfo).Set();
    }
}
```

OR---------- >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

```
class Test
{
    static TimeSpan waitTime = new TimeSpan(0, 0, 1);

    public static void Main()
    {
        Thread newThread =
            new Thread(new ThreadStart(Work));
        newThread.Start();

        if(newThread.Join(waitTime + waitTime))
        {
            Console.WriteLine("New thread terminated.");
        }
        else
        {
            Console.WriteLine("Join timed out.");
        }
    }

    static void Work()
    {
        Thread.Sleep(waitTime);
    }
}
```