# Assignment 6: To Do Reminder

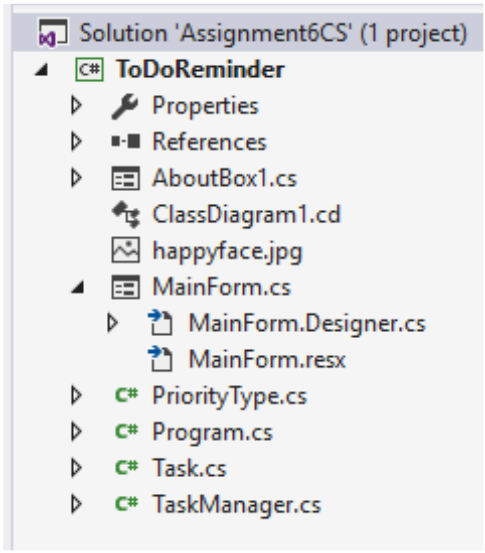
## 1. Where to begin?

The figure below show a suggest project structure

We need at least 5 types, four classes and one enum in addition to the Program class that VS creates for us as the start class. All these together with eventually other component like images and data files work together to solve the given problem.

**Some Notes**:

- Paragraphs or sections that are marked as optional are not a mandatory for a Pass grade.
- The modifier protected is used in connection with inheritance and is not included in this course although the expression may be mentioned in the descriptions.
- Static classes and methods are not taken into account in the discussions.
- The **FileManager** class and all about writing and reading data to and from a text file is described in a separate document and is available on the module. Therefore, it is excluded from the descriptions in this document.




1.1 **Task class:** that handles all about a task. Every reminder task that you save in the program becomes a separate object (or instance) of the Task class. The name that you assign to the object at declaration time (**currentTask**) will then refer to a space in the computer's memory where the data for the task will be saved when the program runs. In other words, a variable of an object type is called a reference variable that, unlike value type variables (int, double etc.) does not contain the data directly, but it can be used to access the data (and operations), e.g. **currentTask**.**Description**)

A task object is created using the Task class as template. The following statement, declares a reference variable (currentTask).

```
Task currentTask = new Task();
```

An object of the Task class is created (at the right side of the equal sign) using the keyword new. This keyword calls the constructor of the Task class **Task( ). Task** without the parentheses ( ) is the name of the class but with the parentheses, it indicates a call

to the constructor of Task. Furthermore, empty parentheses implies that the default constructor of the **Task** class is used.

The **Task** class being a template is going to remain a part of the code (instructions at run-time), while each object of the task will live in the computer's memory (as long as it is referenced, used by other objects) at run-time. A class is a definition but an object is something that exists physically as a unique thing. An object in the computer's memory can be considered as a virtual physical thing.

So the first thing to do is to design the **Task** class, i.e. determine the instance variables and methods so the objects of the class can do its job (details below).

1.2 **TaskManager class:** This class is responsible for maintaining a list of Task objects. Use a List<Task> list to save a Task object which is prepared in **MainForm**. The manager class should have total responsibility to manage the list of tasks (a form of registry), add a new object in the list, change or delete an existing task, and so on. However, for operations that involve a particular task in the list, the manager should use the Task-object's methods.

1.3 **FileManager** class: In this class, we encapsulate operations related to writing and reading text files. We write a method to save all the tasks that are currently saved in our task list (object of **TaskManager**) permanently to a file on the hard disk of the computer. We write another method that reads a data from the text file and stores them back into the task list in the program (object of **TaskManager**).

1.4 **MainForm** class: This class is needed to handle all interactions (input/output) with the user. It has a visual part and a code part. The visual part is the Form and its components (controls) like Buttons, TextBoxes, Labels, and LisstBoxes which are mainly used to allow the user to input values, and display output.

**MainForm** makes use of an object of the TaskManager (taskManager). It creates a an object of Task class, passes the user data to this object, and then save the object in the **taskManager** object. Maiunform requests **taskManager** to calculate some data and produce output, when needed. **MainForm** then shows the output using components like a **Label** or **ListBox** for the user.

1.5 **PriorityType** enum (enumeration): This type helps us define some related constants by which we can represent different types of priority values, high priority, low priority, etc. Every task can have an attribute that can store information about which priority the task is assigned. The value should be only one of the members defined in the enum. So, the **Task** class will need an object of the **PriorityType**. Even **MainForm** needs to use this enum in order to display options to the user. When the user selects one priority option, the selected value is to be saved in the task object.

1.6 **AboutBox** class (optional): to write some information about your application. This class is actually a Windows Form class but Visual Studio has prepared as a predesigned

template that you can add to your project in the same way you add a new Form. Click on the Project name in VS, select Add New Item and choose **AboutBox**.

Let's start with the most independent type of the above classes, and that is the **PriorityType**. This type is not dependent on the other three. It is used by the **MainForm** and the **Task** classes. We could also begin with the **AboutBox**, which does not need any of the other classes.

## 2. The enum PriorityType

We define those priority types that we would like to work with in our application. The figure shows how you can construct this type. In this **enum**, the value **Very_important** gets an underlying value equal to zero by default. The next member **Important** equals to 1 and the rest follows the same pattern, values increment by one. The underscore in some names is because members are constants and follow the rules of naming of identifiers, i.e. names cannot have a space or other symbols.

```
public enum PriorityType
{
    Very_important,
    Important,
    Normal,
    Less_important,
    Not_important
}
```

Notice also that **enum** members are separated by a comma in C#. You can write all the members on one or more lines, but defining each member on a separate line makes your code more readable. This way, you also can provide a comment to each member on each line.

## 3. The Task class (Task.cs)

The next class we can write is the Task class. When you begin writing a new class, you should have the following checklist in mind:

1. **Fields** (private or protected (inheritance not included in this course), – never public) – to store data.
2. **Constructor**(s) – at least one, always required when an object of the class is created.
3. **Properties** – to give access to the values stored in the **fields**.
4. Other methods – to perform other tasks, one method per task (including a **ToString ()** method).

3.1 **Fields**: In the Task class there is actually only a few input values that are required. There is no need for storing internal and output data. These can be returned as output from the methods that calculate them (see Figure 1).

3.2 **Constructor(s):** At least one must be present in every class. If you don't provide any constructor at all, the compiler will create a default constructor and includes it in the compiled code. A default constructor is a constructor that has no parameters.

A constructor is a special method that has two main properties that distinguish it from standard methods.

1. It has the same name as the class.
2. It has no return type – not even void.

A class is only a template. Other classes that need to use the services of the class need to instantiate (create an object of) the class[1]. This is done by using the keyword "new".

```
Task currentTask = new Task( );
```

When the above line of code is executed at run-time, the constructor is invoked automatically. This method may look something like this:

```
public Task()
{
      //initialization cod
}
```

A constructor is used to initialize instance variables and do other initializations if needed. Figure 1 shows the fields of **Task** class and the default constructor. Instance variables are initialized to their default values by the compiler. However, we can initialize the fields to other values.

```
public class Task
{
    private DateTime date;  //date and time for a task
    private string description;  //Task's notes
    private PriorityType priority;  //priority selected from the enum

    //Default constructor
    public Task()
    {
        //set a default priority type
        priority = PriorityType.Normal;
    }
```

Figure 1: Fields and the default constructor

A class can also have other constructors with parameters to initialize some or all of the instance variables. The constructor must follow the rules of method overloading, i.e. they must differentiate in the order or type of parameters.

[1] Classes with **static** methods are exempted from this rule and are not included in this discussion.

```
24  /// <summary>
25  ///Constructor with one paramter
26  ///Calling the constructor with 4 parameters, sending the one parameter
27  ///that comes from the caller (taskDate) and three default value.
28  ///The other constructor is called using the keyword this.
29  /// </summary>
30  /// <param name="taskDate">Input coming as a parameter from the caller.</param>
31  public Task(DateTime taskDate) : this(taskDate, string.Empty, PriorityType.Normal)
32  {
33      //no code needed.
34  }
35  public Task(DateTime taskDate, string description, PriorityType priority)
36  {
37      this.date = taskDate;
38      this.description = description;
39      this.priority = priority;
40  }
```

Figure 2: Chain-calling constructors

Counting the default constructor, the user of this class (MainForm or TaskManager) has three choices to create an object of the Task class.

```
Task task1 = new Task(); //calling default constructor
Task task2 = new Task(DateTime.Now);
Task task3 = new Task(DateTime.Now, "Staff Meeting",
                                    PriorityType.Important);
```

**Take a note** as to how the constructor on line 31 in the above code clip calls the other constructor on line 35 which has four parameters. The first constructor receives one parameter from the caller which it passes to the next constructor together with three default values.

3.3 **Properties** – to give access to private fields of the class.

The setter and getter methods introduced in the previous module are standardized as Properties in C#. In the code example that follows, the property Description (with the capital 'D') is intended to be connected to the instance variable description which is a string and so is the return type of the property.

```
//Prperty connected to the field description
public string Description
{
    get { return description; }
    set
    {
        if (!string.IsNullOrEmpty(value))
            description = value;
    }
}
```

Capital **D**

small **d**

**Note**: If you make the common "beginner's" mistake of writing "return **D**escription" in the get-accessor, the program will crash as the method repeats calling itself indefinitely.

The identifier value in the above code has the same type as the property's return value and that is string.

A property is called much in the same way as using a variable (although a property is a method in its underlying structure and not a variable). In the following line of code, the set-member of the **Description** property is called (by the compiler) because the call to the **Description** property occurs at the left side of the equal sign. The property is receiving a new value, which is at the right side of the equal sign, (**txtDescription.Text**).

```
currentTask.Description = txtDescription.Text;
```

As you should notice from the **Description** property code, when the property is assigned a new value, the **set**-member changes the value of the field **description**.

In the statement below, the get-member of the **Description** property is called because the call to **Description** is on the right-side of the statement. The get-property as seen from the code of the property, returns the value of the field **description** and is received by a variable in the caller method.

```
txtDescription.Text = currentTask.Description;
```

The variable value has a string type in the Description property, but now, in this property, it gets the type **PriorityType** automatically. The variable value can be used exactly as any other variable of the **PriorityTypes,**e.g. accessing its members.

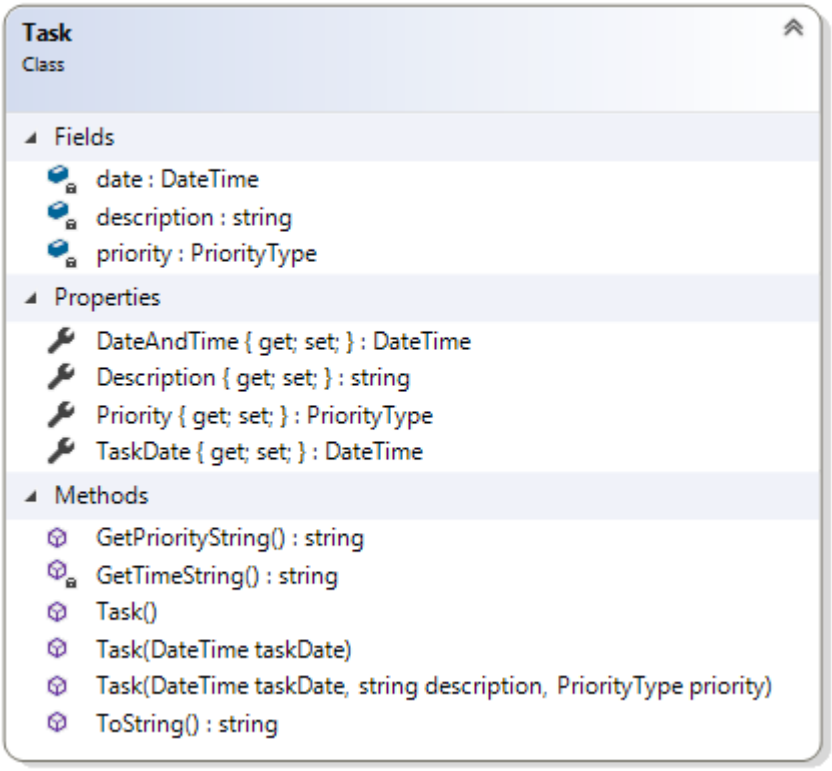
```
public PriorityType Priority
{
    get { return priority; }
    set { priority = value; }
}
```

Write properties to provide access to other fields in the same way as explained in above.

### 3.4 Other methods

In every class you usually also need to write standards methods (other than constructors and properties). These are methods that are either void not returning a value, or they have a return type, with or without method parameters. Some methods as suggested in the class diagram below are **GetPriorityString**, **GetTimeString** and **ToString** (The methods **Task** with no return type are constructors).

**GetPrioirityString()** (optional): is a method that returns a **string** composed of the name of the priority of the task replacing the _ character with a blank space. If **priority** happens to be **Less_important**, this function converts the member to the corresponding string **"Less important**" and returns this string.




If you don't want to do the above, the **ToString** method of the nstance variable **priority** gives the name of the member as it is defined

**GetTimeString():** this method returns a string containing the time saved in **date** field formatted as hh:mm (hours:minutes). Use **date.Hour.ToString**, etc. to format and return a string.

**ToString():** Every object we create in C#, automatically becomes a child of a super object in the .NET Framework that is called **System.Object**. All objects then have access to this object's public members. It has only a few public methods, one of which is the **ToString()** method. This method can be given a new body (implementation) which in object-orientation is known as **overriding** (not the same as overloading).

The idea behind this method is to return a string with some typical information about the object. If you don't provide a new implementation for this method, it returns the memory address of the object which is often of no interest. Therefore, it is always a good idea to let this method give out a formatted string representation of the values of the instances variables of the object.

```
public override string ToString()
{
     string textOut = String.Format("{0, -20} {1, 10} {2, -16}   {3, -20}",
                                    date.ToLongDateString(),
                                    GetTimeString(),
                                    GetPriorityString(),
                                    Description );

    return textOut;
}
```

An alternative way, giving the same output as from above:

```
public override string ToString()
{
    string textOut =  $"{date.ToLongDateString(),-20} {GetTimeString(),10} " +
                      $"{GetPriorityString(),-16} {description,-20}";
    return textOut;
}
```

**Note**: The + sign is used to divide the code into two lines to improve readability.

## 4. The **MainForm class (MainForm.cs)**

The checklist explained earlier may apply for a form class as well. The difference is that a Form class has a visual part and a code part and also that the Form object (**MainForm**) in our application is not used by other objects. This means that we do not need to have any properties in the **MainForm** class, but it has fields, a constructor, and other methods. Another difference is in that a Form object usually contains some special methods that are called event-handlers or event-handler methods, more details in a moment.

4.1 GUI: Draw your GUI as suggested in the assignment description. Assign proper names to your controls. Some of controls do not have a visual function (as other controls such as TextBox, Labels and Buttons do). These controls instead offer functionalities and features that you can use. When you add them to your Form; they are placed by VS at the bottom of the design window, as visualized in the figure here.

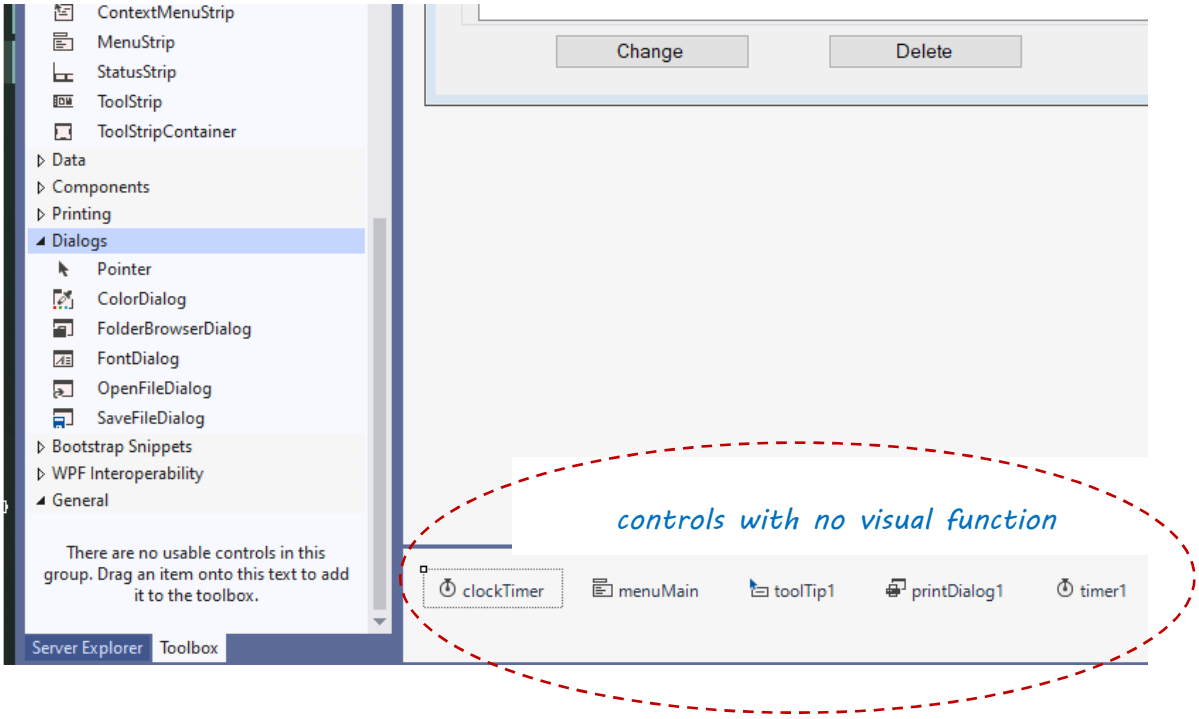



Figure 3: Controls with properties and methods but no visual part

You can view and change their properties by clicking on them and then using the Properties window. The timer, menu strip, tooltip, dialog boxes and several other controls that you find on the Toolbox belong to this category.

In the image, **clockTimer** is a Timer, **menuMain** is the main menu strip and **toolTip1** is a **ToolTip** control.

The class diagram in Figure 5 shows the fields and methods that are used to start, run and close the application. Now back to our checklist for the MainForm:

**Fields:** **taskManager** as an object of TaskManager
**fileName** a string to store the path and the name of file for reading and writing data.

**MainForm** needs an object of the **TaskManager** so it can store and manipulate tasks when receiving data from the user and displaying data for the user. This class has all the features needed to add a new **Task**-object, delete and change the data of tasks. The class also provides an array of strings where each string represents a task in the task list.

**Properties: MainForm** is not meant to be used by any other class and therefore no access to private members is required.

**Methods**: The class diagram given in the previous page shows the methods needed and you are certainly familiar with such methods from your earlier assignments. In addition, all methods that you write can be private, as they are not going to be called by objects of any other class.

**Constructor**: VS creates a constructor with a call to **InitializeComponent** method that VS has created and will maintain it by itself. All generated code by Visual Studio that have to do with Form and its components are saved in the **MainForm.Designer.cs**. Do not do any changes in this file, unless you get a compiler error (for example when you delete an event-handler). You can do your initialization after this statement (**InitializeComponent**) as shown in Figure 4.

```
public partial class MainForm : Form
{
    private TaskManager taskManager;

   //Working with only one file located in the same directory as the
   //application's EXE- file
   private string fileName = Application.StartupPath + "\\Tasks.txt";

   //The default constructor generated by VS
   public MainForm()
   {
       //VS's initialization
       InitializeComponent();

       //From here on - your code
       InitializeGUI();

   }
```

Figure 4: MainForm – fields and constructor

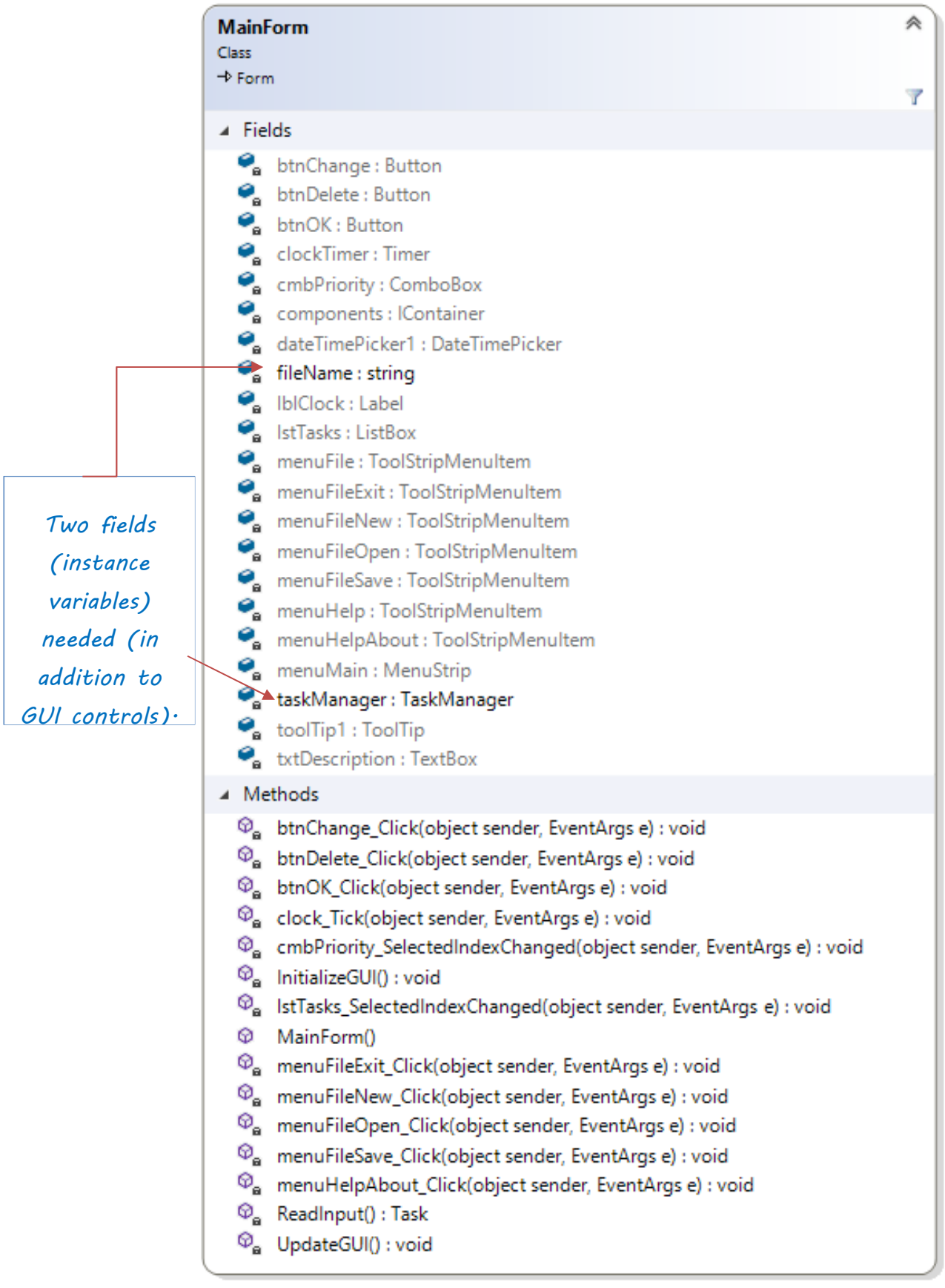


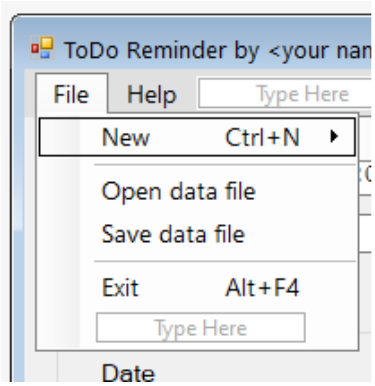

Figure 5: MainForm - Class diagram

The method **InializeGUI** is given below to let you learn and understand the new controls used in this assignment. Do not just copy (re-write) the code; understand every single statement, and if you have questions, go to C# Documentation online (or press F1 on the component) and acquire facts. You can of course ask your question in the forum on the module.

```
//Do all initialization here.  The method can be called
//again to reset the program.
private void InitializeGUI()
{
    this.Text = "ToDo Reminder by <your name>";

    taskManager = new TaskManager ( );

    cmbPriority.Items.Clear();
    cmbPriority.Items.AddRange(Enum.GetNames(typeof(PriorityType)));
    cmbPriority.SelectedIndex = (int)PriorityType.Normal;

    lstTasks.Items.Clear();
    lblClock.Text = string.Empty;
    clockTimer.Start();

    txtDescription.Text = string.Empty;

    dateTimePicker1.Format = DateTimePickerFormat.Custom;
    dateTimePicker1.CustomFormat = "yyyy-MM-dd   HH:mm";

    toolTip1.ShowAlways = true;
    //complete with more initialization

    toolTip1.SetToolTip(dateTimePicker1, "Click to open calender for date, writ in time here.");
    toolTip1.SetToolTip(cmbPriority, "Select the priority");

    //string textOut = string.Format("You have {0} tasks.", lstReservations.Items.Count);
    string tips = "TO CHANGE: Select an item first!" + Environment.NewLine;
    tips += "Make you changes in the input controls," + Environment.NewLine;
    tips += "Click the Change button." + Environment.NewLine;

    toolTip1.SetToolTip(lstTasks, tips);
    toolTip1.SetToolTip ( btnChange, tips );

    string delTips = "Select an item first and then click this button!";
    toolTip1.SetToolTip ( btnDelete, delTips );

    string desTips = "Write your sins here!";
    toolTip1.SetToolTip ( txtDescription, desTips );

    menuFileOpen.Enabled = true;
    menuFileSave.Enabled = true;
}
```

Some of the new controls are described in more details in the following sections

### 4.2 File - New




When the user clicks the submenu item **File-New**, the program should be put into the start mode. All you have to do here is to call the method **InitializeGUI.**

### 4.3 File – Open data file and Save data file

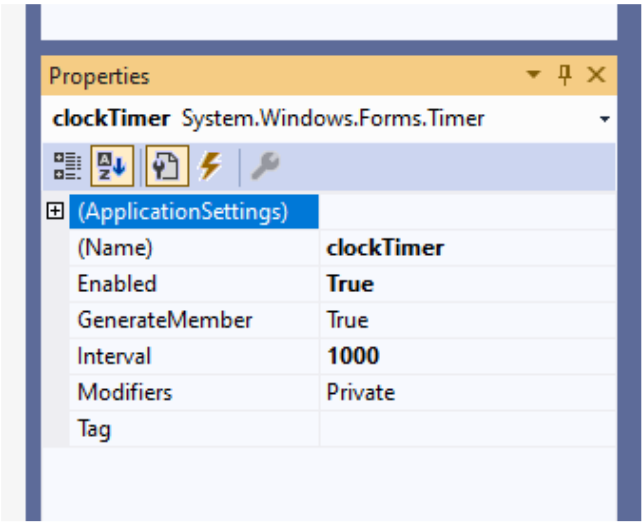
These submenu items are explained in details in a separate help file covering writing and reading data to and from text files.

### 4.4 Menu_Exit

Click on this menu item should close the form, but to make your application more user-friendly, you should let the user confirm or the chance to cancel exiting. Here is how you can program this feature:

```
private void menuFileExit_Click(object sender, EventArgs e)
{
    DialogResult dlgResult = MessageBox.Show("Sure to exit program?",
                  "Think twice", MessageBoxButtons.OKCancel);
    if (dlgResult == DialogResult.OK)
        Application.Exit();

}
```

### 4.5 Timer:




Drag a **Timer** control from the **Toolbox** ub VS, and drop it on the form. Then, select **Properties** from the popup-menu that will show up.

Change the values as in the figure.

Now double-click on the **Timer** control and VS will create the event-handler method for the timer. Complete the method as follows:

```
private void clock_Tick(object sender, EventArgs e)
{
    lblClock.Text = DateTime.Now.ToLongTimeString();
}
```

### 4.6 **Add New Task**

Get user input from controls on **MainForm**, store the values in a new **Task** object (a local variable is suitable). If any input is invalid, a message should be shown to the user. If you want to check a time given by the user within a valid **DateTime** range, this can be done using **dateTimePicker**'s **MinDate** and **MaxDate**. You don't need to do that for the purpose of this assignment....

```
private void btnOK_Click(object sender, EventArgs e)
{
    Task task = ReadInput();
    if (taskManager.AddNewTask ( task ))
    {
        UpdateGUI ( );
    }
}
```

Write the read input method to read and save description, date and time and the priority of the task being created. The date and time is simple to fetch from the dateTimePicker.

```
task.TaskDate = dateTimePicker1.Value;
```

```
private void UpdateGUI()
{
    lstTasks.Items.Clear ( );
    string[] infoStrings = taskManager.GetInfoStringsList ( );
    if (infoStrings != null)
        lstTasks.Items.AddRange ( infoStrings );
}
```

To convert the selected priority from the ComboBox to its corresponding PriorityType, you can use the following construction:

```
task.Priority = (PriorityType)cmbPriority.SelectedIndex;
```

## 5. The TaskManager class

Create a dynamic list to store Task object and then write methods to add, change, delete a task object. Do not write any Properties to give other objects access to the list as the list is an object and returning an object returns the address of the object. Changes in the caller method will also affect the original list.

```
class TaskManager
{
    //Declare a List<T> object where T is a Task type
    List<Task> taskList;

    //Create the taskList in the constructor
    public TaskManager()
    {
        taskList = new List<Task>();
    }
```

A method to add a new task. The new task is (must be) created in the caller method (**MainForm**).

```
/// <summary>
/// Add a new object at the end of the taskList
/// </summary>
/// <param name="newTask">The task object that is to be added.</param>
/// <returns>true if the task is added successfully, false otherwise.</returns>
public bool AddNewTask(Task newTask)
{
    bool ok = true;

    if (newTask != null)
        taskList.Add(newTask);
    else
        ok = false;

    return ok;
}
```

To prepare and return a list of strings representing the tasks currently in the task list:

```
public string[] GetInfoStringsList()
{
    //create a local array of string element with a capcity = num
    //of elements in the taskList
    string[] infoStrings = new string[taskList.Count];

    for (int i = 0; i < infoStrings.Length; i++)
    {
        infoStrings[i] = taskList[i].ToString();
    }
    return infoStrings;
}
```

Good Luck.

*Farid Naisan*
Course Responsible and teacher