



Programming Using C#, Basic Course

Assignment 5

OOP - Encapsulation

Customer Registry

Mandatory

[Farid Naisan](#)
University Lecturer
Department of Computer Sciences
Malmö University, Sweden



Contents

Assignment 5: Customer Registry	3
1. Objectives	3
2. Description	3
3. The Project	4
4. Requirements	5
5. The Address Class (Address.cs)	7
6. The Contact class (Contact.cs)	8
7. The Countries Enum (Countries.cs)	9
8. The Customer class (Customer.cs)	9
9. The Email and Phone classes	9
10. The CustomManager class (CustomManager.cs)	10
11. The GUI Class (MainForm.cs)	10
12. The CustomerForm Class (CustomerForm.cs)	12
13. Guidance and hints	14

Assignment 5: Customer Registry

1. Objectives

The main objectives of this assignment are to take a real first step into the world of the object-oriented programming (OOP) by learning to work with one of the essential aspects of OOP, namely encapsulation. We will also practice data-hiding to complete the process of encapsulation. This assignment covers the following concepts:

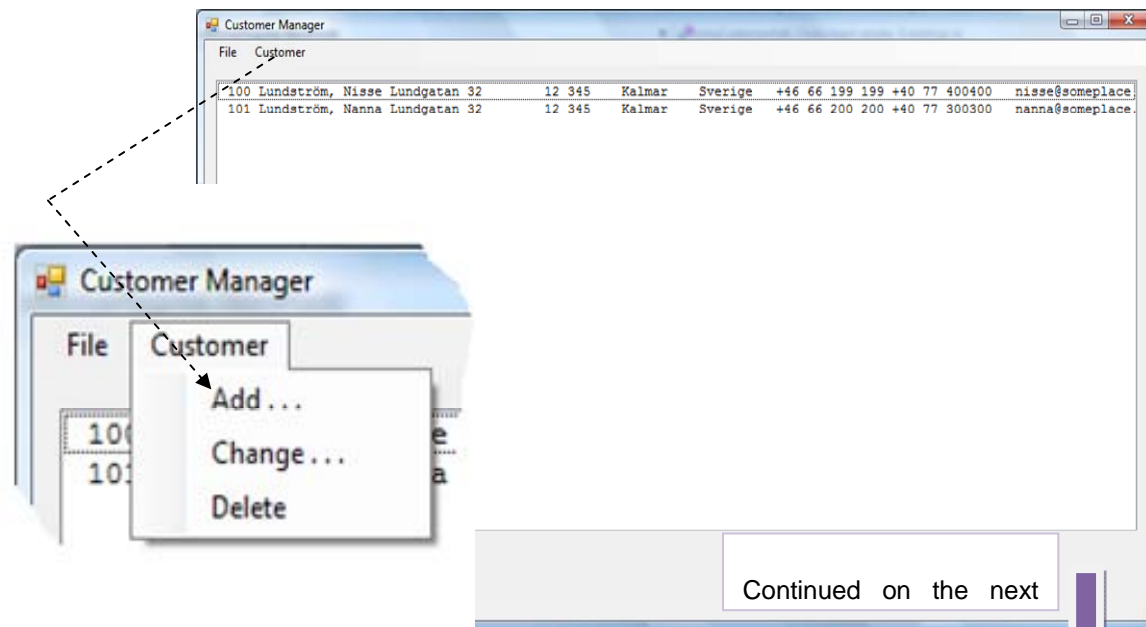
- Encapsulation
- Constructors
- “**Has a**” relation between objects
- Properties
- Array of objects
- Menus

This assignment is intended to be used as a basis for the next assignment. So make sure to think ahead and backup your solution for reuse in the future.

2. Description

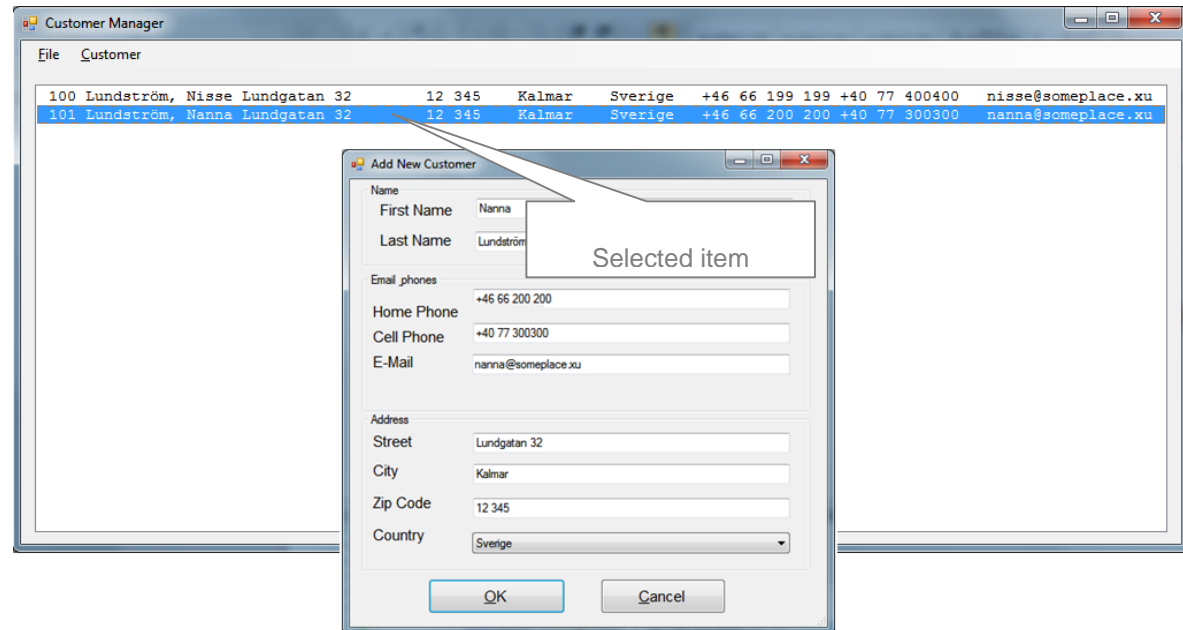
Companies as well as organizations usually maintain information about their members. Members can be customers, students, or other type of individuals, which can collectively be considered as customers.

Your job in this assignment is to develop a program that maintains a general register of customers. The information to be stored in the register includes, name, address, phones and emails. The user of the program should be able to add new customers, change entries for a registered customer or delete a customer from the register. A sample run session is shown below.



When the user clicks the OK button, after filling the form fields, the changes are made to the register and its list is updated, but if the user chooses to cancel the operation, the information entered is discarded.

While you can make use of an ordinary fixed-size array or a dynamic array of objects to serve as the registry for storing customer objects, .NET provides a very handy object, a collection, called **ArrayList** for this purpose. An **ArrayList** is a collection that dynamically can increase or shrink in size. Thus, you don't have to know the size of the array at compile time - it may increase and decrease in size as you run the program. In addition, working with an ArrayList is quite easy. (Those who are already acquainted with use of ArrayLists are recommended to try the generic type, List <T> instead).



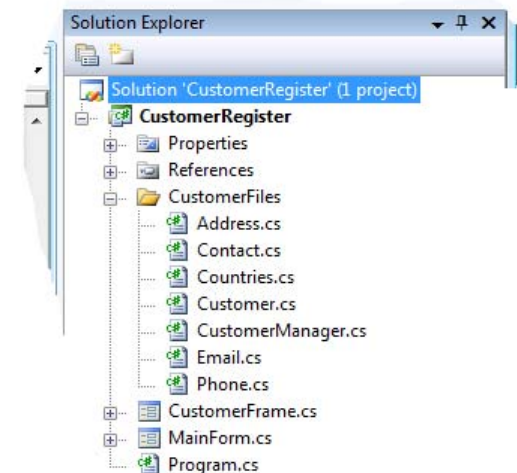
3. The Project

The figure to the right shows the classes that form the application. You may of course choose other names for your classes or add new classes. Note that all files pertaining to customer handling is organized in a separate folder in the project.

Make sure that you have a clear understanding of the concepts of constructors and properties, before starting this assignment. Read your textbook and use the material on It's learning in case you need to review the theories.

The class diagram shows "has a" associations between the classes by drawing open head arrows (usually arrows with open triangle) pointing from the container object to the member object.

A detailed description of the classes as well a simple class diagram, showing the associations between the object of classes shown in the project (figure at the right) are presented in this document to help you accomplish a recommended solution.



4. Requirements

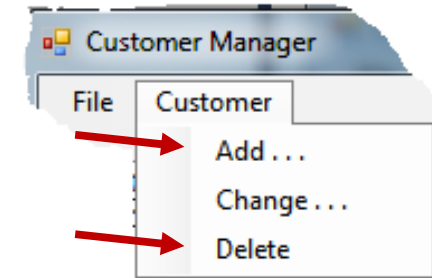
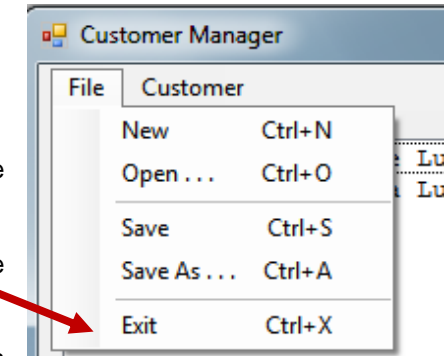
Requirement for a passing grade:

To train with menus, you are to design the menus File and Customer on your GUI, but only a few of the menu items are to function as described below.

- 4.1 Draw the File menu with the New, Open, Save, Save As Exit submenus; these do not have to be functional except the Exit menu.
 - 4.1.1 When the user clicks the Exit menu, a message box with “OK/Cancel” or “Yes/No” buttons should be displayed to the user to confirm exiting the application. The application should not end if the user selects Cancel or No.
- 4.2 The Add sub-menu of the Customer menu must work well. Check the user input by requiring some basic information, as, first name, last name and a phone number. Do not require that all the entries should be given (so much information is not always available).
- 4.3 The Delete sub-menu of the Customer menu must work. However, you can force the user to select the customer which is to be deleted from the registry in the listbox . When no selection is made (i.e. when SelectedIndex = -1), no action needs to be taken.
- 4.4 The CustomerForm object should not directly do any action on the customer registry (customerManager) of the MainForm.. In other words, the MainForm should not pass the whole registry object (customerManager) to the CustomerForm object. The MainForm should not pass a reference to itself either. The communication between the MainForm can take place at least in two ways:
 - 4.4.1 Declare a field in the CustomerForm that is of the type Customer. Write then a set and get method. The customer object can transport information for a customer that is added, or is to be changed using the get and set properties.
 - 4.4.2 A customer object (and other information) can be passed by reference to the CustomerForm as argument in the CustomerForm's constructor.

However, the constructor can be used only once when the object is created. The get and set properties are needed anyway.

- 4.5 Constructors in the same class must chain call each other (See the class Address, later in this document).



Requirement for a higher grades:

For higher grades, A and B (ECT) or VG (Swe), you should also do the following, in addition to the above requirements:

- 4.6 The **Change** menuitem should also work. You may require the user to first select a row in the ListBox. Your program then should pass the information for the selected customer to the **CustomerForm** object. The **CustomerForm** object should then display the information in the related controls.
- 4.7 The user makes changes and then if the user chooses OK, the changes should be saved in the registry.
- 4.8 Optional: if the user has not select a row in the listbox but still wants to delete or change information about a customer, your program should then search for a user with the help of the first and last name given in the customer Form object or using any other search criteria..

A simple Class Diagram

MainForm creates an instance of **CustomerManager** as a private field (instance variable).

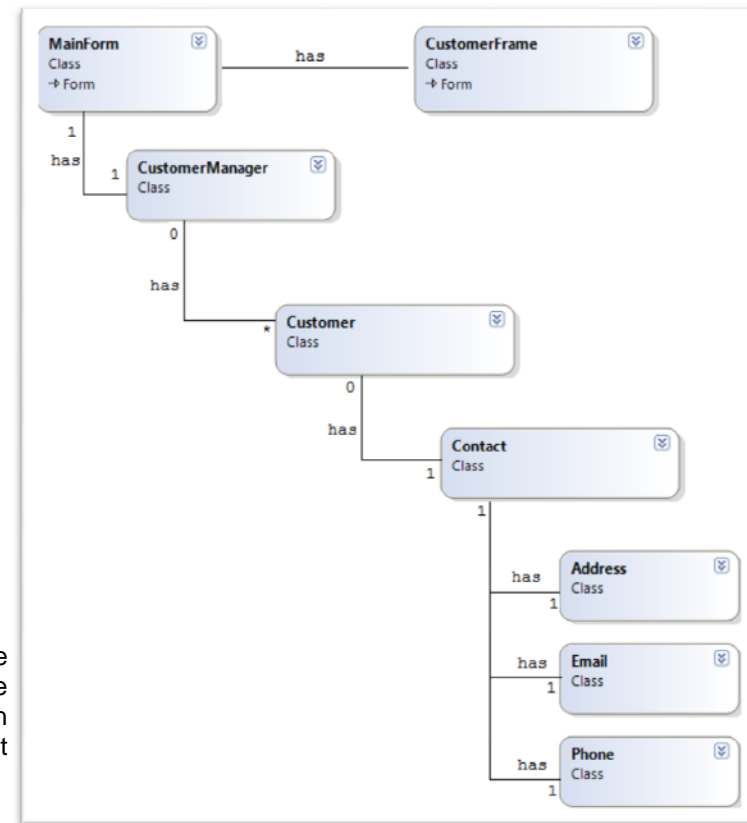
CustomerManager has an array of **Customer** objects (ArrayList, or List<T>), as private field

Customer has an instance of **Contact**, as a private field.

Contact has an instance of **Address**, **Email** and **Phone** as its fields.

MainForm uses a local instance of **CustomerForm** (local variable inside the event-handler methods)

CustomerForm has **Customer** as instance variable.



The instructions below are provided to help you program the contents of the classes shown in the class diagram. Remember that you don't have to follow the instructions given in this document step by step. You can implement your own solution, provided that you consider the above requirements and implement encapsulation well. Let's begin with the code files first.

5. The Address Class (Address.cs)

The class diagram (done in VS) shows the fields and the methods.

5.1 **Fields:** As in the Address class diagram. The field **m_strErrorMessage** is meant to contain eventual errors. It should be emptied after the message is used.

5.2 **Constructors:**

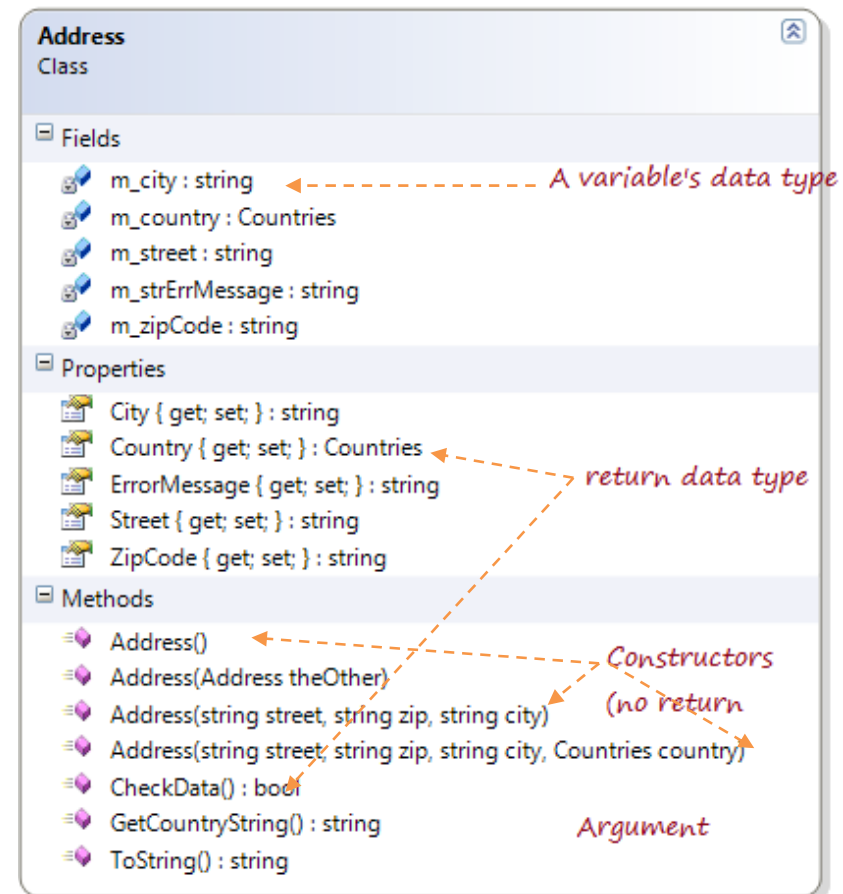
5.2.1 Write **three** constructors as follows:

5.2.2 One default constructor in which you assign all instance variables of the type **string** to **string.Empty**. This is to avoid **null**-problem with strings.

5.2.3 One constructor with 4 parameters:

```
public Address(string street,
               string zip,
               string city,
               Countries country)
```

5.2.4 A constructor with 3 parameters, eliminating country from the above.



Chain calling of constructor: Where you write more than one constructor in a class do not copy and paste same code! Put your initialization and other code in the constructor that has the most number of parameters. Call constructors in a chain, with the one that has less number of parameters calling the one that more parameters 3 parameters. In the Address class here, the default constructor should call the constructor with three parameters, passing default values for street, zip and city and the constructor that has 3 parameters should call the constructor with 4 parameters, passing a default value for the country (Sverige).

5.3 **Properties:**

5.3.1 Write **get** and **set** properties for every field of the class.

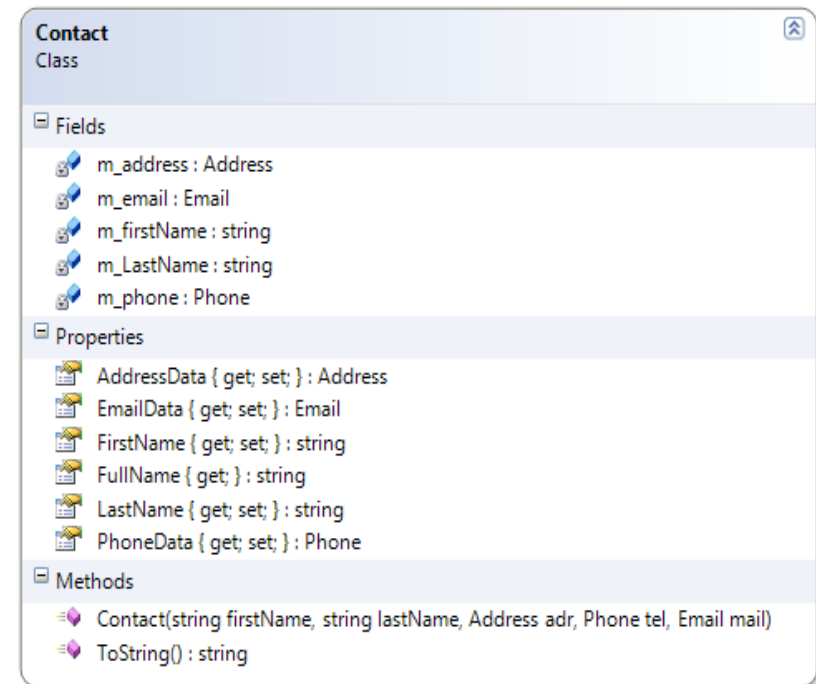
5.4 Methods:

- 5.4.1 A **CheckData** method performing some type of a control for the validity of the address information given by the user. The program may require that (for example) the input boxes intended for street and city are not empty. The method should return `true` if the control is successful and `false` otherwise. The field `strErrorMessage` should contain the error message when the control fails.
- 5.4.2 Override the **ToString** method so that it returns a formatted string with all information about the address (street, zip code, city and country). This string may then be used by the GUI class.
- 5.4.3 Write a method that returns a string containing the value of the Country. The Enum, `Countries`, containing a complete list of all countries of the world, follows this assignment. Some of the country names contain the character `'_'`. Replace these by a blank space when presenting them on the GUI. Example:

`United_States_of_America` should be converted to “United States of America” when showing to the user.

6. The Contact class (Contact.cs)

- 6.1 **Fields:** as in the class diagram.
- 6.2 **Constructors:** a constructor with parameters for first name, last name, address, phone, and email.
- 6.3 **Properties:**
- 6.3.1 `get` and `set` properties for all fields.
- 6.3.2 A `get` property (read-only), returning a string containing the first name and the last name in a format of your choice, ex “Lundström, Nisse”, or “Nisse Lundström”.
- 6.4 **Methods:**
- 6.4.1 Override the **ToString** method to return a string formatted with full name, address, phone and email, by calling the property **FullName**, and the **ToString** methods of fields `m_address`, `m_phone` and `m_email`.

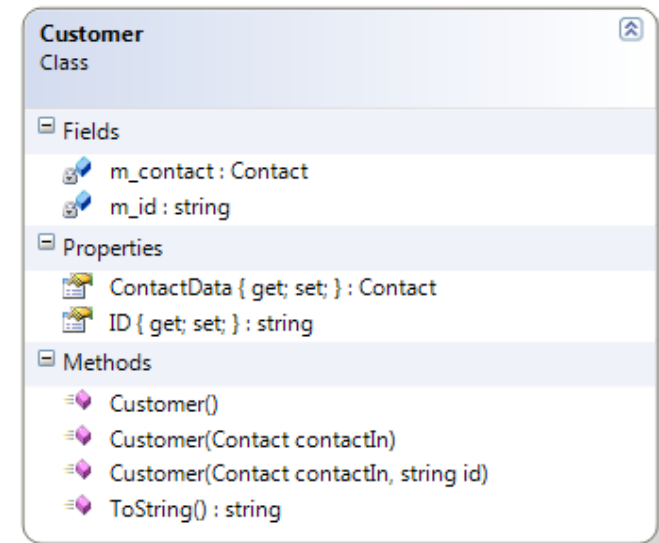


7. The Countries Enum (Countries.cs)

7.1 This file accompanies the assignment and is available for downloading.

8. The Customer class (Customer.cs)

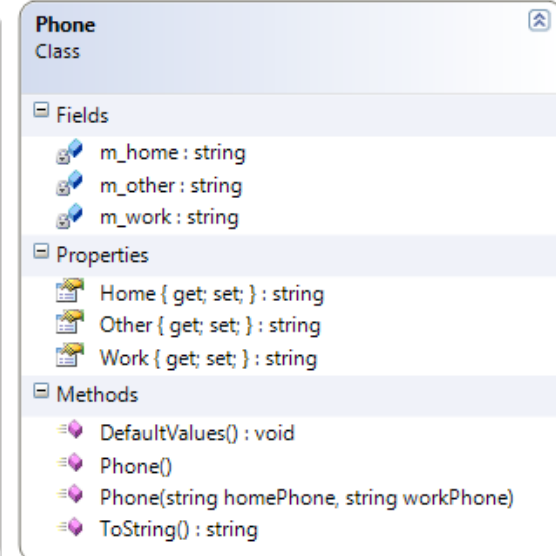
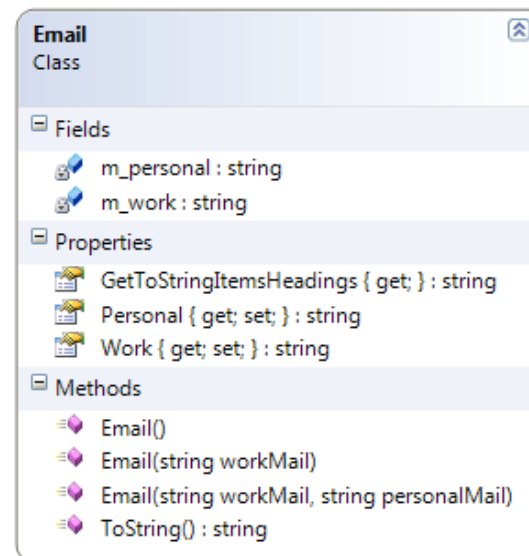
- 8.1 **Fields:** this class should use an object of Contact to store contact information (name of the contact person, address, etc). It should also maintain a ID determined by the **CustomerManager** object (see page 10), as shown in the class diagram
- 8.2 **Constructors,** Write at least two constructors, one default and one with a contact object and ID as parameters.
- 8.3 **Properties:** **set, get** for each of the fields.
- 8.4 **Methods:** Override the **ToString** method, making use of the **m_contact** object's **ToString** method.



9. The Email and Phone classes

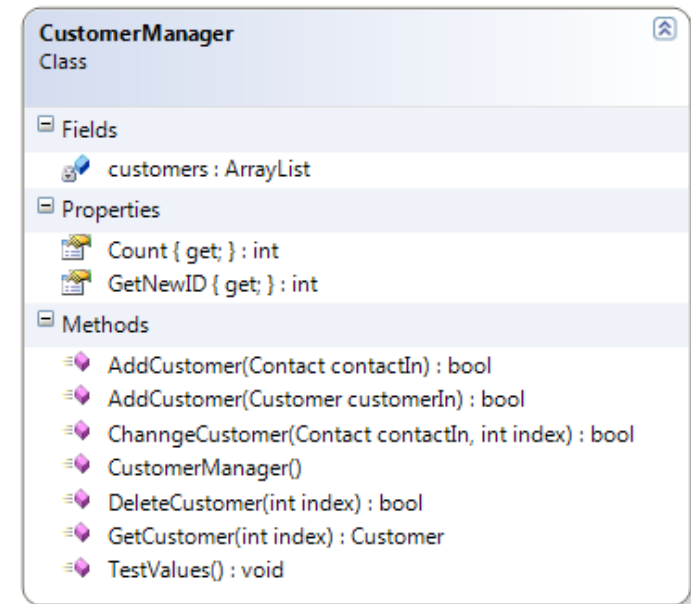
Follow the same pattern and do at least the following:

- 9.1 **Fields:** as shown in the class diagram.
- 9.2 **Constructors,** Write at least two constructors, one default and one with one or more parameters. Make your own decision.
- 9.3 **Properties:** **set and get** for each of the fields.
- 9.3.1 **Methods:** Override the **ToString** method to return information stored in the current object.



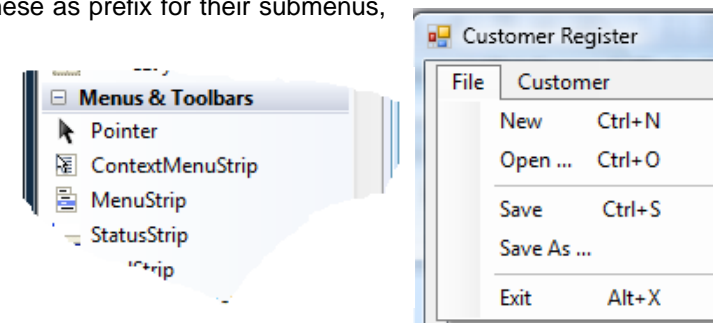
10. The CustomManager class (CustomManager.cs)

- 10.1 **Fields:** Declare an ArrayList (or a List) for storing objects of the type Customer.
- 10.2 **Constructors:** Write a default constructor and create the ArrayList here.
- 10.3 **Properties:** A read-only [get](#) property returning the number of registered items in the ArrayList (Use ArrayList's Count property).
- 10.4 **Methods:** Write methods for adding, changing and deleting customer objects in the ArrayList. Although the class diagram shows two overloaded AddCustomer methods, it is sufficient with one defining a Customer object as parameter.
- 10.5 When adding a new Customer, provide also an ID which may be a number starting for example at 100.



11. The GUI Class (MainForm.cs)

- 11.1 Draw your GUI as in the example given earlier. If you are already familiar with Listboxes, you may certainly try a ListView instead.
- 11.2 Design your menus using the **MenuStrip** component on the toolbox in Visual Studio.
- 11.3 The Listbox must be updated when the user chooses OK in the **CustomerForm** window.
- 11.4 For the Delete submenu, no data other than the row number in the Listbox is required.
- 11.5 Begin menu names as **mnuFile**, **mnuCustomer** for top-level menus and use these as prefix for their submenus, **mnuFileOpen**, **mnuCustomerAdd**, etc.
- 11.6 Double-click on each submenu item in the Visual Studio to write your code for the click-event of the item. It works much like a button click event.
- 11.7 The File menu needs only to be designed on the GUI. The menu items, except for the Exit, are not required to function.
- 11.8 To draw a separator, write a "-" (minus sign) as the submenu's text property.





- 11.9 Assign also shortcut keys to **File** sub-menus, using the **ShortcutKey** property of the menu item.
- 11.10 You may practice your own way of coding the GUI class, but make sure that you do not write long methods. Use a good programming style for the design and structure of your code. The figure below is provided to help you manage the coding in the **MainForm** class. Comments have been deleted to save space (but don't take this as an argument or excuse to save space in your code project!).
- 11.11 **Fields:** In addition to GUI components, declare and create an instance of the CustomerManager class.
- 11.12 **Methods:** Write a method, **UpdateCustomerList**, so:
- it clears the ListBox,
 - gets information for each registered customer from the **customerMngr**, format it into a string, and add it to the ListBox.
- 11.13 The Class Diagram at the right show the fields and the method of the MainForm class. As you see, the class contains both my declaration of the customerMngr and those which are generated by Visual Studio as I have put my controls and menus on the form. The methods include event-handler methods and also my methods (UpdateCustomerList).
- 11.14 Write code in the event-handler methods connected to click events of the sub-menus to perform tasks, such as File-Exit, Customer-Add, etc. Double-click on a menu item (for ex File-Exit) in Visual Studio, and VS will prepare the handler method for you. Below is an example of the handler method for the menu item Customer – Add.
- 11.15 To display the CustomerForm from MainForm, you need to create an instance of the CustomerForm. You can then call the instance's ShowDialog method to display the form. As an example, check the following code in which an instance of CustomerForm, created and initiated with values for a selected customer from the MainForm. The values can then be displayed in the related input controls on the CustomerForm. This is very practical if you will add members having same address or other common data.

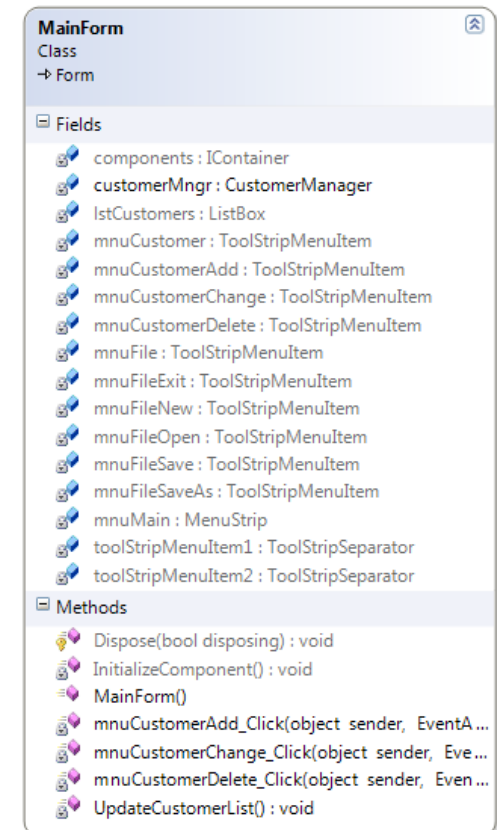
```
public partial class MainForm : Form
{
    CustomerManager customerMngr = new CustomerManager();

    public MainForm()
    {
        InitializeComponent();
        //Start with test values(comment out in the release version)
        customerMngr.TestValues();

        //Do other initialization
        //InitializeGUI() //for later use

        UpdateCustomerList(); //fill the listbox
    }
}
```

Constructor - used to do initializations



```
private void mnuCustomerAdd_Click(object sender, EventArgs e)
{
    //Create an instance of customer frame (constructor with title as param)
    CustomerForm frmCustomer = new CustomerForm("Add New Customer");

    int index = lstCustomers.SelectedIndex; //selected customer in the listbox

    //If a customer is selected, export data for the selected customer to CustomerFrame
    if (index != -1)
        frmCustomer.CustomerData = new Customer(customerMgr.GetCustomer(index).ContactData);

    if (frmCustomer.ShowDialog() == DialogResult.OK) //Show the CustomerFrame object
    {
        //The user has chosen the OK button - add the new customer object
        customerMgr.AddCustomer(frmCustomer.CustomerData);
        UpdateCustomerList(); //update results
    }
}
```

If ListBox not empty, or a row is selected, fetch values for the selected customer. Here the frmCustomer's set-property, CustomerData (see next page) is called

An object of the CustomerForm, frmCustomer, is initiated with a string (title). · · CustomerForm has a constructor with a string parameter

12. The CustomerForm Class (CustomerForm.cs)

12.1 You may practice your own solution even in this part, but the idea here is to teach you that:

- the constructors of a Form class can be overloaded, as for any other class,
- `get` and `set` properties can be used to communicate with the caller object.

12.2 The method ReadInput calls other Read-methods. Some code excerpts are given below as a little help:

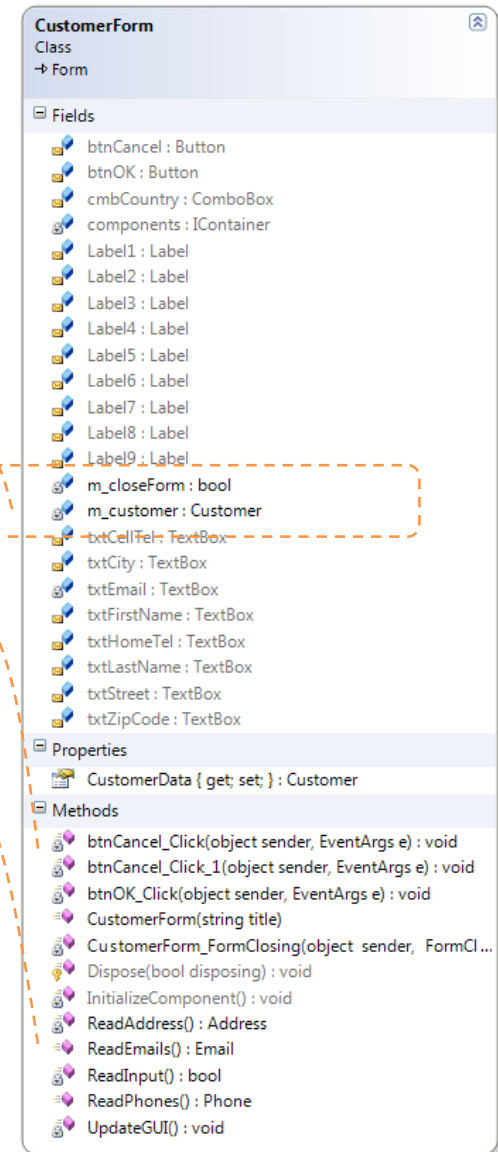
```
public partial class CustomerForm : Form
{
    //Customer object receiving input and/or sending output
    private Customer m_customer;
    //Flag to handle the closing of the form
    private bool m_closeForm;

    //Constructor with one parameter (title of the form)
    public CustomerForm(string title)
    {
        InitializeComponent();
        //Other initialization.
        this.Text = title;
        m_closeForm = true;
    }

    public Customer CustomerData
    {
        get { return m_customer; }
        set
        {
            if (value != null)
                m_customer = value;
            //update input controls
            UpdateGUI();
        }
    }
}
```

- 12.3 Closing the form created and opened by the MainForm appropriately needs a little work. Handle the FormClosing event of the CustomerForm and copy the following code:

```
private void CustomerForm_FormClosing(object sender, FormClosingEventArgs e)
{
    if (m_closeForm)
        e.Cancel = false; //Close the Customer Form
    else
        e.Cancel = true; //Do not close (user has chosen Cancel)
}
```



CustomerForm
Class
→ Form

Fields

- btnCancel : Button
- btnOK : Button
- cmbCountry : ComboBox
- components : IContainer
- Label1 : Label
- Label2 : Label
- Label3 : Label
- Label4 : Label
- Label5 : Label
- Label6 : Label
- Label7 : Label
- Label8 : Label
- Label9 : Label
- m_closeForm : bool
- m_customer : Customer
- txtCellTel : TextBox
- txtCity : TextBox
- txtEmail : TextBox
- txtFirstName : TextBox
- txtHomeTel : TextBox
- txtLastName : TextBox
- txtStreet : TextBox
- txtZipCode : TextBox

Properties

- CustomerData { get; set; } : Customer

Methods

- btnCancel_Click(object sender, EventArgs e) : void
- btnCancel_Click_1(object sender, EventArgs e) : void
- btnOK_Click(object sender, EventArgs e) : void
- CustomerForm(string title)
- CustomerForm_FormClosing(object sender, FormClosingEventArgs e)
- Dispose(bool disposing) : void
- InitializeComponent() : void
- ReadAddress() : Address
- ReadEmails() : Email
- ReadInput() : bool
- ReadPhones() : Phone
- UpdateGUI() : void



13. Guidance and hints

- 13.1. Fields that are of object type, e.g. `m_Address` in the `Address` class, must be created with `new` either in its container class (constructor is a good place) or in the caller method. If you forget this, the program will crash during the runtime.
- 13.2. Remember that `string` variables are objects and they are initiated to `null` by the compiler. This will give you a real head-ache if a string variable is not assigned any value and you are using it. To remedy this problem, make this as your habit to either always check so a string variable is not null (`if strMyText != null`), or simply set all fields that are of the type `string` to `string.Empty`. This assigns the variable a value that is an empty string (but is not equal to `null`). If you wish to check so a string variable is neither null nor empty, you can use the following example:

```
if (string.IsNullOrEmpty(strMyTextStrign))
```

- 13.3. A constructor in a class can call another constructor in the same class by special syntax, shown below:

```
public Address(string street, string zip, string city):  
    this(street, zip, city, Countries.Sverige)  
{  
  
}
```

- 13.4. An `ArrayList` is declared and created as follows::

```
private ArrayList customers;  
  
arrayObj = new ArrayList(); //place this line in the constructor
```

- 13.5. To add an element into the array

```
arrayObj.Add(new objName(...))
```

Don't forget that you have to first create the object `objName` with `new` either in a separate statement, or as in the above example.

- 13.6. `arrayObj.Remove` deletes an element in the `ArrayList arrayObj`, and `arrayObj.Count` gets the number of elements in the array.

That's about all you need to know to begin using an `ArrayList` in most of the cases. There are many other usable methods (like `sort`) as well. Read the documentation about the `ArrayList` in the MSDN to find out.

As mentioned earlier, the collection List < > is a better choice than ArrayList, and it easier to use (you don't need to use typing to get an object from the list).

- 13.7. Other classes like **FormMain** would of course need to access data stored in the ArrayList. Your ArrayList is private and you need to somehow deliver data from the array to other objects. What would you do? Would you write a Property that makes the array reference accessible as below?

```
return customerList 'Very bad programming
```

Never do that! Another way is to write a method that returns the reference to an element at a position in the array:

```
return (Customer)customerList(index); 'Better
```

This method is better but still not very safe because of the fact that we are passing the address of our object in the memory. However, if the object has set and get methods (as in our case), that encapsulates and protects the instance variables of the objects stored in the ArrayList, there is a degree of safety. A better and safer way is to return a copy of an object, but we skip this discussion for the time being.

Furthermore, the ArrayList needs to know about the type of the object it has stored in a particular position as it is possible to store different types of objects in the same ArrayList, and that is why the above statement includes a typecasting to avoid compiler error.

- 13.8. **Properties**, set and get that are related to fields that are of an object type, e.g. m_email in the Contact class, can send or assign a reference to the variable, as an simplification in view of the preceding discussion.

```
public Email EmailData
{
    get { return m_email; }
    set { m_email = value; }
}
```

- 13.9. A **copy constructor** makes life a lot easier when you would like to initiate an object with values of another object of the same class. Although you don't need to do that to solve assignment, but it is good to make it a habit to write a copy constructor to every new class. The code on the next page shows how a copy constructor may look like.



```
//Copy constructor
//Don't need to copy all members
public Address(Address theOther)
{
    m_street = theOther.m_street;
    m_zipCode = theOther.m_zipCode;
    m_city = theOther.m_city;
    m_country = theOther.m_country;
}
```

The above constructor can be called as follows:

```
Address address2 = new Address (address1);
```

13.10. Last but not least, write “**prop**” inside a class file in the Visual Studio and then press the “tab” key on your keyboard twice, and you’ll see a magic. Try it!

This is not a difficult assignment, but hopefully a good learning opportunity. Nevertheless, even small things are difficult if you don’t know how to handle them. So, if you have questions, ask in the Forum on ITS and we will provide more help.

Good Luck!

Programming is fun. Never give up. Ask for help!

Farid Naisan

Course Responsible and Instructor