



**MALMÖ HÖGSKOLA**  
Teknik och samhälle

**Malmö University**  
School of Technology

Programming Using C#

**OOP and Encapsulation**

**Assignment 5 – Alt 1**

**Contact Registry**

**Mandatory only for Grade C or G**

**(If you aim at grades A, B or VG, do not do this assignment)**

[Farid Naisan](#)

University Lecturer  
Department of Computer Science

## Contact Registry

### 1. Objectives

The main objectives of this assignment are to take a 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:

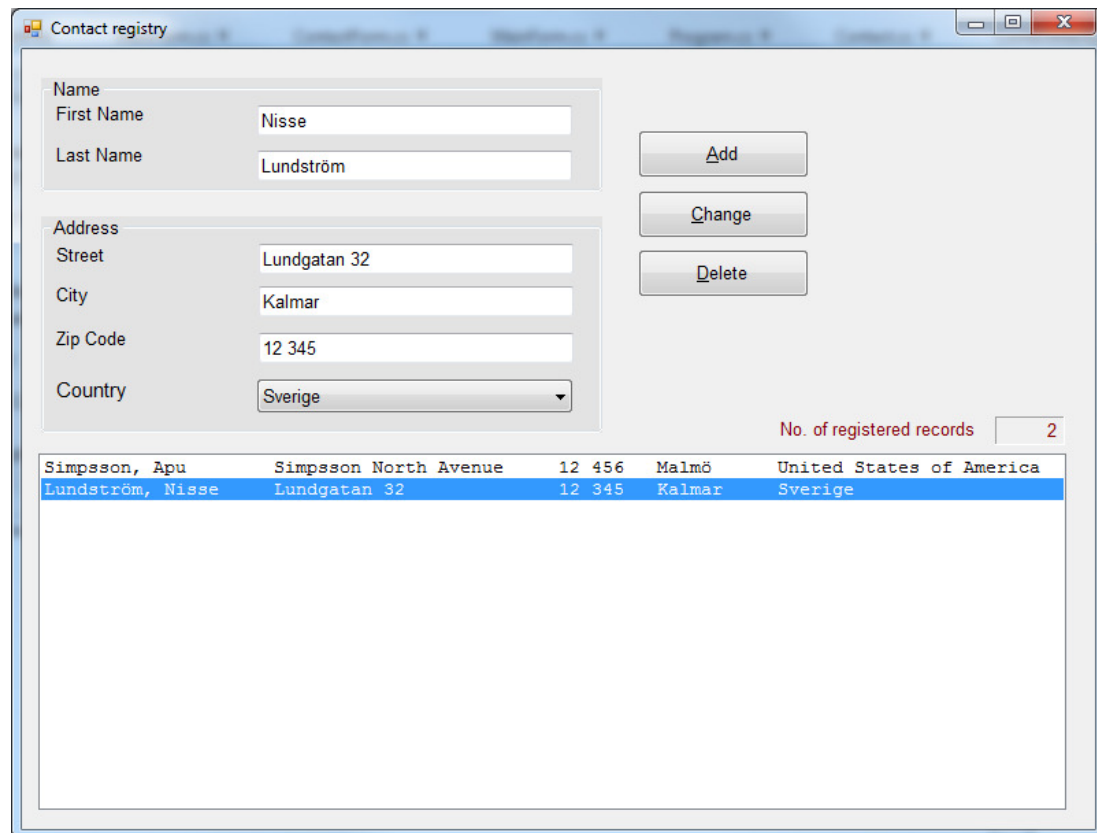
- Encapsulation
- Constructors
- “**Has a**” relation between objects
- Properties
- Collection of objects

### 2. Description

Write a Windows Form Application that saves a list of contacts. Every contact is a person that has the following data:

First name  
Last name  
Address including:  
    Street address  
    Zip code  
    City  
    Country

The GUI should allow the user to add a new contact, change or delete an existing one. It should also present a list of all contacts saved in the registry. The list is to be updated after every change in the registry.



Name  
 First Name: Nisse  
 Last Name: Lundström  
 Address  
 Street: Lundgatan 32  
 City: Kalmar  
 Zip Code: 12 345  
 Country: Sverige

Add  
 Change  
 Delete

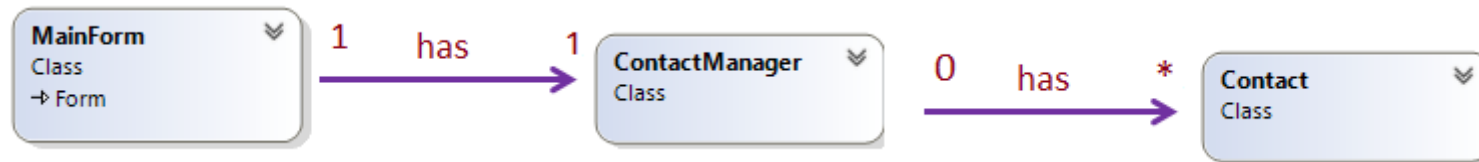
No. of registered records: 2

Simpsson, Apu	Simpsson North Avenue	12 456	Malmö	United States of America
Lundström, Nisse	Lundgatan 32	12 345	Kalmar	Sverige



### 3. Requirements for a Pass grade:

3.1 Write at least three classes as in the class diagram below:



3.2 Use a **List<T>** (or an **ArrayList**) for the registry.

3.3 The registry should be handled in a separate class (**ContactManager**).

3.4 Write a class **Address** for handling addresses. The **Address** class should have three constructors:

3.4.1 One default constructor.

3.4.2 One constructor with 3 parameters (for street, zip code and city).

3.4.3 One constructor with 4 parameters (street, zip code, city, and country).

3.4.4 **Chain calling**: The constructor with 3 parameters should call the one with 4 parameters (using the keyword **this**). This way of methods calling each other (to avoid rewriting) same code is called "chain calling". We implement chain calling on our constructors, but remember that this requires a special syntax (see the Help section later in this document).

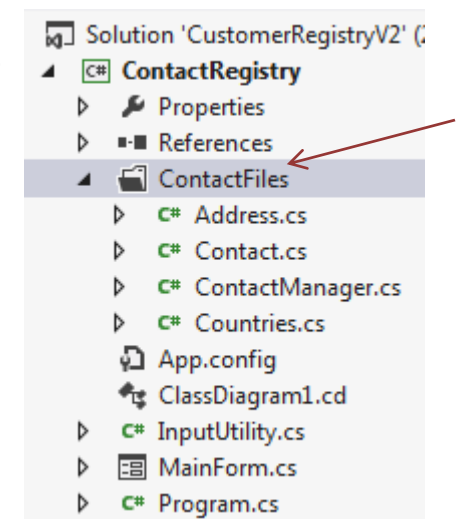
3.4.5 Override the **ToString** method so the method returns a string formatted with values saved in the address object.

3.5 The **Contact** class should maintain data about the first name and last name as well as the address of a person.

3.6 Override also the **ToString** method in the **Contact** class. The method should format a string with the first and last name and the address of the contact. For the address part, make use of the **ToString** of the **Address** class.

3.7 The **ContactManager** should be responsible for adding, deleting, changing and all other operation on the contact list. However, it should use the services (call methods) of the **Address** class.

3.8 All instance variables in all classes are to be **private** (as before) and access to them should be handled via properties. However, do not write a property that returns an array or a list (e.g. the contact list in the **ContactManager** class). Write instead a method that returns an element of the list saved in a given position.



- 3.9 Organize your contact files in a separate folder in VS (see the previous figure). VS will create the same file structure on your computer too when you save the project.
- 3.10 Provide a brief documentation of every method and every class with XML-compatible comments (using ///).
- 2.1 Do not forget to write your name on top of each file, please!

## HELP AND GUIDANCE

In the sections that follow, you are given some code snippets as screen dump images. It is very important that you understand every step and not just imitate the code. Use the forum in the module to ask questions and discuss the things you are not sure about.

### 4. The Address Class

An overview of the Address class is shown here.

Note: the comments are deleted from the code images to save space. Don't forget to comment all methods of the classes as well as the classes themselves, using XML-compatible comments (with ///), to avoid resubmission.

```
public class Address
{
    private string m_street;
    private string m_zipCode;
    private string m_city;
    private Countries m_country;

    /// <summary>
    /// Default constructor calling another constructor in this class.
    /// </summary>
    public Address() : this(string.Empty, string.Empty, "Malmö")...

    //Constructor - call the next constructor for reuse
    public Address(string street, string zip, string city): this(street, zip, city, Countries.Sverige)...

    //code only at one place
    public Address(string street, string zip, string city, Countries country)...

    public string Street...
    public string City...
    public string ZipCode...
    public Countries Country...

    /// <summary>
    /// This function simply deletes the '_'s from country names as saved in the enum
    /// </summary>
    /// <returns>The country name without the underscore char. </returns>
    public string GetCountryString()
    {
        string strCountry = m_country.ToString();
        strCountry = strCountry.Replace("_", " ");
        return strCountry;
    }

    /// <summary>
    /// Override the ToString method to return a string made of the address detail,
    /// formatted in one line.
    /// </summary>
    /// <returns></returns>
    public override string ToString()
    {
        string strOut = string.Format("{0, -25} {1,-8} {2, -10} {3}", m_street, m_zipCode, m_city, GetCountryString());
        return strOut;
    }
}
```

*Properties*

## 5. The Contact class

Declare instance variables for first name, last name and address. For the address, declare an instance of the Address class and create the instance in the Contact class' constructor.

Write two constructors and a **ToString** method.

```
public class Contact
{
    //things to remember - initialize strings
    private string m_firstName = string.Empty;           //can use = "" also
    private string m_LastName = "";

    //Aggregation - "has a"
    private Address m_address;

    //create the m_address object in the constructor
    public Contact()...

    public Contact(string firstName, string lastName, Address adr)...
```

---

```
    //Properties
    public Address AddressData...

    public string FirstName...

    public string LastName...

    public string FullName
    {
        get { return LastName + ", " + FirstName; }
    }

    /// <summary>
    /// Format the the contact details. Call the address' ToString
    /// </summary>
    /// <returns></returns>
    public override string ToString()
    {
        string strOut = string.Format("{0, -20} {1}", FullName, m_address.ToString());
        return strOut;
    }
}
```

## 6. The ContactManager class

This class is a container class for **Contact** objects. It saves contact objects in a dynamic list of the type **List**. Notice how the list is declared in the class and created the constructor of the class.

The class has methods for adding a new contact, deleting and changing an existing contact. Have a close look at the method **GetContactInfo** at the end of the image. This method prepares an array of strings, where every string is made up of information about the contact object. This method can be called from the **MainForm** to update the **ListBox**.

```
public class ContactManager
{
    private List<Contact> m_contacRegistry;

    public ContactManager()
    {
        m_contacRegistry = new List<Contact>();
    }

    public int Count...
    public bool AddContact(string firstName, string lastName, Address addressIn)...
    public bool AddContact(Contact ContactIn)...

    public bool ChanngeContact(Contact contactIn, int index)...

    public bool DeleteContact(int index)...

    public Contact GetContact(int index)
    {
        if (index < 0 || index >= m_contacRegistry.Count)
        {
            return null;
        }
        else
        {
            return m_contacRegistry[index];
        }
    }

    //return an array of strings where every string represents a contact.
    public string[] GetContactsInfo()
    {
        string[] strInfoStrings = new string[m_contacRegistry.Count];

        int i = 0;
        foreach (Contact contactObj in m_contacRegistry)
        {
            strInfoStrings[i++] = contactObj.ToString();
        }
        return strInfoStrings;
    }
}
```

## 7. The MainForm class

Sketch your GUI as suggested by the sample run example presented at the beginning of the document.

Double-click on the **ListBox** in VS at design time. VS will prepare the method for the **SelectedIndexChanged** event of the **ListBox**. All code you write in this method will be automatically called when the user highlights an entry in the **ListBox** (clicks on a row in the **ListBox**). What you need to do here is to get the information related to the entry from the registry (**ContactManager** object) and fill the input boxes with the data to make it easier for the user to edit.

In the code example here, the method **UpdateContactInfoFromRegistry()** is called to accomplish the task. Make sure that you understand the code written in the mentioned method.

```
public partial class MainForm : Form
{
    private ContactManager m_contacts; //contact registry

    public MainForm()
    {
        InitializeComponent();

        //create an instance of the contact registry
        m_contacts = new ContactManager();
        InitializeGUI();
    }
    //Add Countries to the combo box and do other initializations.
    private void InitializeGUI()...
    private bool ReadInput()...
    private bool ReadName(out string firstName, out string lastName)...
    private Address ReadAddress()...

    private void btnAdd_Click(object sender, EventArgs e)
    {
        if (ReadInput())
            UpdateGUI();
    }
    //Clear the ListBox, call the method GetContactsInfo of the m_contacts
    //object and using the ListBox's AddRange method, add the array to ListBox
    private void UpdateGUI()
    {
        lstResults.Items.Clear();
        lstResults.Items.AddRange(m_contacts.GetContactsInfo());
        lblCount.Text = lstResults.Items.Count.ToString();
    }

    private void lstResults_SelectedIndexChanged(object sender, EventArgs e)
    {
        UpdateContactInfoFromRegistry();
    }
    //Fill the input boxes with information for the highlighted contacts in the
    //ListBox
    private void UpdateContactInfoFromRegistry()
    {
        Contact contact = m_contacts.GetContact(lstResults.SelectedIndex);

        cmbCountry.SelectedIndex = (int)contact.AddressData.Country;
        txtFirstName.Text = contact.FirstName;
        txtLastName.Text = contact.LastName;
        txtCity.Text = contact.AddressData.City;
        txtStreet.Text = contact.AddressData.Street;
        txtZipCode.Text = contact.AddressData.ZipCode;
    }
}
```

## 8. ArrayList och List<T>

**ArrayList** and **List** are two data structures (two collections) that are defined in the .NET Framework. Both are objects and are created in the same way as other objects:

```
ArrayList myArrayList = new ArrayList();  
List<T> myList = new List<T>();
```

where **T** in the case of using **List** can be any type of object, a primitive type such as **int**, **double**, **bool** or classes such **Car**, **BankAccount**, **RealEstate**, etc.

### Which one to use?

**List** and **ArrayList** share similar syntax and both are easy to work with. An **ArrayList** can have objects of different types while a **List <T>** cannot. Therefore, **List** should be preferred when working with objects of the same type. **List <T>** is faster and easier to use. When declaring and creating an object of **List <T>**, you provide the type of the object it is going to contain by replacing the **T** with the class name.

```
private List<Contact> m_contactRegistry;
```

An **ArrayList** is declared and created as follows:

```
private ArrayList m_contactRegistry; //type of elements not known!
```

An **List<T>** object is created as follows:

```
m_contactRegistry = new List<Contact>();
```

**Question:** what do the parentheses before the semicolon are supposed to denote?

An **ArrayList** object is created as follows:

```
m_contactRegistry = new ArrayList();
```

Both of the list types have the **Add**, **Remove**, **RemoveAt**, **Insert**, **Count** and several methods and properties that are used exactly in the same way. However, when fetching an element from an **ArrayList** object, you must use casting so the compiler knows what type of object is saved in the element. The following code will work with a **List<T>** without any problem but it will cause a compiler error when used with an **ArrayList**.

```
return m_contactRegistry[index];
```



When using an ArrayList object, you must write as follows:

```
return (Contact)m_contactRegistry[index];
```

The table below outlines some of the most useful methods that are common to both of the types.

Add a new object to the list (at the end of the list)	Add()
Remove an object from the list	Remove()
Insert an element at a certain position	Insert()
Remove all elements	Clear()
Info on the number of elements in the list	Count
Access an element using	[ ]

Remember that every time an element is to be saved in the list, it must have been created (somewhere). To avoid abnormal program termination, make sure to check always the validity of an object as in the example below:

```
if (contactIn != null)
{
    //code
}
```

## 9. Accessing Array elements (an advanced discussion – can skip it)

Other classes like **MainForm** would of course need to access data stored in the **m\_contactRegistry** of its **ContactManager** object. Because **m\_contactRegistry** is (and definitely should be) declared as **private**, 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 m_contactRegistry ‘‘Bad programming – (not an accepted way in solving this assignment)
```

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

```
return m_contactRegistry[index];    ‘(accepted in solving this assignment)
```

This method is better but still not very safe because of the fact that we are passing the address of our object (a **Contact** object saved in the position index) in the memory. However, if the object's data is protected by declaring them **Private** and access to the data is provided through **set** and **get** methods (as in our case), there is a degree of safety. A better and safer way is to return a copy of an object.

- 1.1.1 Create a new object of the **Contact** type
- 1.1.2 Copy all fields from the original object (**m\_contactRegistry[index]**) to this copy object
- 1.1.3 Return the copy object.

```
public Contact GetContactCopy(int index)
{
    Contact origObj = m_contactRegistry[index];

    //Create a copy of the contact element saved in position = index
    Contact copyObj = new Contact(origObj.FirstName, origObj.LastName, origObj.AddressData);

    //return the copy
    return copyObj;
}
```

To make the problem a little more complicated but make life easier, copying an object can be effectively done by using a so called **copy constructor**. A copy constructor can be used to initialize an object with values of another object of the same type. Although you don't need to do that to solve this assignment, it is undoubtedly going to pay back once you have learned how to use it. It is good to make it a habit to write a copy constructor to every new class that you write. The code below shows how a copy constructor may look like.

```
//Copy constructor
public Address(Address theOther)
{
    m_street = theOther.m_street;
    m_zipCode = theOther.m_zipCode;
    m_city = theOther.m_city;
    m_country = theOther.m_country;
}
```

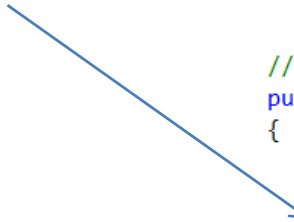
The above constructor can be called as follows:

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

All field members (as programmed in the copy constructor) of the object address1 are copied to the corresponding members of address2.

A copy constructor in the Contact class may be written is shown here:

**Note:** The Address class' copy constructor is called in the last statement.



```
//Copy Constructor
public Contact(Contact theOther)
{
    this.m_firstName = theOther.m_firstName;
    this.m_LastName = theOther.m_LastName;
    this.m_address = new Address(theOther.m_address);
}
```

## 10. Submission

Upload your project to Its Learning as in earlier assignments.

Good Luck!

**Farid Naisan,**  
Course Responsible and Instructor